Master thesis : Development of a Flutter module for ATHLETin

Auteur : Bulut, Stephan
Promoteur(s) : Mathy, Laurent
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en informatique, à finalité spécialisée en "computer systems security"
Année académique : 2023-2024
URI/URL : http://hdl.handle.net/2268.2/19541

Development of a Flutter module for ATHLETin

Thesis realized with the aim of obtaining the Master's degree of Computer Science Engineering

STEPHAN BULUT

Supervisor :

LAURENT MATHY

UNIVERSITY OF LIÈGE
FACULTY OF APPLIED SCIENCE
Academic year 2023-2024

Abstract

In the sphere of sports and athlete management, injuries often arise due to inadequate information and communication channels. *ATHLETin* presents itself as an innovative solution designed to tackle this challenge. The project originated from the mind of Julien Paulus, who believes that a better communication results in fewer injuries. In a nutshell, *ATHLETin* is a versatile mobile and web-based application that modernizes athlete management for trainers and health specialists.

ATHLETin is divided into several modules, each contributing a part of the solution : a calendar module for scheduling athlete-specific events, a training module for organizing training session, a medical module for handling athletes' health tracking, and an administration module for allowing authorized members to access and manage pertinent data (including athlete lists and their responses to questionnaires). Furthermore, a communication module is available to allow members to communicate with each other. This thesis focuses the development of a system for *Roles, Affiliations,* and *Groups* to complement the administration module.

The Role system, Affiliation system, and Group system are imperative to enable members to create and access only the data they are authorized to manage. Consequently, the implementation of the administration module must provide excellent performance while being easy to integrate into the current project. To achieve this goal, the implementation takes three key approaches. Firstly, the implementation minimizes requests through an efficient implementation design of the administration platform. Secondly, we reduce the backend workload by implementing efficient queries and minimizing the number of tables involved on the server and database sides. Lastly, the implementation leverages `Flutter` technologies efficiently by managing component states to target only essential rebuilds and thus avoid unnecessary reconstruction.

Regarding the components of the high-level architecture, the project is composed of a mobile application and a web application developed using the `Dart` language within the `Flutter` framework, which forms the frontend of this project. Additionally, it includes a `REST` server implemented in the `Go` language, along with a `PostgreSQL` database, which forms the backend of this project.

This work constitutes an integral component of the larger *ATHLETin* project initiated by Professor Laurent Mathy and his team. Consequently, these modules may see additional enhancements in the future. The principal objective was to offer an efficient solution that aligns perfectly with the ongoing project, facilitating future integration.

Acknowledgement

I would like to begin by expressing my sincere gratitude to my family and friends for supporting me on this journey. I extend my special thanks to my mother, who has been by my side throughout my entire academic journey and has never stopped believing in me to this day.

Next, I would like to express my gratitude to Professor Laurent Mathy and Gaulthier Gain for not only allowing me to immerse myself in a captivating project in the field of Computer Science but also for being patient and understanding, especially in the initial stages. I am thankful for your guidance, which was conveyed through detailed feedback and transparent communication. I genuinely believe that this project has contributed significantly to my personal growth.

It is also essential to acknowledge that this thesis would not have been possible without the valuable teachings of the faculty members in the Faculty of Applied Sciences. I express my deep gratitude to the professors and teaching assistants I have encountered during my academic career.

Finally, I would like to dedicate a special thank you to my three fellow engineering comrades Maxime, Cédric and Jean-David, who have supported and assisted me more than anyone else by sharing a part of this journey with me. They have given me courage and support through their presence, jokes, and shared struggles. None of this would have been possible without them.

Contents

Part I

Introduction

Chapter 1

Introduction

The introduction provides the history of *ATHLETin* from the inception to its current form. Indeed, before delving into this thesis, it is important to establish the general context of the *ATHLETin* project to fully understand its objectives and constraints. This will be followed by a presentation of the key modules and the challenges encountered.

1.1 Origin story of *ATHLETin*

While *ATHLETin* was officially created in 2020, its origins trace back a decade earlier. Its creator, Julien Paulus, was at the time the physical trainer at the rugby training center in the Sart-Tilman. He was responsible for the physical training of several young athletes and had to regularly prepare training sessions. However, these athletes trained not only at the Sart-Tilman rugby center but also in their respective teams. Additionally, some of them might have had training sessions with the national team. Considering these factors, it was often challenging to provide training sessions that met the different athletes' expectations. The problem was clear: *how to manage and follow these various athletes in the best possible way ?*

In order to respond to this challenge, an initial solution was developed. A co-worker of Julien Paulus created a web application called *MyLBFR*, designed to facilitate communication between coaches and athletes. Unfortunately, the *MyLBFR* project was later abandoned due to a lack of funding.

However, despite this first setback, Julien Paulus, convinced of the great potential of his idea, persisted. The initial solution met the demand for smoother communication and athlete management. This was enough to convince Mr. Paulus to continue the project. With the assistance of Xavier Picard, a specialist in sports organization management and marketing, he entered into a collaborative agreement with Professor Laurent Mathy and his team for the development of *ATHLETin* at the University of Liège. Currently, *ATHLETin* is being developed by Laurent Mathy, his team, and a few students, including myself.

1.2 Functionalities

The primary goal of *ATHLETin* is the comprehensive management of athletes by coaches and healthcare professionals. To achieve this, the application is divided into several modules, each of which partially contributes to addressing the central challenge.

1.2.1 Calendar Module

The calendar module, or agenda module, simply provides an interconnected agenda for coaches and healthcare professionals with athletes. It enables athletes and managers to handle events. In this way, athletes can view events scheduled specifically for them and notify about potential absences. On the other hand, managers can create events, reschedule them, add athletes, and track scheduled events for a particular athlete.

1.2.2 Training Module

The training module is highly significant within the application. It allows athletes to provide feedback after each training session by responding to dedicated questionnaires. This module also facilitates the display and analysis of questionnaire results based on the collected data. These data are processed through semi-automatic algorithms that analyze the results and provide a detailed view of athletes' performance.

1.2.3 Medical Module

As mentioned earlier, a fundamental principle behind *ATHLETin* is that increased communication leads to fewer injuries. The medical module handles athletes' health tracking, enabling players to precisely describe their injuries when they are injured, with the aim of quickly obtaining the best possible diagnosis. This module allows for quick access to the context and dates of injuries sustained by athletes. It also informs which events will be missed by the injured athlete. In addition to effectively diagnosing injuries, healthcare personnel responsible for athletes can communicate this information directly to coaches through the application. Finally, the application will enable healthcare professionals to schedule medical appointments with the injured athletes [1].

1.2.4 Administration Module

The administration module essentially allows administrators to perform all the tasks mentioned earlier. By administrators, we mean coaches and healthcare professionals responsible for athletes. Administrators can:

- View and manage calendars

- Manage training sessions and attendance records
- Manage and communicate with athletes and members
- Export data in formats like PDF, Excel, or CSV [2]

Furthermore, the administration module provides members with the ability to administer different permissions and access based on their roles. This is the Role system, which offers complete flexibility in representing the hierarchy within a sports organization. For example, a healthcare staff member of a club could have different permissions than a secondary coach or an assistant.

The application also features an Affiliation and Group system, allowing administrators and athletes to be grouped according to the team or group they belong to. This affiliation and group system is essential for maintaining data privacy within *ATHLETin*. **This module is the one implemented for this thesis.**

1.2.5 Communication Module

Simply allows members to communicate with each other through a traditional messaging service.

1.3 Challenges to overcome

As the application needs to be user-friendly and ergonomic, it requires an efficient implementation design. Since the project was initiated by Professor Laurent MATHY and his team, I needed to reuse the technologies chosen at the beginning of the project to facilitate future integration. Consequently, the application is implemented in `Dart` using the `Flutter` framework. The team also opted for a `REST` server implemented in `Go` and a `PostgreSQL` database to ensure quality maintenance in the future.

On the administrator's side, web platforms were developed for each module. A mobile application was also created for athletes to access their calendars and respond to questionnaires. During the development process, certain choices made at the project's outset had to be revisited to achieve better scalability and compatibility. One of the challenges addressed in this thesis is the reimplementation of the administrator module to accommodate various changes, including the addition of Role, Affiliation and Group systems. This module has the particularity of encompassing all other modules of the admin platform, and therefore had to be developed very conscientiously. It was also essential to maintain this implementation's flexibility for future changes.

—

This introduction provides an overview of the *ATHLETin* project's context and its objectives. Part two will delve into the details of the technologies used for the backend and frontend.

Part II

Technical Background

Chapter 2

Backend

The backend refers to the behind-the-scenes components of code that are not accessible to the end-user of an application or software, but which enables its functionality. In other words, the backend represents all the parts that the user does not see but serves to execute actions on a computer system or application. In the case of this project, it corresponds to the database and the `REST API`. This chapter outlines the various technologies used for the backend of the *ATHLETin* project.

2.1 PostgreSQL

`PostgreSQL` [3] is a powerful open-source relational database management system (`RDBMS`) used for organizing, storing, and managing structured data. Its origins date back to 1986 as part of the `POSTGRES` project at the University of California, Berkeley, with over 35 years of active development on the main platform.

The main features of `PostgreSQL` are:

- Free & open-source : allows many businesses and organizations to use it without licensing costs, which can help reduce support and maintenance expenses
- Robust security : has high reliability with data integrity assurance allowing to handle sensitive data
- Strong performance & scaling : allows to handle complex workloads and advanced queries and handle index management and replication
- `SQL` standards compliance : simplifies application development and database management while providing a recognized and accepted framework within the industry
- Highly extensible : allows to customize and extend the database to meet the specific needs of the application without fundamentally altering it

`PostgreSQL` is widely used across various applications, including mobile apps, websites, and enterprise management systems, especially for data analysis. `PostgreSQL` was therefore chosen for *ATHLETin* due to its perfect alignment with the project's requirements.

2.2 Go

The Go language [4], also known as Golang, is an open-source programming language developed by Google. Its origins date back to 2007 when three engineers, Rob Pike, Robert Griesemer, and Ken Thompson, frustrated by the complexity of C++ and the lack of a simple language that allowed efficient compilation and execution, started to design a new language. Go addresses several needs, such as simplicity, concurrency, and performance for network application development, system programming, and the creation of highly concurrent software.

The main features of Go are:

- Simple & readable syntax: contributes to making the programming language more user-friendly, efficient, and conducive to creating high-quality software
- Native concurrency management (Goroutines): simplifies concurrent programming and allows for better resource utilization
- High performance: enables efficient handling of large volumes of data and traffic
- Garbage Collector (GC): reduces memory-related errors and improves security
- Static typing system: helps developers detect and prevent errors at an early stage while improving code readability
- Module system for dependency management: assists developers in efficiently managing dependencies and enhancing version control
- Rich standard library: enables Go developers to enhance their productivity

Go is also known for its fast compilation, making it a popular choice for high-performance application development. Additionally, another driving factor for using Go is the GORM framework, which simplifies interactions with databases. For all these reasons, Go was chosen to implement the REST server for the *ATHLETin* project.

2.3 GORM

GORM [5] is an Object-Relational Mapping (ORM) framework for the Go programming language. It allows interaction with relational databases in a high-level, object-oriented manner. An ORM is designed to simplify database operations, enabling developers to work with databases without writing complex SQL queries and handling low-level interactions with the database.

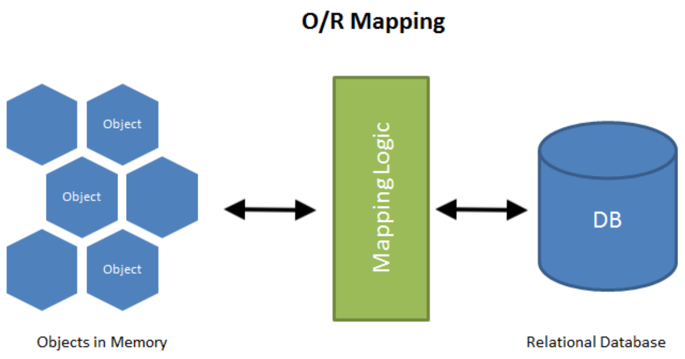


Figure 2.1: Object-Relational Mapping (ORM) [6]

The main features of GORM are:

- Modeling: allows to define Go structure types that represent database tables (these structure types can include fields corresponding to columns in the database)
- Abstraction: provides methods for creating, reading, updating, and deleting records in the database without the need to write raw SQL queries
- Complex query building: enables the construction of complex database queries using a chainable API
- Auto-Migration: greatly simplifies the process of updating the database schema as your application evolves
- Open-source & active: open-source and the libraries are continually improved, with bugs being addressed promptly and effectively
- Pluggable dialects: can work with various relational databases such as PostgreSQL, MySQL, SQLite, and others

Overall, GORM simplifies the process of working with databases in Go applications, making it a popular choice for developers looking to build database-based applications with Go. It's for these reasons that the GORM framework was chosen for the *ATHLETin* project.

2.4 REST API

First, it's necessary to define what an API is. An Application Programming Interface (API) is a set of definitions and protocols that serve as a facade through which software provides services to other software. It can be considered as a contract between an information provider and a user, defining the content requested by the consumer (the request) and the content provided by the producer (the response).

A REST API is an Application Programming Interface that adheres to the constraints of the RESTful architectural style [7]. In other words, it's an API based on the Representational State Transfer (REST) architectural model, which emphasizes simplicity, scalability, and performance of web services.

The key constraints defined by a REST API are:

- Uniform interface: resources are identified by Uniform Resource Identifiers (URIs), and standard methods (GET, POST, PUT, DELETE) are used to interact with these resources
- Client-server architecture: maintains a clear separation between the client and the server, allowing them to evolve independently
- Stateless communication: the server does not retain information about the client's state between requests, improving reliability and scalability
- Cacheable: responses indicate whether they can be cached by the client, improving performance by reducing the server's load
- Layered system: A REST API may consist of multiple layers (e.g. proxies or firewalls) between the client and the server. Each layer only needs to know about its adjacent layers, enhancing system flexibility and extensibility.
- Representation: resources can have different representations (e.g., XML or JSON), and clients can negotiate the representation they prefer

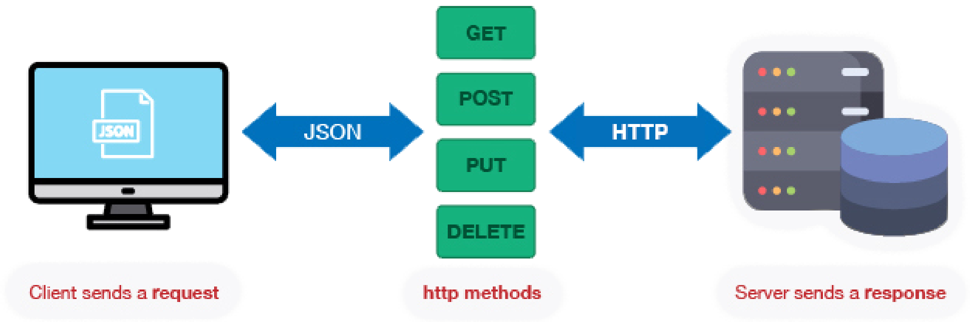


Figure 2.2: RESTful API [8]

These constraints help in designing simple, scalable web APIs suitable for various use cases. In summary, a REST API simplifies communication between clients and servers, exposing resources via URLs and standardizing operations through HTTP methods.

2.5 Postman

In addition to everything mentioned above, we will also be using `Postman`. `Postman` [9] is a tool used for `API` development and web service testing. Essentially, it allows developers to create, test, document, and share `APIs` in an efficient and clear manner.

—

This chapter has thoroughly reviewed the tools used for the backend of the *ATHLETin* project. The choice of the `Go` language, along with the `GORM` framework, and a `PostgreSQL` database was made with the goal of maintaining a performant and reliable `REST` server.

The next section will delve into the details of the technologies used to implement the application's frontend.

Chapter 3

Frontend

The frontend is the part of a software or an application that the end-user directly interacts with. It is composed of the user interface, design, and user experience elements that enable users to interact with the system. In simple terms, the frontend is what the user sees and interacts with when using an application.

For the *ATHLETin* project, the frontend includes the user interface components, such as the calendar, training module, medical module, administration module and messaging system. This chapter will delve into the details of the technologies and design considerations used in implementing the frontend of the *ATHLETin* project.

3.1 Dart

Dart [10] is a programming language created by Google in 2011. It is an open-source language primarily used for mobile and web application development. Dart is a versatile programming language known for its readable syntax, active community, good performance, and close association with the `Flutter` framework for cross-platform mobile application development. Dart can be summarized as *a client-optimized language for fast apps on any platform*. Let's now explore the specifics of this language.

3.1.1 Type safety

In a programming language, type safety is an essential concept aimed at ensuring that data types are used correctly and consistently. Type safety in Dart means that the language is designed to detect type errors at compile time, catching potential issues even before the program is executed.

While Dart does offer dynamic typing features for situations where flexibility is needed, it is primarily a statically typed language. This means that the type of each variable is determined at compile time, enabling the detection of type inconsistencies during the development phase, before the code is executed.

Furthermore, Dart allows developers to use type inference, allowing the compiler to deduce a variable's type from its value. This makes the syntax more concise while still maintaining type safety.

Lastly, Dart performs type verification to ensure that operations on variables conform to the defined typing rules. If an operation violates these rules, a compilation error is generated.

In summary, Dart's type safety helps reduce type-related programming errors, making development more reliable and facilitating code maintenance. It enables developers to catch typing issues early in the development process, which is particularly important for building robust applications like *ATHLETin*.

3.1.2 Null safety

Null safety is another important concept in programming which purpose is to avoid errors related to null values in the code. In fact, null values often lead to exceptions and bugs in many programming languages.

In Dart, null safety involves explicitly declaring whether a variable can hold a null value or not [11]. Two types of variables exist in this context: Non-nullable and nullable. Non-nullable variables cannot hold null values. Developers must assign a value to them at the time of variable declaration, and the variable cannot hold a null value after this assignment. On the other hand, nullable variables can contain null values and are declared with a "?" after the variable's type.

Null safety in Dart offers several advantages:

- Error reduction: explicitly handling null values eliminates many errors related to accessing null values during the development phase

- Improved performance: knowing that certain variables will never be null, Dart can optimize the code for better performance

- Readable code: null safety improves code readability by clearly expressing the developer's intentions regarding null value handling

- Enhanced documentation: indicating which variables can be null and which cannot enhances code documentation

In summary, null safety is a significant concept in Dart that aims to enhance code reliability, performance, readability, and documentation by ensuring that null values are explicitly managed. Null safety helps reduce bugs related to null values and improves code quality by preventing errors that result from unintentional access of variables set to null.

3.1.3 Garbage Collection

Garbage Collection (`GC`) is a form of automatic memory management that is essential in programming languages.

In Dart, Garbage Collection [12] is used to automatically free memory and manage objects that are no longer in use. It reclaims memory that was allocated by the program but is no longer referenced (such memory is called garbage).

Here are the key points about garbage collection in Dart:

1. First, garbage collection identifies and collects unused memory within the program. It is responsible for releasing memory occupied by objects that are no longer accessible within the program. This mechanism prevents memory leaks, where memory would gradually be consumed by unused objects.

2. In Dart, developers typically don't need to worry about memory release because memory management is automatic. Garbage collection automatically identifies and cleans up unused objects.

3. Garbage collection significantly improves performance and reduces programming errors related to incorrect memory management. In addition to being user-friendly, automatic memory management enhances performance by maintaining efficient memory utilization.

In summary, garbage collection in Dart is a crucial mechanism for memory management that allows developers to create without the need for manual memory management. It enhances the safety, reliability, and efficiency of Dart applications by eliminating potential memory leaks. It is also compatible with null safety, meaning it correctly manages null objects.

3.1.4 Asynchronous programming

A synchronous operation blocks other operations from executing until it completes. In contrast, asynchronous operations allow a program to continue its work while waiting for another operation to complete.

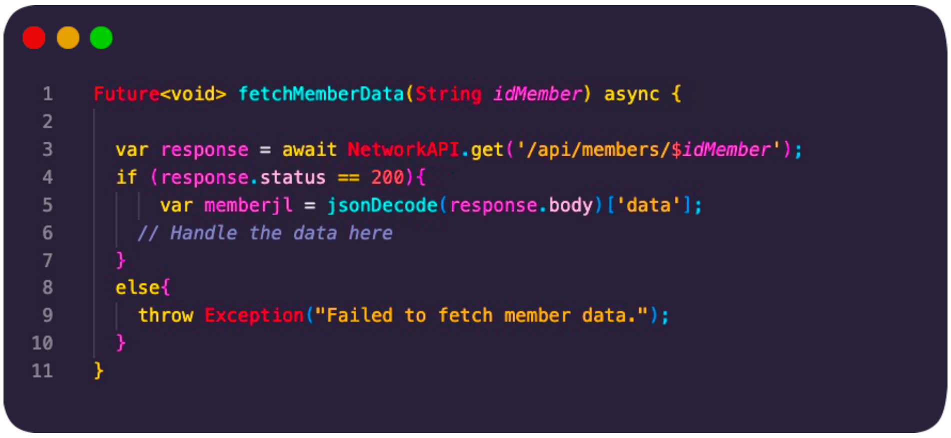
In Dart, asynchronous operations enable tasks to be performed in parallel without blocking the execution of the main program [13]. This is crucial for applications that need to carry out long-duration operations, such as fetching data from a network, reading a file or writing to a database.

Dart's asynchronous mechanism relies on three key elements: `futures` and `async/await`.

A `Future` is an object that represents a value that may not yet be available. A `Future` allows the program to proceed with its execution without waiting for the operation to finish. When an asynchronous function is called, it usually immediately returns a `Future` and then continues its execution in the background.

The `await` keyword is used to await the completion of a `Future`. When an asynchronous function employs the `await` keyword, it means that the function suspends its execution until the `Future` is complete. However, during this time, the main thread can continue to execute other tasks. The `async` keyword is used to indicate that the function is asynchronous, containing asynchronous code.

Here's a simple example:



```
Future<void> fetchMemberData(String idMember) async {

  var response = await NetworkAPI.get('/api/members/$idMember');
  if (response.status == 200){
      var memberjl = jsonDecode(response.body)['data'];
    // Handle the data here
  }
  else{
    throw Exception("Failed to fetch member data.");
  }
}
```

Figure 3.1: Dart asynchronous operations example

In this example, the `fetchMemberData()` function is defined as asynchronous using the `async` keyword and returns a `Future`. The function makes a `GET` request using the `NetworkService` library to obtain data for the given member. Once the response is available, the code checks the response status code. If the response is successful (status code `200`), it can then manipulate the member's data. Otherwise, it throws an exception.

In summary, asynchronous operations are essential for maintaining efficient performance in Dart applications, especially web and mobile applications. They allow tasks to be carried out in the background, ensuring that the user interface remains responsive, without blocking the main program's execution.

3.1.5 Platforms

Dart's compiler capabilities [14] offer different ways to run code :

1. `Native Platform`: When developing applications for mobile and desktop platforms, Dart provides a combination of a Dart VM featuring just-in-time (JIT) compilation and an ahead-of-time (AOT) compiler to generate machine code.

2. `Web Platform`: In the context of web applications, Dart allows compilation for both development and production needs. The web compiler within Dart translates Dart code into JavaScript.

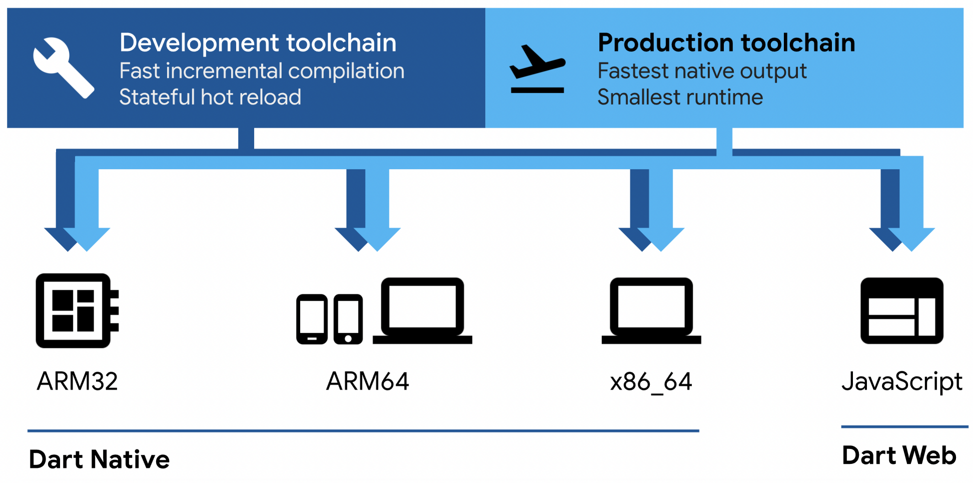


Figure 3.2: Dart Platforms [14]

Regarding `Dart Native`, a fast development cycle for iteration is crucial during development. The Dart VM offers a just-in-time compiler (JIT) with incremental recompilation (enabling hot reload), full debugging support, and real-time metrics collection (powering `DevTools`).

When applications are ready to be deployed in production, Dart's ahead-of-time (AOT) compiler can compile to native `ARM` or `x64` machine code, allowing the application to launch with a consistently short startup time.

`Dart Web` enables the execution of Dart code on web platforms powered by JavaScript. With `Dart Web`, Dart code is compiled into JavaScript, which then runs in a web browser.

Dart Web offers two compilation modes:

- An incremental development compiler that provides a quick development cycle.

- An optimizing production compiler that compiles Dart code into fast, compact, deployable JavaScript.

Regardless of the platform or the compile method used, code execution requires a `Dart runtime`. This runtime is responsible the following critical tasks:

- *Memory management*: Dart uses a managed memory model where unused memory is reclaimed by a garbage collector (GC).
- *Enforcement of the Dart type system*: While most type checks in Dart are static (at compile time), some type checks are dynamic (at runtime).
- *Isolate management*: The `Dart runtime` controls the main isolate (where code usually runs) and any other isolates created by the application.

3.1.6 Libraries

Dart offers a rich collection of core libraries [15] that serve as building blocks for a wide range of everyday programming activities such as performing mathematical computations (`dart:math`), encoding/decoding data (`dart:convert`), managing collections (`dart:core`) or asynchronous programming (`dart:async`).

3.2 Flutter

Flutter [16] is an open-source framework created by Google in 2017 with the aim of simplifying the process of developing cross-platform applications. Flutter enables developers to create high-performance applications with attractive user interfaces, ensuring a smooth user experience on various operating systems, including iOS and Android, as well as desktop, web, and embedded devices. This framework aims to enhance developer productivity, application consistency, code quality, and reduce development complexity in a modern way. Furthermore, Google provides active support and regular updates. For these reasons, Flutter has become increasingly popular, with many companies and developers adopting Flutter to create applications for mobile, web, and other platforms.

3.2.1 Architectural overview

Flutter is designed as a layered system [17], represented as a series of independent libraries, with each layer depending on the one beneath it. The architecture of Flutter consists of three key components: the Dart framework, the Flutter engine, and platform-specific embedders. No layer has privileged access to the layer below, and every part of the framework level is designed to be optional and replaceable. Here's how these different layers interact to create a Flutter application.

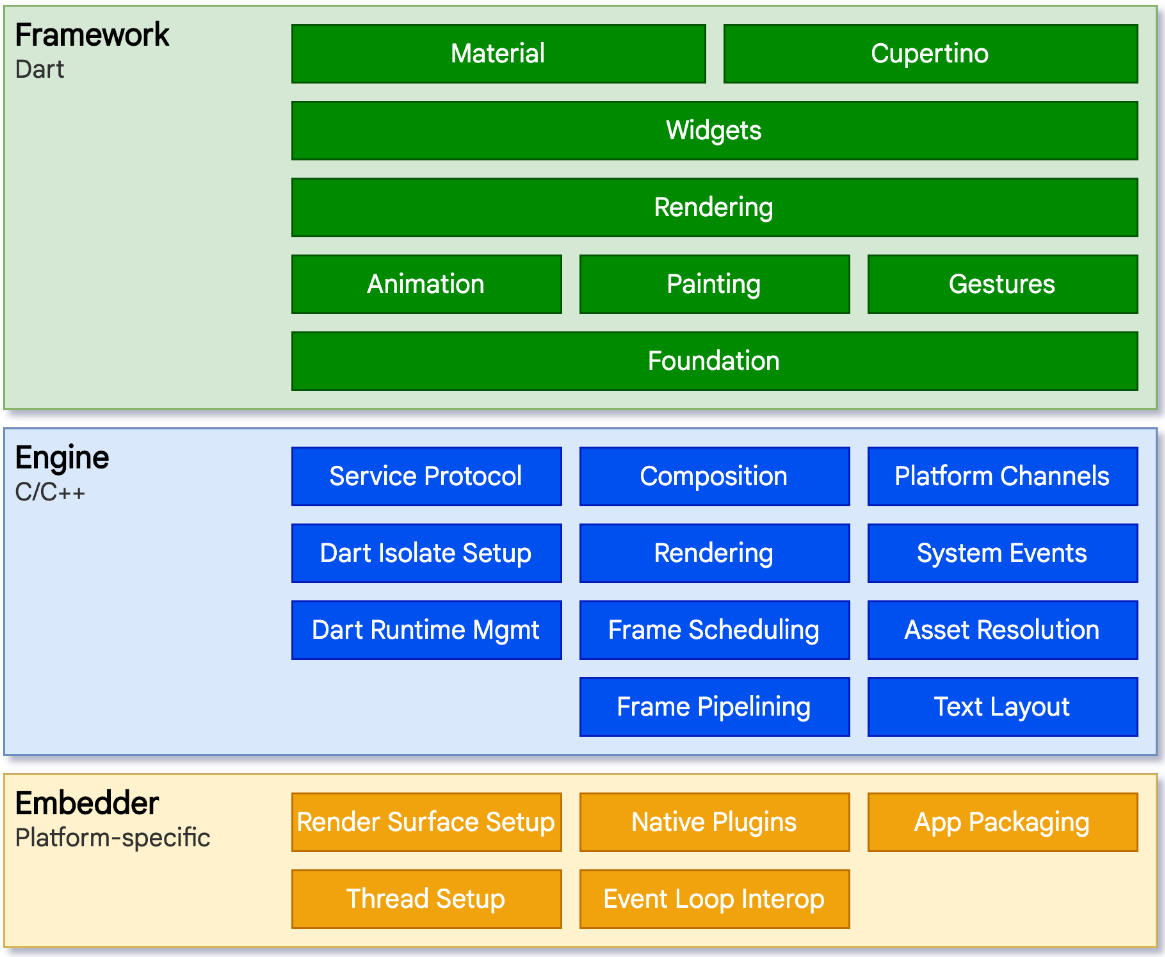


Figure 3.3: Flutter - Architecture overview [17]

Dart Framework

The Dart framework is the top layer of Flutter, allowing UI creation by providing a set of widgets, classes, and libraries. It includes pre-designed widgets for creating custom and responsive user interfaces, along with libraries for navigation, gesture management, state management, and many other features. The primary development of Flutter applications is done using the Dart framework, which is written in the Dart programming language.

Engine

The engine serves as the intermediate layer of Flutter, acting as a bridge between the Dart framework and the underlying operating systems (such as iOS, Android, macOS, Windows, Linux). It is primarily written in C++ and provides the foundation for rendering, application state management, input handling, and other essential functionalities. The Flutter engine is responsible for rasterizing user interface elements into pixels and displaying them on the screen. It manages the rendering process of Dart widgets using rendering engines like Skia and Impeller. It acts as an interface between the Dart code written in Flutter and the native services provided by platform-specific embedders.

Platform-specific Embedders

Platform-specific embedders are components specific to each platform (e.g., Android, iOS, macOS, Windows, Linux) that integrate the Flutter engine into the native environment of the platform. They provide an entry point for launching Flutter applications on a specific platform. For example, the Android embedder is written in C++ or Java and handles interactions between the Flutter application and the Android system. Embedders facilitate and streamline communication between the Flutter engine and the underlying operating system. They also handle window creation, input event management, access to native features, and more. Embedders are responsible for running Flutter applications on the target platform, both in development and production modes.

To sum up, the Dart framework enables application logic and UI development, the Flutter engine represents an intermediary for rendering and system interaction, and platform-specific embedders enable the execution of Flutter applications on different target platforms, integrating the engine into the native environment. This architecture allows Flutter to offer high performance and a consistent user experience across various operating systems.

3.2.2 Flutter web support

Regarding Flutter for the web, there are some unique characteristics in its architecture that are worth highlighting.

Dart compiles to JavaScript with a toolchain optimized for both development and production purposes. As a result, the Flutter framework, being written in Dart, was relatively straightforward to compile to JavaScript.

However, the Flutter engine, written in C++, isn't designed to interact with a web browser but rather with the underlying operating system. A different approach is required for the web. That's why on the web, Flutter offers a reimplementation of the engine using standard browser APIs. Currently, there are two options available for rendering Flutter content on the web: HTML mode and WebGL [18].

In HTML mode, Flutter uses HTML, CSS, SVG, and Canvas. In WebGL mode, Flutter uses a version of Skia compiled into WebAssembly called CanvasKit. The latter mode

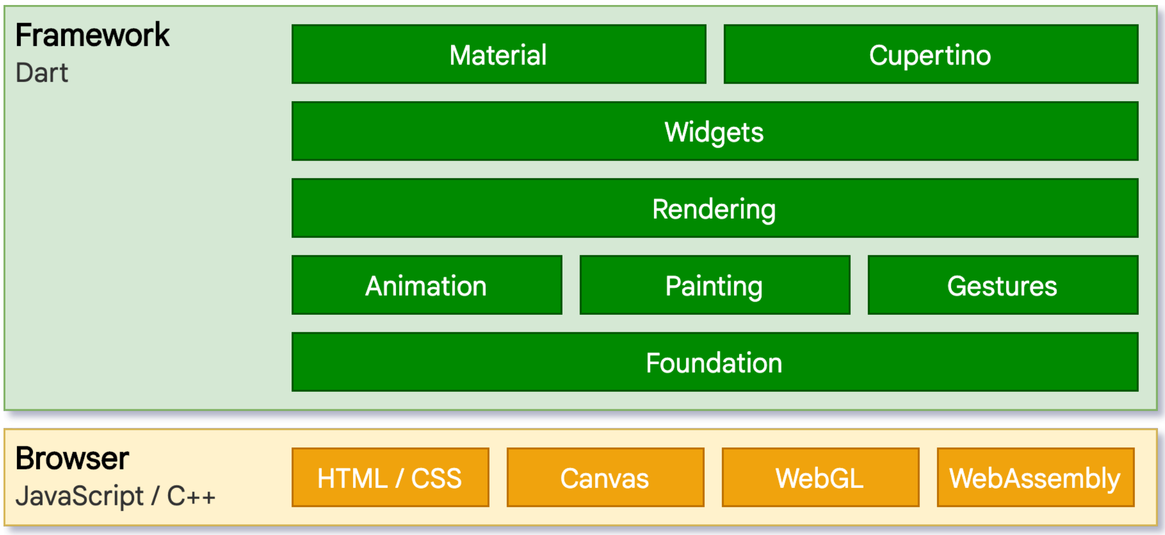


Figure 3.4: Flutter - Web architecture overview [17]

offers the fastest path to the browser's graphics stack and slightly higher graphical fidelity, while HTML mode provides the best code size characteristics.

For Flutter web, unlike other platforms on which Flutter runs, there is no need for Flutter to provide a Dart runtime. Instead, the Flutter framework is compiled to JavaScript.

During development, Flutter web uses `dartdevc`, a compiler that supports incremental compilation and allows hot restart for apps. On the other hand, when creating a production app for the web, `dart2js`, Dart's highly-optimized production JavaScript compiler, is used. It compiles the framework, the core of Flutter, and the application into a minified source file that can be deployed on any web server.

3.2.3 Widgets

A fundamental concept to grasp when learning Flutter is the idea of a building block called a "Widget". Widgets are the essential elements of constructing a Flutter app. They are the basic building blocks of a Flutter application's user interface, with each widget immutably representing a part of that user interface. Widgets define both the structure and appearance of user interface elements, such as buttons, text fields, images, and more.

These widgets are organized hierarchically in a tree-like structure, where a parent widget can hold child widgets. Each child widget is nested within its parent and can inherit the context of its parent. This hierarchical structure propagates all the way up to the root widget, where the root acts as the container that hosts the Flutter application. This root widget is typically either `MaterialApp` [19] or `CupertinoApp` [20].

The widget tree enables the creation of complex and modern user interfaces by composing simpler widgets within one another, like building blocks, to form the complete user interface. This composition and hierarchy of widgets are fundamental to building Flutter applications.

Building widgets

Widgets are typically composed of many small, single-purpose widgets that combine to produce more powerful and complex effects. Constructing a widget involves overriding its `build()` function to return a new tree of elements. The `build()` function represents the declaration of what a widget is made of, and the widget should return a new tree of widgets every time the function is called, regardless of what it returned previously.

During each rendered frame, Flutter recreates only the parts of the user interface where the state has changed by invoking the `build()` method of the relevant widget. This automated comparison process is highly efficient and enables the creation of high-performance, interactive applications. This design simplifies your code by focusing on declaring what a widget is composed of, rather than the complexities of updating the user interface from one state to another.

Stateless and stateful widget

There are two main types of widgets in Flutter: stateless widgets and stateful widgets.

Stateless widgets are immutable, meaning they do not change over time once they are created. Their content and appearance are determined solely by their own properties. Therefore, they are typically used for static elements of the user interface, such as icons, and can be easily reused in larger constructions.

Stateful widgets are considered mutable because they can change over time in response to user interactions or other factors. They maintain an internal state that can be modified. State is information that can be read synchronously when the widget is built and may change during the widget's lifetime. To update the state of a stateful widget and trigger a rebuild of the modified part of the user interface, the `setState()` method is used. This method takes an anonymous function that specifies the changes to be made to the state. After `setState()` is called, Flutter recalls the `build()` method of the widget to rebuild only that part of the user interface. Stateful widgets are essential when the part of the user interface being described can change dynamically.

Lifecycle

The lifecycle of a widget describes how a widget is constructed, updated, and destroyed over time. It is crucial to understand this lifecycle to properly manage a widget's state and optimize an application's performance. The key stages of a stateful widget's lifecycle in Flutter consist of the following points.

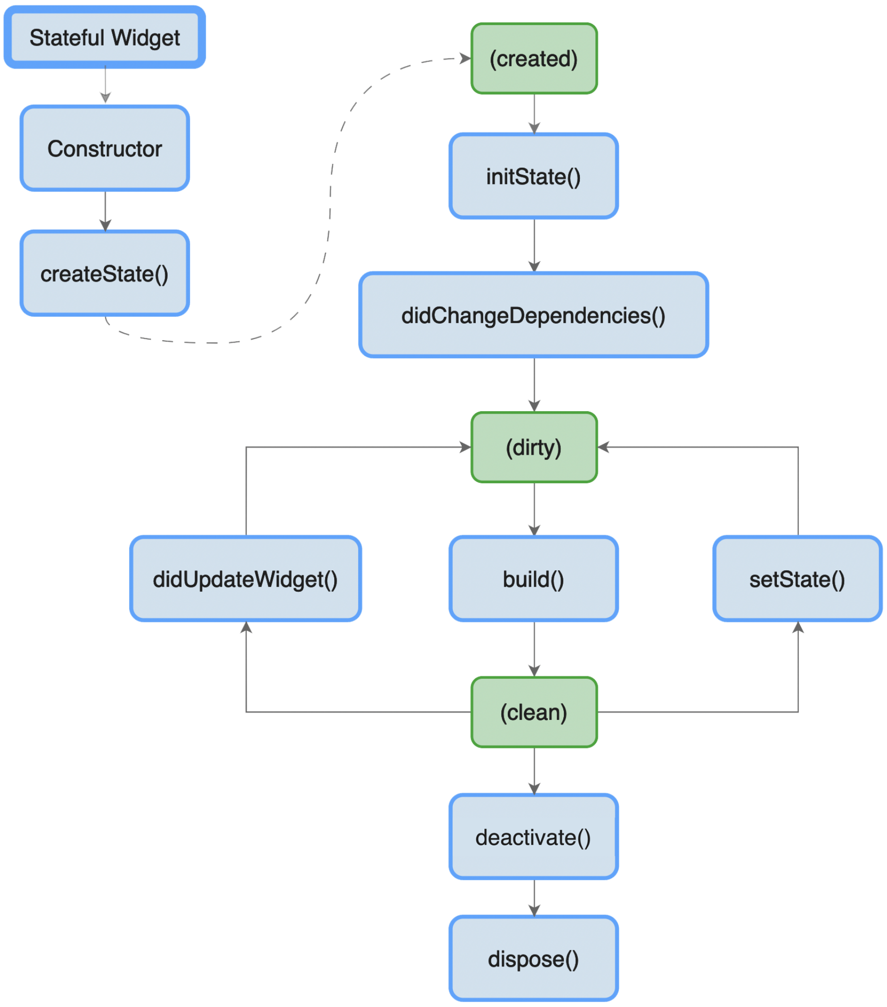


Figure 3.5: Flutter - Lifecycle of a stateful widget

1. `createState()`: The `createState` method creates a state object associated with the widget.

2. `initState()`: The `initState` method is typically used for initializations that require access to the context (such as fetching data from an external source) and is called after `createState`.

3. `didChangeDependencies()`: The `didChangeDependencies` method is called when dependency of the State object changes via InheritedWidget.

4. `build()`: The `build` method is the most crucial part of a widget's lifecycle. It is called whenever the widget needs to rebuild the associated part of the user interface.

5. `setState()`: The `setState` method is called whenever the widget's state has changed, triggering the widget to rebuild via the `build` method.

6. `didUpdateWidget()`: The `didUpdateWidget` method is called if the parent widget

of a stateful widget changes (for example, if new properties are passed). This allows the widget to adapt to these changes and update its state if necessary.

7. `deactivate()`: The `deactivate` method is called when the widget is no longer active. This typically occurs when the application is paused or when the widget is removed from the widget hierarchy.

8. `dispose()`: The `dispose` method is called when the widget is permanently removed from the widget hierarchy.

On the other hand, the lifecycle of a stateless widget in Flutter is relatively simple because these widgets do not have internal state that changes over time. It mainly involves building the UI using the `build` method during the initial creation and when reconstruction is needed due to changes in the application or the widget's parent widgets.

3.2.4 State management

After observing the existence of widgets with state, it's time to understand how to manage them. Indeed, there are various possible approaches [21] to share the application state between screens, throughout an application. However, before delving into state management, here are some details about the state in Flutter. There are in fact two conceptual types of state in any Flutter application [22].

Ephemeral vs App state

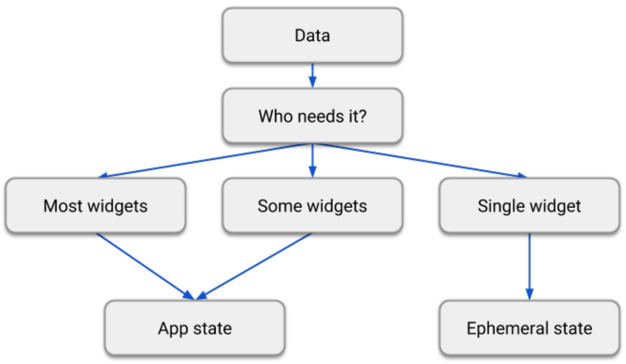


Figure 3.6: Flutter Ephemeral vs App state [22]

`Ephemeral State` (sometimes called local state) is the state that can be neatly contained within a single widget (for example, the current page in a PageView). Other parts of the widget tree rarely need to access this type of state. Since ephemeral state doesn't change in complex ways, there's no need to use state management techniques for this kind of state. Ephemeral state can be implemented within a stateful widget using `State` and `setState()`, and is often local to a single widget.

`App State` is the state that isn't ephemeral. It's shared between many parts of the application and is preserved between user sessions. For example, application state includes user preferences and login information. The choice of how to manage app state depends on the complexity and nature of the application. Now, let's explore some techniques for managing state in Flutter.

setState method

The `setState` method [23] in Flutter is a crucial method used to manage state in stateful widgets. A call to `setState` notifies the framework that the internal state of this object has changed. This then triggers the reconstruction of the widget via its `build` method, updating the user interface based on the new state. It is very useful for managing ephemeral states on a specific widget and is therefore better suited for managing ephemeral states rather than app states.

InheritedWidget

`InheritedWidget` [24] is a fundamental widget in Flutter used to propagate data through the widget tree to its descendants. It is a crucial tool for managing and sharing state in a Flutter application for several reasons. Firstly, it enables a clean and efficient propagation of state. `InheritedWidget` allows sharing data (state) with many descendant widgets without the need to pass them explicitly through constructors, keeping the codebase clean. Moreover, it offers automatic widget rebuilding. When data within an `InheritedWidget` changes, it automatically triggers the rebuilding of all descendant widgets that depend on this data. This ensures the user interface stays synchronized with the current data, guaranteeing a responsive user experience. Additionally, `InheritedWidget` provides flexible control over state management. It allows creating a "context" for specific parts of the widget tree that need access to specific data. Lastly, `InheritedWidget` is designed to be performant, avoiding unnecessary rebuilds and performance overhead.

In summary, `InheritedWidget` is a valuable tool for state management in a Flutter application as it simplifies and provides flexibility in transmitting data through the widget tree without the need for excessive repetitive code.

Provider

`Provider` [25] is a Flutter package that simplifies state management in a Flutter application. It acts as a wrapper for `InheritedWidget`, making its implementation easier and more reusable. It enables efficient state sharing and management among different widgets, streamlining the development process. Specifically, it allows data to be shared with a descendant widget without the need to pass it through intermediate widget constructors each time. Three key components make this possible.

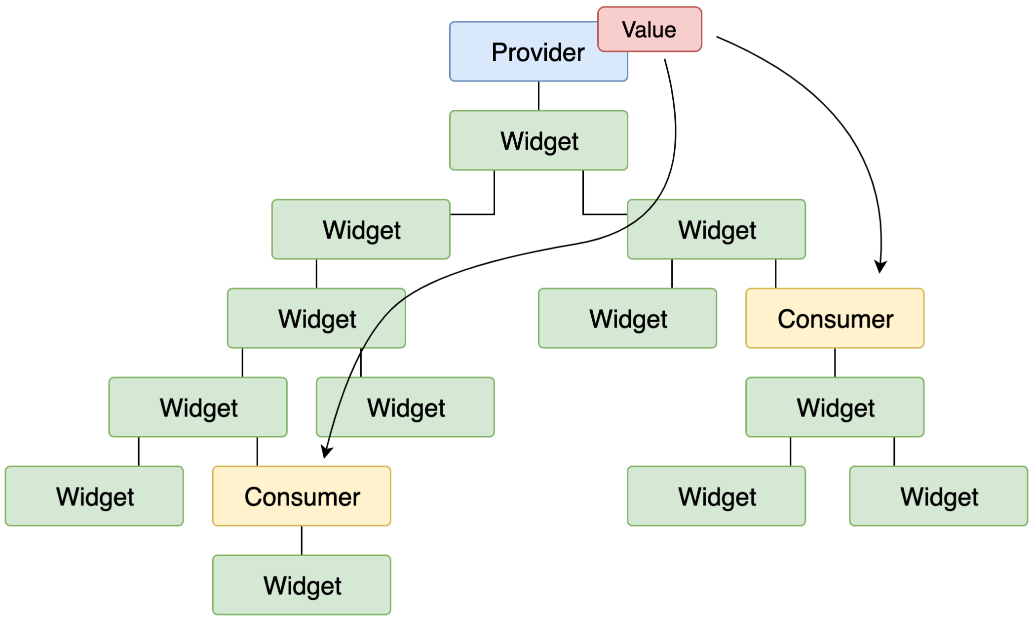


Figure 3.7: Flutter `Provider` tree

First, the Flutter `ChangeNotifier` class provides access to the `notifyListeners()` method, which is used to notify listening widgets to rebuild when the state changes. Second, the `Consumer` class enables the selective rebuilding of child widgets without affecting other widgets in the widget tree. Third, the `Provider.of` method allows descendant widgets to access the state object of the nearest `Provider` in their widget tree.

In summary, `Provider` is valuable for state management in a Flutter application because it simplifies state management by centralizing it, enhances the responsiveness of the user interface by creating reactive widgets that automatically update when the state changes, and promotes good development practices.

3.2.5 Rendering and layout

Now, let’s delve into the rendering pipeline [17], which is the sequence of steps followed by Flutter to turn a hierarchy of widgets into actual pixels painted on a screen. Flutter has a fast and straightforward pipeline for how data flows to the system, as depicted in the following sequencing diagram :

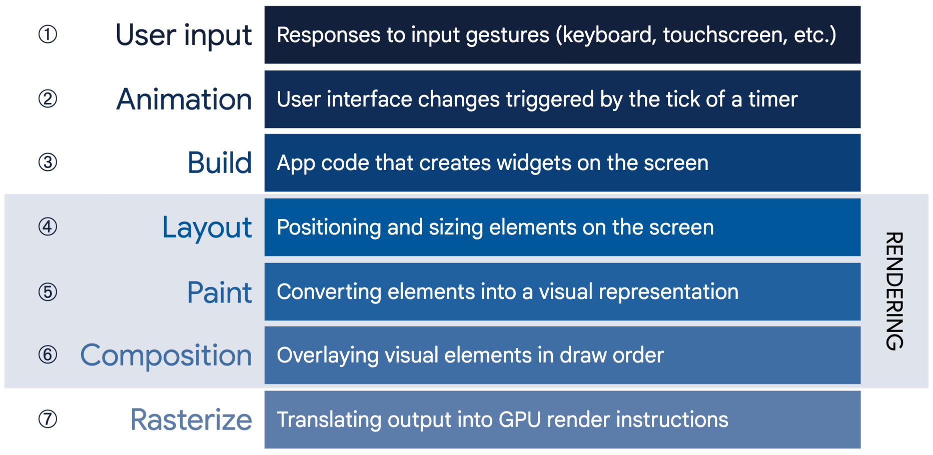

① User input
Responses to input gestures (keyboard, touchscreen, etc.)
② Animation
User interface changes triggered by the tick of a timer
③ Build
App code that creates widgets on the screen
④ Layout
Positioning and sizing elements on the screen
⑤ Paint
Converting elements into a visual representation
⑥ Composition
Overlaying visual elements in draw order
⑦ Rasterize
Translating output into GPU render instructions
RENDERING

Figure 3.8: Flutter Render pipeline [17]

Let’s dive into more detail on some of these steps.

Build: from Widget to Element

Consider, for example, a blue `Container` widget with a `Row` child containing an `Image` and a `Text`. When Flutter needs to render this fragment, it calls the `build()` method, which returns a subtree of widgets and generates the user interface. However, the `build()` method can introduce new widgets during this process if necessary. For instance, if a `Container` has a `color` property, it inserts a `ColoredBox` representing the color into the widget hierarchy, which is the case in this example. Therefore, the final widget hierarchy is often deeper than what the initial code represents.

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

1. `ComponentElement`: a host for other elements
2. `RenderObjectElement`: an element that participates in the layout or paint phases, intermediate between their widget analog and the underlying `RenderObject`

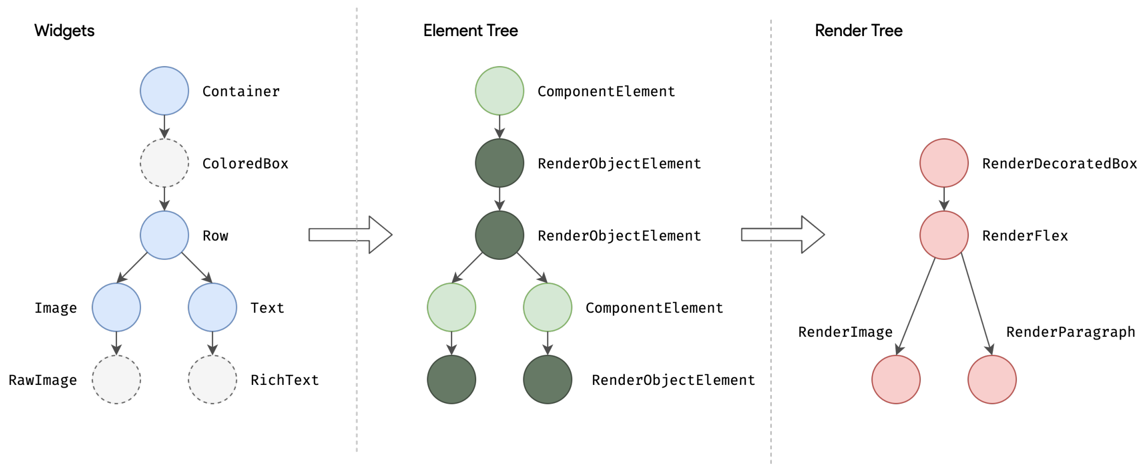


Figure 3.9: Flutter Rendering trees [17]

The element associated with any widget can be referenced using its `BuildContext`, which acts as a handle to the widget's location in the tree. This context is what you find in function calls like `Theme.of(context)`, and it's provided as a parameter to the `build()` method.

The element tree persists from frame to frame, playing a critical role in performance. Flutter can treat the widget hierarchy as disposable while caching its underlying representation. By only walking through the widgets that have changed, Flutter can rebuild only the parts of the element tree that require reconfiguration.

Every node in the render tree is of the base class `RenderObject`, which defines an abstract model for layout and painting. Each `RenderObject` knows its parent but knows little about its children, except how to visit them and what constraints they have. This model is highly general to provide `RenderObject` with sufficient abstraction to handle a variety of use cases. During the build phase, Flutter creates or updates an object that inherits from `RenderObject` for each `RenderObjectElement` in the element tree. This enables efficient layout and painting of widgets in the Flutter framework.

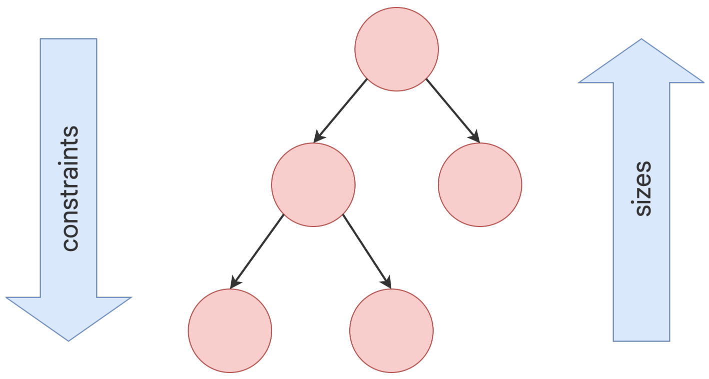


Figure 3.10: Flutter - Constraints and size propagation in the widget tree [17]

Now, to perform the layout, Flutter traverses the render tree in a depth-first manner and passes size constraints from parent to child. The child must adhere to the size constraints imposed by its parent while determining its size. Children respond by returning a size to their parent object within the constraints established by the parent.

A single traversal of the tree is sufficient for every object to have a defined size within its parent's constraints and be ready to be painted by calling the `paint()` method. The box constraint model is very powerful for efficiently laying out objects in linear time $O(n)$.

3.2.6 Hot reload & Hot restart

One of Flutter's key features is *Hot Reload*. It allows developers to instantly see the changes made to their source code in the running application without needing to restart it [26]. *Hot Reload* is extremely fast, speeding up the development process. Indeed, it works by compiling just the altered code and promptly inserting it into the active Dart Virtual Machine `(VM)`. It also preserves the current state of the application, unlike a complete restart. *Hot Reload* can be used with other debugging tools like the Flutter debugger to quickly spot and fix errors.

However, it's important to note that *Hot Reload* might not be suitable for all situations, especially for significant changes in the application's architecture. In such cases, a full *Hot Restart* might be necessary to properly reflect the modifications. *Hot Restart* completely restarts the running Flutter application, resetting it to its initial state.

3.2.7 Navigation

Flutter also provides a complete system for screen navigation and app traversal. For smaller applications without complex links, it's preferable to use the `Navigator` [27], while more complex apps can also make use of the `Router` to manage navigation correctly. Proper handling of deep links on iOS and Android, as well as staying in sync with the address bar for a web application, is crucial.

```
onPressed: () {
  Navigator.of(context).push(
    MaterialPageRoute(
      builder: (context) => const SongScreen(song: song),
    ),
  );
},
child: Text(song.name),
```

Figure 3.11: Flutter - Example using `Navigator` [27]

The `Navigator` widget manages and displays screens in a stack-like fashion. To navigate to a new screen, the Navigator obtained via the `BuildContext` of the current route can use imperative methods like `push()` or `pop()`.

```
@override
  Widget build(BuildContext context) {
  return MaterialApp(
    routes: {
      '/': (context) => HomeScreen(),
      '/details': (context) => DetailScreen(),
    },
  );
}
```

Figure 3.12: Flutter - Example using named routes [27]

As mentioned earlier, named routes can be used in a Flutter app, and they need to be defined in the `main.dart` file of the application. The `MaterialApp` widget provides a `routes` parameter for this purpose.

—

This chapter has provided a detailed overview of the technologies used for the entire frontend of the *ATHLETin* project. The choice of technologies, namely the `Dart` language and the `Flutter` framework, was made with the goal of offering a highly performant and modern solution.

With all the necessary tools for the reader's comprehension now provided, the next section will delve into the details of the module implemented in this thesis, beginning with an introduction to the project's architectural overview.

Part III

Module Implementation

Chapter 4

Requirements

Before delving into the details of implementing the solution, it is important to elucidate the requirements that will guide its development. Therefore, the purpose of this chapter is to describe the objectives and requirements of the three new systems implemented within the administration module.

4.1 Role system requirements

The role system aims to provide members of an affiliation the ability to organize their personnel in a fully flexible and dynamic hierarchy. This should be achieved thanks to roles. A role should be composed of a label, a group of members, and a list of permissions. The permission list of a role explicitly outlines what actions the members associated with that role are authorized to perform within the affiliation. There should be five permissions:

1. **Manage studies**: allows members to create, modify, or delete studies
2. **View questionnaire results**: enables members to see a summary of questionnaire results
3. **Manage athletes**: allows members to create, modify, or delete athletes
4. **Manage members**: permits members to create, modify, or delete members
5. **Manage roles**: enables members to create new roles and modify or delete existing roles except the administrator role

A member without the permission to manage athletes should not be able to create an athlete within the affiliation, for instance. This member should not see the athlete creation form on the athletes' page within the administrative platform, and should not be authorized to submit the athlete creation request on the server side.

Regarding the administrator role, it should be automatically generated upon the creation of an affiliation and logically possesses all permissions. It cannot be altered, not even by administrators, except for its member list, which can be expanded or reduced.

4.2 Affiliation system requirements

The affiliation system aims to group members and athletes of an institution. It is crucial to enable members to access only the data of athletes they genuinely have permission to view. In the absence of an affiliation, a member from one team would potentially have access to the data of an athlete belonging to another team, which is not desirable. In other words, affiliation is the essential entity under which the different institutions using *ATHLETin* organize themselves.

Every member should be able to create a new affiliation or join an existing one, provided they have received an invitation link. If a member creates a new affiliation, it should happen as follows: once created, the new affiliation should consist solely of the creator, automatically designated as an administrator. The creator should be assigned the administrator role and can then add other members and athletes, initiating the *ATHLETin* experience. If a member joins an existing affiliation, they should be simply added to that affiliation as a member and should not possess any role by default.

More generally, affiliation should enable the user to access all *ATHLETin* modules such as the calendar module, the medical module, etc., as it encompasses them.

4.3 Group system requirements

The group system allows to group and organize affiliations with a dynamic level of abstraction. A group should enable a member to group their sub-groups and sub-affiliations for clearer organization. It should also allow members of this group to search among all members and athletes within the sub-groups and sub-affiliations included in this group. A relevant comparison would be the following: a group can be seen as a folder, and an affiliation as a file.

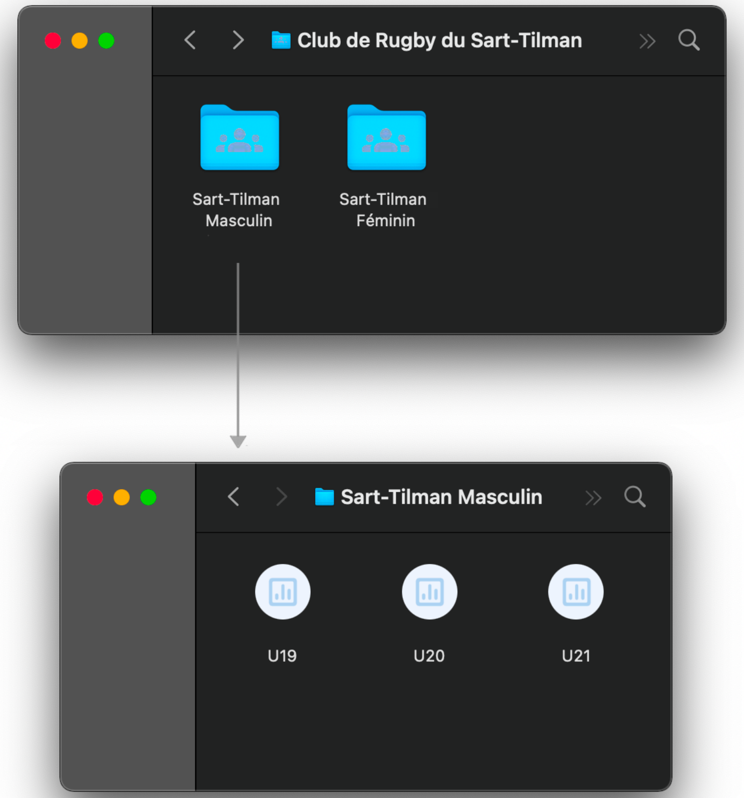


Figure 4.1: Group folder comparison

In the example above, the parent group is the *Club de Rugby du Sart-Tilman* and consists of two sub-groups, namely, the *Sart-Tilman Masculin* and *Sart-Tilman Féminin*. The *Sart-Tilman Masculin* sub-group further comprises three affiliations representing the *U19, U20,* and *U21* teams, respectively.

Every member should be able to create a group or join an existing (sub)group provided they have received an invitation. Similar to affiliations, once the group is created, the creator becomes the group's administrator. Similarly, when a member joins a (sub)group, they are simply added to the group as a member. They are also added to all sub-groups and sub-affiliations of the group they just joined.

The administrator of a group should be able to create sub-groups and sub-affiliations. They can also remove members from the group, unlike a regular member who can only use the search function.

4.4 Platform requirements

As mentioned in the introduction, the three systems — Role, Affiliation, and Group — must be integrated into the administration module conscientiously. They encompass the entirety of the other modules and therefore need to be developed efficiently and compatibly. The frontend, designed by Mr. Mathy and his team, is being developed in Dart and Flutter, so the new code must function in the latest stable versions of Dart and Flutter. Furthermore, the solution must be developed and integrated to be compatible with mobile devices and the null safety mechanism.

On the backend side, the solution needs to be integrated into a database already designed by Mr. Mathy and his team. On one hand, the new requests should be implemented to leverage the maximum potential of the existing content while optimizing wherever possible. On the other hand, the authentication based on the member's role must be implemented on top of everything else to prevent unauthorized requests. Additionally, the solution should be implemented in a way that facilitates future integration and improvements.

—

In this chapter, we've explored the various requirements for the three systems as well as for the platform. Now, let's delve into a description of the overall project architecture. La figure

Chapter 5

High-Level Architecture

Let's now take a look at the high-level architecture of the project. Understanding the different components and their interactions is crucial. Therefore, the purpose of this chapter is to provide an overview of the project's architecture to fully comprehend the solution implemented later on. The figure 5.1 represents the different components of the architecture and the interactions that are established between them.

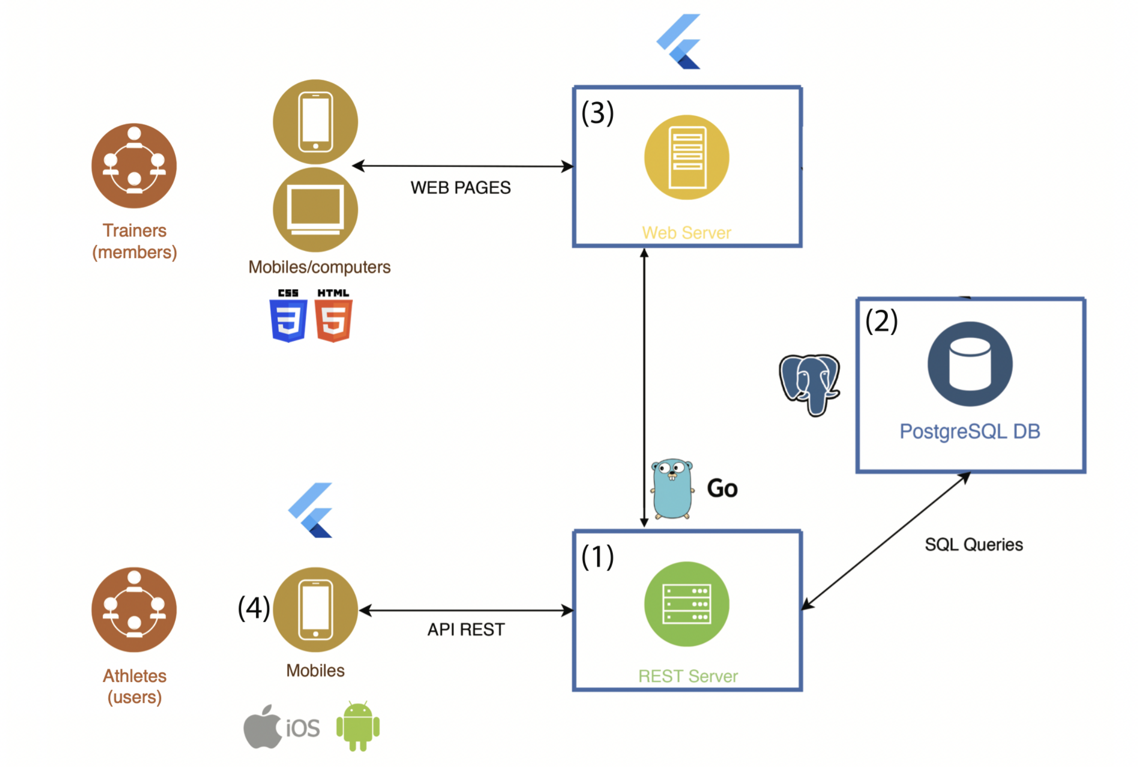


Figure 5.1: Interactions between the different components of the IT architecture

1. The **REST Server** in `Go` is the only entity that can interact directly with the database. It allows other components to retrieve information stored in the database through

a `REST API` [2.4]. To achieve this, it generates `SQL` queries to the database via `GORM` [2.3]. This approach significantly enhances security by centralizing database communications through a single trusted entity.

2. The **PostgreSQL database** stores information for both the web server and the mobile application.

3. The **Web server** is responsible for the web pages accessed by the trainers and can only communicate with the `REST server`. It is achieved through HTTP requests. It uses `JSON` to perform `HTTP` requests to retrieve and send data from/to the `REST server`. The `Web server` is composed of all the web modules described in section 1.2 and is implemented in `Flutter`.

4. The **Mobile application** is used by athletes and can only communicate with the REST server. This component is outside the scope of this thesis.

—

Now, we will delve deeply into the organization of the various components of the architecture and their behavior. This will be further explored in the following chapter.

Chapter 6

The Database

A database aims to efficiently and securely store information while ensuring quick and precise accessibility. For this purpose, a `PostgreSQL` database has been configured within a `Docker` [28] container. This database stores personal information of members and athletes, along with other module-related data such as studies and questionnaires. To maintain an organized and optimized database, specific tables have been designed to perform certain tasks.

In the context of this thesis, organizing the new tables was far from being trivial. Indeed, an efficient database design involves careful consideration of optimal data organization within tables, determining suitable fields to represent data and defining relevant table relationships. As a result, three sets of new tables were designed to meet the requirements of the three new systems — *Role, Affiliation,* and *Group* systems. The goal of this modularity among these different sets of tables is to simplify module comprehension and allow for easy integration. Fourteen new tables were created, bringing the total database tables to 49.

For the sake of development simplicity, the initial set of tables added to the database were those dedicated to the *Role* system. Subsequently, the table sets for the *Affiliation* and *Group* Systems were added on top, as they encompass the entirety of the experience. Tables were thoughtfully designed to minimize their number and interactions during user experience. For tables requiring unique identifiers, the generation of `UUIDs` [29] was utilized.

The figures 6.1, 6.2, and 6.3 respectively represent the sets of tables designed to meet the requirements of the *Role, Affiliation,* and *Group* systems. Additionally, tables 6.1, 6.2, and 6.3 provide more detailed descriptions of the database tables. Primary keys are colored in yellow and Foreign keys in blue. The complete structure of the database can be found in Appendix A.1.

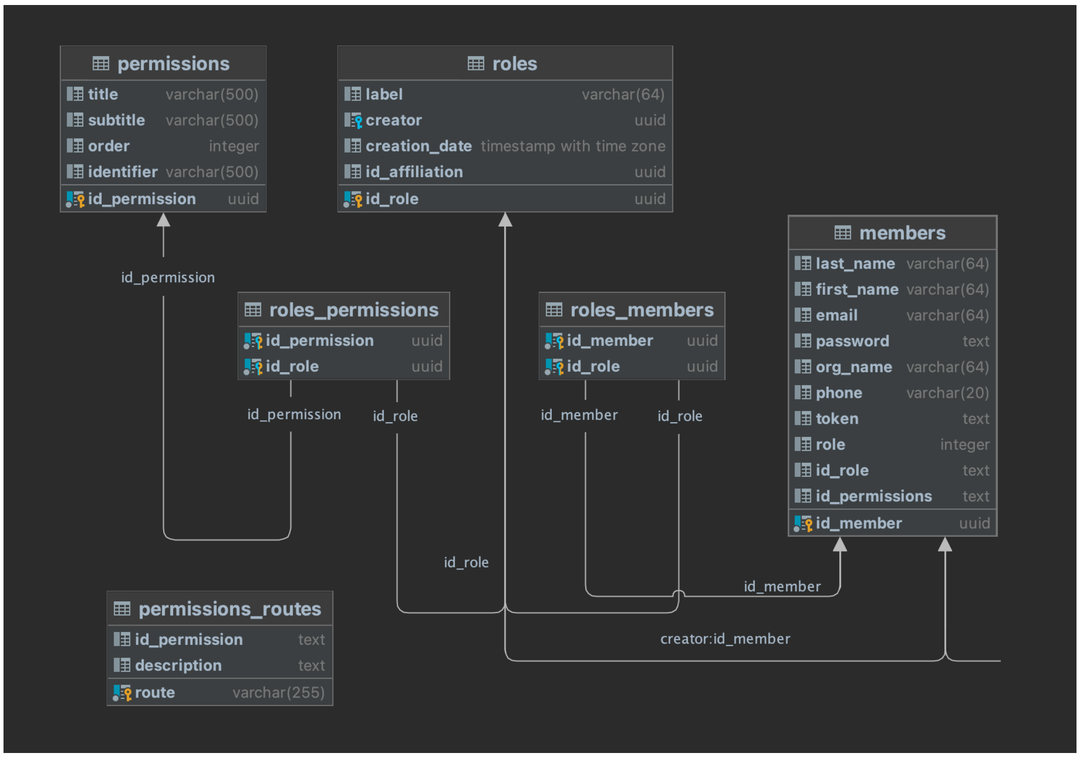


Figure 6.1: Internal structure of the database - *Role* system part

Table Name	Description
roles	Represents a role within an affiliation (e.g., coach or assistant) that provides the ability to assign different permissions to members of an affiliation. The `id_affiliation` field contains the identifier of the affiliation in which the role was created.
roles_members	Contains the mapping between role(s) and member(s). This table allows knowing which member possesses which role.
roles_permissions	Contains the mapping between role(s) and permissions. This table allows understanding which permissions are associated with each role.
permissions	Represents a permission (e.g., view results or manage studies). This table stores a list of predefined permissions and its content is meant to be read-only. Currently, the database holds 5 permissions on the administration website (4.1). The solution has been designed efficiently to allow the addition of future permissions without any issues.

permissions_routes	Contains the mapping between permission(s) and route(s). This table associates permissions with routes and therefore with requests made to the server. Its usage will be detailed in section 7.2.
members	Represents a member (e.g a coach or a healthcare professional) who can access the administration website. The table contains all the important information about a member, such as their name and email address. Additionally, the `id_role` field stores the identifier of the selected role if the member was assigned a role upon creation. The `id_permissions` field communicates the list of permissions that the member has within a given affiliation.

Table 6.1: Database architecture - *Role* system tables

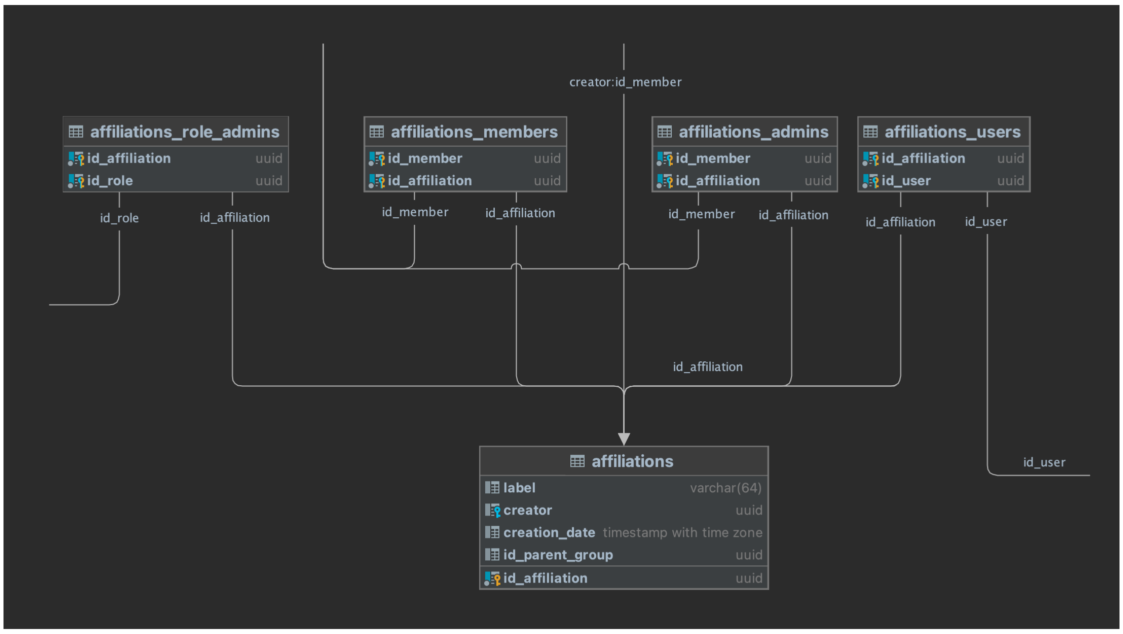


Figure 6.2: Internal structure of the database - *Affiliation* system part

Table Name	Description
affiliations	Represents an affiliation. The `id_parent_group` field (which can be null) aims to store the identifier of the parent group (if any) in which the affiliation was created. In the case where the affiliation does not have a parent group, it is a root affiliation.

affiliations_members	Contains the mapping between member(s) and affiliation(s). This table indicates which member belongs to which affiliation.
affiliations_admins	Contains the mapping between administrator member(s) and affiliation(s). This table shows which member is an administrator of which affiliation.
affiliations_users	Contains the mapping between user(s) and affiliation(s). This table indicates which athlete belongs to which affiliation.
affiliations_role_admins	Contains the mapping between administrator role and affiliation. Indeed, upon the creation of an affiliation, an administrator role associated with that new affiliation is also created.

Table 6.2: Database architecture - *Affiliation* system tables

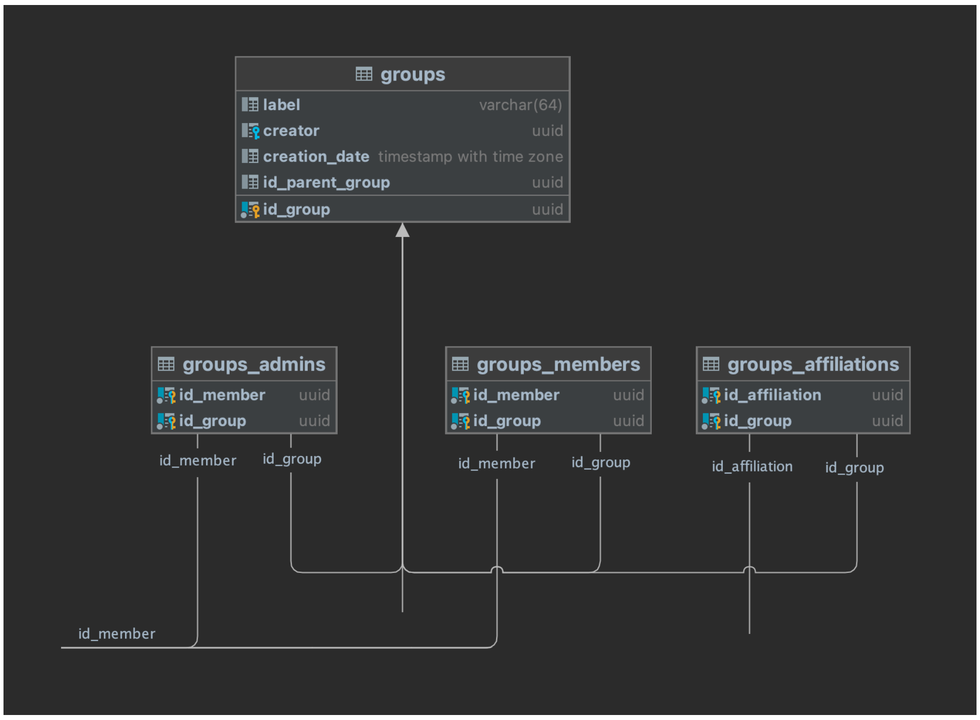


Figure 6.3: Internal structure of the database - *Group* system part

Table Name	Description
groups	Represents a group. A group can be composed of sub-groups as well as sub-affiliations. The `id_parent_group` field designates the potential parent of a subgroup. If the field is null, then the group is not a subgroup and will be referred to as a root group.
groups_members	Contains the mapping between member(s) and group(s). This table indicates which member belongs to which group.
groups_admins	Contains the mapping between administrator member(s) and group(s). This table shows which member is an administrator of which group.
groups_affiliations	Contains the mapping between affiliation(s) and group(s). This table indicates which affiliation originates from which group.

Table 6.3: Database architecture - *Group* system tables

Chapter 7

The REST Server

The REST server is the sole intermediary between users/members and the database. Its purpose is to facilitate access and manipulation of resources on the server through `HTTP` requests. In this section, we'll explore the various details of the REST Server implementation.

7.1 Architecture

7.1.1 Controller-Model pattern

The REST Server has its components organized according to a specific architectural pattern: the *Controller-Model pattern*. It draws significant inspiration from the Model-View-Controller pattern [30]. Let's delve into its specifics.

- The **controller** part aims to perform high-level procedures upon receiving a request. These procedures include tasks like path parameter parsing, `JSON` decoding, or `UUID` checking. Once these procedures are executed, the controller delegates the remaining tasks to the model part. This Controller-Model pattern efficiently coordinates the flow of tasks and interactions.

- The **model** part handles the representation and management of data. That is where the `SQL` tables are represented, defined using `Go` structures with `JSON` annotations. Besides table definitions, the model part contains all the necessary logic to interact with the database tables, with interactions executed through `GORM`.

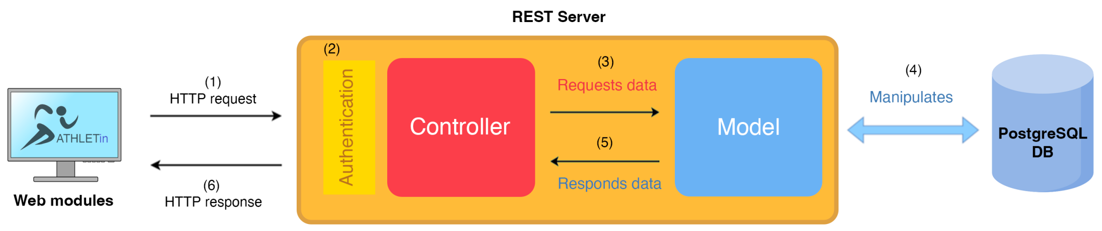


Figure 7.1: REST Server operation

Here's what happens when the server receives a request:

1. The member/user sends an HTTP request to the server via the web modules.
2. The server directs the request through an authentication & role checking phase. It verifies the token's validity and ensures that the member has the authorization to perform this request (via their role and permissions). The authentication & role checking phase is elaborated further in section 7.2.
3. The Controller part decodes the JSON file and parses the different parameters. If there are no errors, the data request is passed to the Model part.
4. The Model part uses the parameters to generate requests to the database and retrieves/manipulates the requested data using GORM.
5. The Model part returns the response to the Controller.
6. The REST server sends the HTTP response to the client.

7.1.2 Files organization

In addition to the Controller-Model architectural pattern, the server files are organized into different folders. This organizational system categorizes services based on similarities between different tasks, ensuring good modularity. The table 7.1 lists the folders composing the server.

Package Name	Description
auth	Contains the authentication components of the REST server. See section 7.2 for further details
controllers	Contains the high-level procedures (e.g. JSON decoding, UUID checking...).
docs	Contains the REST API documentation automatically generated by Swaggo. See section 7.4 for more details.
html	Contains the HTML code sent by email to a newly registered member.

models	Contains the representations of the SQL tables and interacts with the database using GORM.
res	Contains sample data that is used to populate the database in debug mode.
routes	Contains the list of routes exposed by the server.
swaggo	Defines additional structures only used for the API documentation.
tests	Contains unit, integration, and performance tests.
utils	Contains utility functions used in the project (e.g UUID checking).

Table 7.1: Files organization of the REST server

7.2 Authentication & Role Checking

Authentication

The server uses JWT tokens as an authentication mechanism. JWT (JSON Web Token) is an open standard (RFC 7519) utilized for authentication and authorization in web systems and APIs. It secures communication through a digital signature, ensuring the token's authenticity and integrity [31]. Once a member successfully logs in, they receive a dedicated token enabling identification in subsequent requests to access routes, services, and resources. Presently, each request includes a 'bearer authentication' in the header. The token comprises a member/user ID representing the logged-in user and a `sessionID` that corresponds to the current user session. The `sessionID` is used by the mobile application and is beyond the scope of this thesis.

Role Checking

After verifying the token's validity, the server initiates the member's role checking phase based on the received request. The principle is as follows: the server retrieves the member's role(s) within the current affiliation and verifies whether their permissions allow them to execute this request. The aim is to prevent unnecessary workload on the server and the database by directly identifying if the member is authorized to execute this request based on their permissions. To achieve this, several new Go structures have been defined within the server, with the main ones being `role` and `permission`. Let's now see how this works.

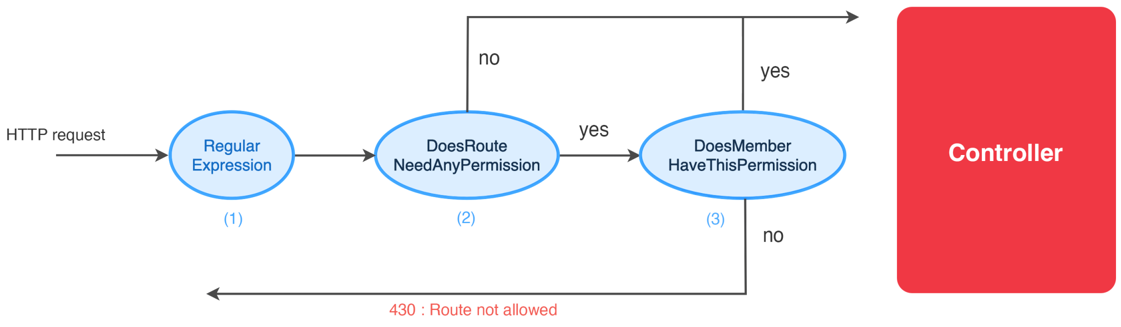
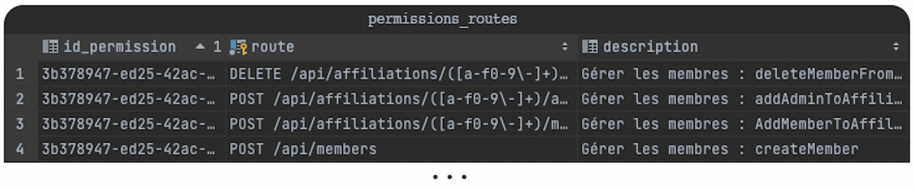


Figure 7.2: REST Server – Role Checking Steps

1. The request is first replaced by its regular expression [32]. This allows for subsequent comparison without worrying about potential UUIDs present in the request.

```
GET /api/groups/1c6444e0-…f476   →   GET /api/groups/([a-f0-9\-]+)
```

2. Checks if the request is a route that requires permission. To achieve this, the request transformed into its regular expression is compared to the list of routes requiring permission stored in the `permissions_routes` table. If there's a match, the route requires specific permission. Otherwise, if there's no match, the request doesn't require any permission and is directly transferred to the Controller part for processing (such as `POST /api/members/login`, for instance).



permissions_routes

	id_permission ▲ 1	route	description
1	3b378947-ed25-42ac-…	DELETE /api/affiliations/([a-f0-9\-]+)…	Gérer les membres : deleteMemberFrom…
2	3b378947-ed25-42ac-…	POST /api/affiliations/([a-f0-9\-]+)/a…	Gérer les membres : addAdminToAffili…
3	3b378947-ed25-42ac-…	POST /api/affiliations/([a-f0-9\-]+)/m…	Gérer les membres : AddMemberToAffil…
4	3b378947-ed25-42ac-…	POST /api/members	Gérer les membres : createMember

...

3. If the route requires permission, the roles and permissions of the member sending the request are retrieved. Roles are retrieved in plural because a member with multiple roles in an affiliation has the combined permissions of those roles. If the member has the permission required for the request among their permissions, it proceeds to the Controller part. Otherwise, an error message is returned.

This solution has been implemented to allow easy and flexible integration of future permissions and routes requiring authorization. Currently, there are 54 routes that require authorization on the server. All these routes can be found in appendix A.2. Each of these routes corresponds to one of the 5 permissions (as seen earlier in section 4.1). Additionally, the solution also flexibly manages different members having distinct permissions across various affiliations without any complications.

7.3 Roles, Affiliations & Groups implementation

Roles implementation within the server

The goal here is to provide a member within an affiliation the ability to create roles with a specific name, selected members, and their chosen permissions. The member should also have the ability to update them. After several days of consideration, I opted for a solution that offered complete flexibility while minimizing the number of interactions and tables.

When a member creates a role with a label, members, and permissions on the web platform, three requests are sent to the server. The first request creates the role in the database directly in the `roles` table, recording its label. The second request registers the members associated with the roles by adding their `ID`s with the created role `ID` in the `roles_members` table. The third request saves the associated permissions by writing their `ID` with the created role `ID` in the `roles_permissions` table. Similarly, to modify a role, each parameter can be updated via a dedicated request.

Affiliations implementation within the server

The affiliation is the central entity of the *ATHLETin* experience, offering access to all application modules. Its implementation must thus be efficient and high-performing. A member should be able to create or join an affiliation, while an administrator (or a member with the necessary authorization) should have the ability to add/delete members and athletes to it.

When a member creates an affiliation, multiple requests are sent to the server. The first involves creating the affiliation in the `affiliations` table. The second is the automatic creation of an administrator role in the `roles` table and registering its connection to this affiliation in the `affiliations_role_admins` table. Subsequently, the creator is added to the affiliation and assigned to the administrator role in the `affiliations_members` (as a member), `affiliations_admins` (as an admin), and `roles_members` tables (for the admin role). The creator can then add/retire members and athletes to the affiliation via the `affiliations_members` and `affiliations_users` tables. They also have the ability to upgrade a member to an admin role, as well as update all affiliation parameters.

When a member joins an existing affiliation, they are simply added to the dedicated `affiliations_members` table of that affiliation, becoming a regular member. For simplicity, the current invitation link for an affiliation is its ID.

In addition to that, there's another noteworthy aspect to mention. Indeed, a member has the ability to create an affiliation (thus becoming its administrator) and later leave it. In such a scenario, an affiliation without an administrator, referred to as a 'ghost' affiliation, would persist in the database unnecessarily consuming space. Hence, when a request is made to remove an administrator or a member, the server checks if the affiliation ends up without any administrators or members. If this condition is met, the affiliation is consequently deleted following that request.

Groups implementation within the server

The group organizes members into subgroups and sub-affiliations for better organization. A member should be able to create a group or join an existing (sub)group. A group administrator can create/delete subgroups and sub-affiliations, as well as remove/upgrade other members to administrators.

When a member creates a group on the web platform, three requests are sent to the server. The first request creates the group in the database by writing it into the `groups` table. The second and third requests register the creating member in the `groups_members` and `groups_admins` tables. If an admin member decides to create a subgroup or sub-affiliation, the process remains the same as above, except that the `id_parent_group` field will contain the ID of the parent group.

When a member joins an existing group, they are simply added to the `groups_members` table of that group, as well as to all subgroups within that group. They are also added to the `affiliations_members` table if there are sub-affiliations within the topology. For simplicity, the current invitation link for a group is its ID.

Another aspect worth mentioning is the deletion of a group. An administrator holds the authority to delete a group, and consequently, the entire hierarchy it encompasses, including sub-groups and sub-affiliations. The implemented solution draws significant inspiration from Preorder Traversal [33]. Visualized as a tree, the group hierarchy is traversed during deletion, descending the tree until reaching a leaf node (a sub-group without sub-groups or a sub-affiliation), which is deleted before moving back up the nodes. It's implemented by recursively exploring sub-groups until reaching one without any child groups (i.e., there are no groups in the `groups` table with its ID as an `id_parent_group`). This method has also been used for establishing the hierarchy while searching a member/athlete within a group tree. The diagram 7.3 illustrates the chronological sequence of the *Standard de Liège* group deletion process.

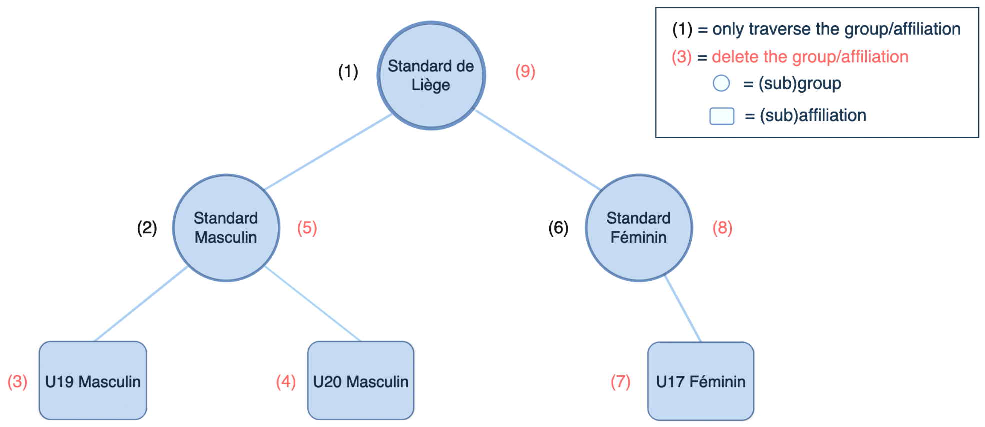


Figure 7.3: Chronological sequence of a group deletion process

7.4 Swagger for documentation

Documentation is a critical part of programming, aiding developers in understanding and correctly utilizing code while reducing potential future errors. The `REST` server employs `Swaggo` [34], a package based on `Swagger` [35], to provide clear and effective documentation. `Swagger` automatically generates `API` documentation by adding comments to the code. It's excellent for maintaining consistent naming conventions, adhering to best practices, and saving coding time on the client side. All the methods exposed in the `API` (methods in the `controller` folder) are described using `Swaggo` syntax, allowing for the visualization of the entire `API` documentation in a web browser. Additionally, it is possible to extract a `JSON` collection from `Swagger` documentation to create a Postman collection, facilitating testing for each `API` entry point.

7.5 Postman as a testing platform

Another tool that significantly accelerated the development phase is Postman [9]. It's straightforward and efficient for creating, testing, and debugging `API`s. It offers a user-friendly interface to construct `HTTP` requests and visualize `API` responses. At the beginning of this project, a dedicated environment and collection were provided via Postman to better understand the `API`. Personally, I extensively used it when testing a newly implemented method on the `REST` server. It saved considerable time by allowing me to verify functionality without launching the frontend each time. Additionally, it facilitated clear transmission of the collection containing newly implemented requests to future developers.

Chapter 8

Administration platform

The administration platform is the web server with which members interact during the *ATHLETin* experience. It serves as the project's facade and, therefore, must be performant, intuitive, and clear. In this chapter, we will explore how the project's frontend is designed, along with its new features.

8.1 Architecture

8.1.1 Controller-Model pattern

Similarly to the `REST` server (7.1.1), the web server is organized following the *Controller-Model* architectural pattern. Here, the **controller** part interacts with the `REST` server to retrieve the requested data and translates the received `JSON` data into formats that can be manipulated by Flutter structures. On the other hand, the **model** part defines these Flutter structures and the requests used to interact with the `REST` server.

8.1.2 Files organization

The administration platform's files are organized into different folders. This facilitates clear categorization and good modularity, which are essential in a project of this scale. Table 8.1 lists the folders that make up the web server.

Package Name	Description
controllers	Contains procedures for managing interactions with the `REST` server and translating received `JSON` data.
models	Contains Flutter representations of the structures to manipulate, as well as the implementation of requests to the REST API.
pages	Contains the implementation of all the pages of the administration platform.

services	Contains the services used to interact with the REST API, along with the necessary singleton [36] implementations. Further details will be provided in the following subsection.
utils	Contains utility functions often used in the project (e.g E-mail validating).
test	Contains the unit tests.

Table 8.1: Files organization of the administration platform

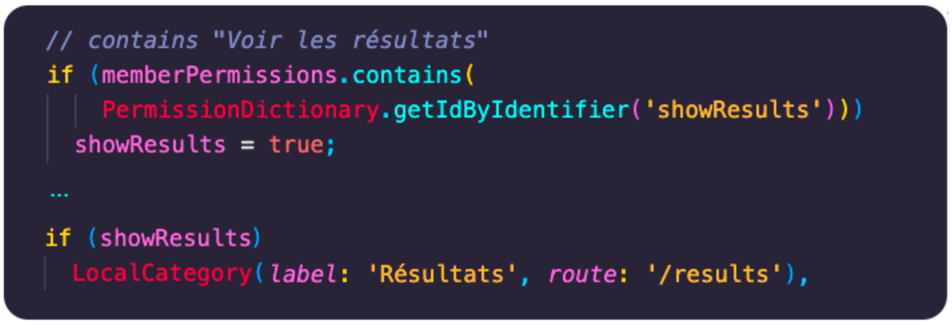
8.1.3 Services

In addition to the already implemented services interacting with the REST API in the `services` folder, I needed to define singletons to manage the currently chosen role, affiliation, and group most effectively. The singleton pattern [36] is used in object-oriented programming to ensure that a class has only one instance and provides a global point of access to it. That's why it was used to represent the following classes.

PermissionsDictionnary

The `PermissionsDictionary` class aims to store and recognize *ATHLETin* permissions regardless of their ID. It represents a dictionary mapping the permission's ID (which can change) to a literal identifier (which does not change). Indeed, a permission's ID might change from one execution to another if desired by the developer, while the literal identifier does not (`'handleStudies'` for example). This way, the web server receiving the set of permissions upon a member's connection can subsequently manage permissions, knowing what to display without relying on hard-coded parts.

As a matter of fact, it's impossible to determine whether the admin platform should display the 'Results' section without a fixed permission identification system (that remains unchanged over time). My initial approach relied on permission IDs, but that would imply that permission IDs cannot change, which isn't the case. Instead, the permission dictionary enables the code to cleanly identify whether a member should see this section/widget or not. Here's how it's implemented.



```
// contains "Voir les résultats"
if (memberPermissions.contains(
    PermissionDictionary.getIdByIdentifier('showResults')))
  showResults = true;

...

if (showResults)
  LocalCategory(label: 'Résultats', route: '/results'),
```

Figure 8.1: `PermissionsDictionnary` utility example

ChosenAffiliation

The `ChosenAffiliation` class simply aims to represent the currently selected affiliation. It provides a global access point to the affiliation (`ID`, `label`, `members`, `admins`, and `role_admin`) throughout the code.

ChosenGroup

In the same way, the `ChosenGroup` class aims to represent the currently selected group. It provides a global access point to the group (`ID`, `label`, `members`, `admins`, and `id_parent_group`). However, as a group can potentially contain subgroups, the logged-in member has the ability to explore as deep as the topology permits. That's why the `ChosenGroup` class is, in reality, a **stack** structure [37] used to represent the current state concerning the group(s). Initially, when the member is on the main selection menu, the stack is empty. Once the member selects a group, that group is added to the stack via a `push` operation. If the member decides to return to the selection menu, the group is removed from the stack using a `pop` operation. Here's an illustrative example of the `ChosenGroup`'s groupstack.

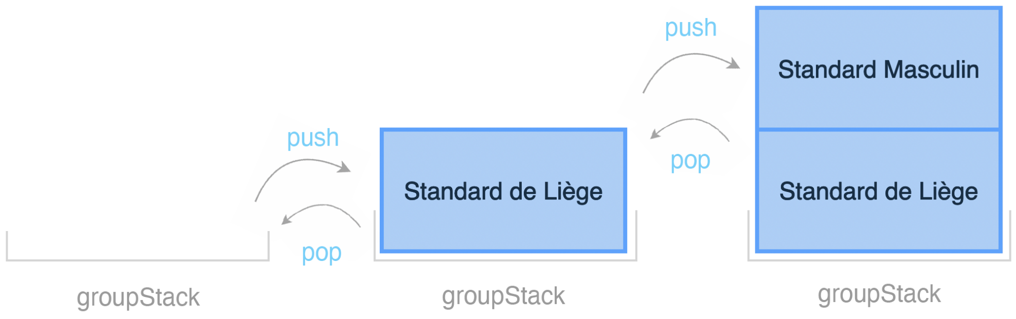
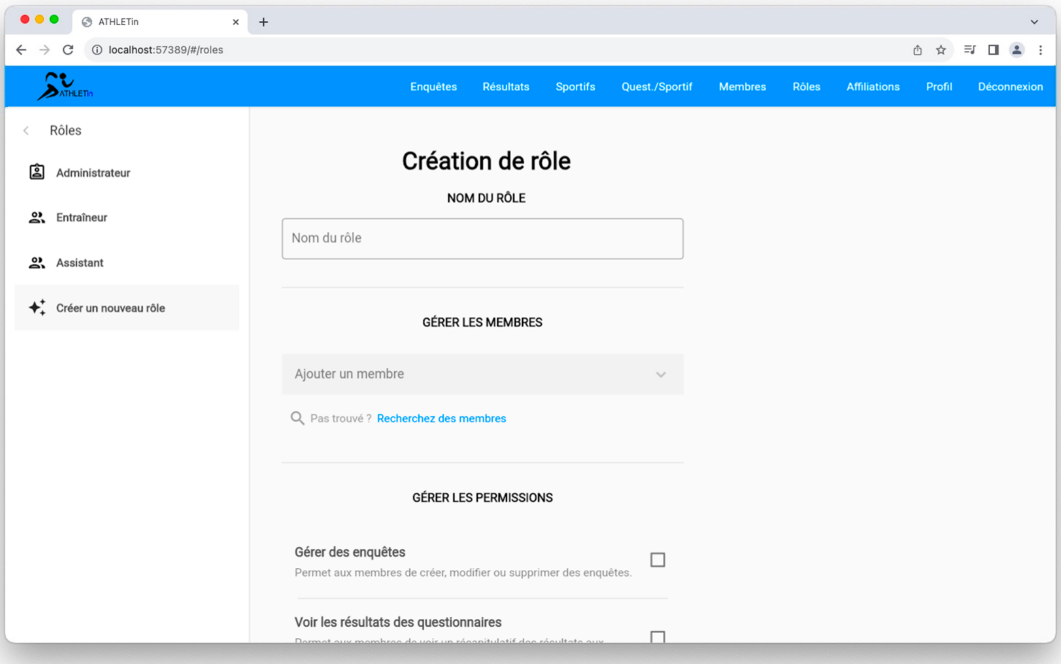


Figure 8.2: `ChosenGroup` stack example

8.2 Role pages

The Role pages serves as the primary interface for implementing the role system. It enables authorized members to create new roles and modify existing ones. A `NavigationDrawer` [38] facilitates access to various sections, initially highlighting the role creation option.

Role creation page

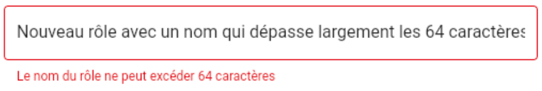


Permet aux membres de créer, modifier ou supprimer des enquêtes.

Figure 8.3: Role creation page

The role creation page allows members to create a role with a label, selected members, and their desired permissions. The only constraint is that the role label should neither be empty nor exceed 64 characters. It's entirely possible to create a role without any members or permissions initially, with the intention of modifying it later.

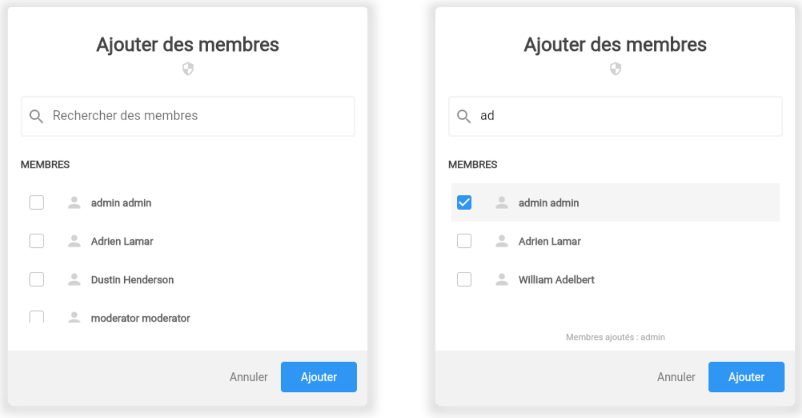
The text field for the label is subject to a validator, ensuring that the label is neither empty nor exceeds 64 characters in length.



Nouveau rôle avec un nom qui dépasse largement les 64 caractères

Le nom du rôle ne peut excéder 64 caractères

To add members, the role creator can either directly choose them from the dropdown list or select them via search. The member search functionality is an `AlertDialog` [39] (also known as a basic dialog) that appears upon clicking 'Search for Members'. This pop-up enables users to search for members based on their names, and the member list automatically filters with each character entered.


Ajouter des membres

Rechercher des membres

MEMBRES

- [ ] admin admin
- [ ] Adrien Lamar
- [ ] Dustin Henderson
- [ ] moderator moderator

Annuler Ajouter

Ajouter des membres

ad

MEMBRES

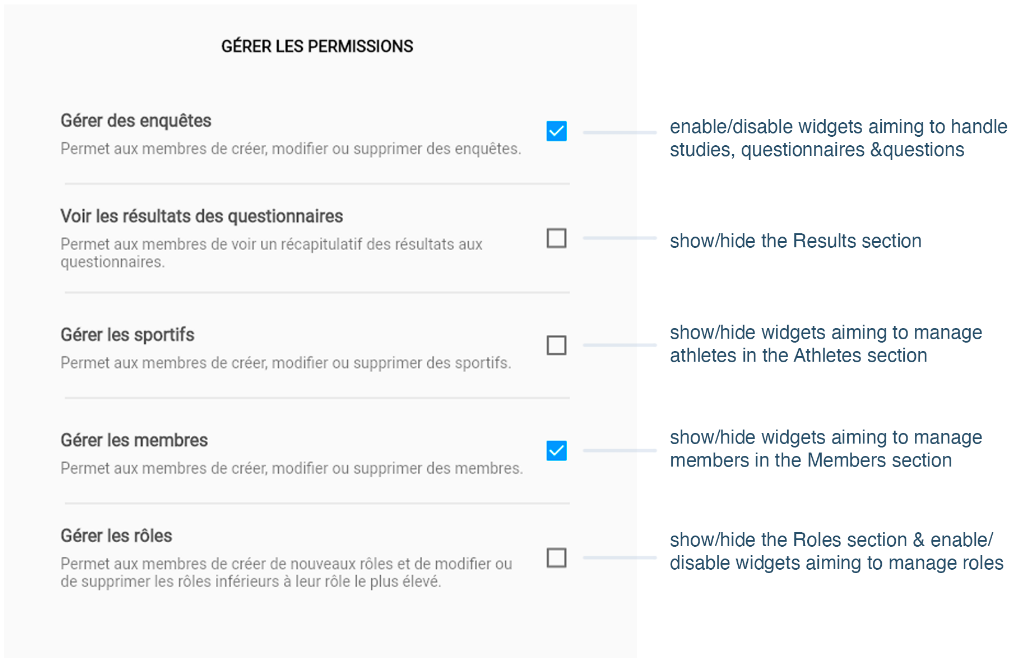
- [✓] admin admin
- [ ] Adrien Lamar
- [ ] William Adelbert

Membres ajoutés : admin

Annuler Ajouter

Figure 8.4: Add member dialog filtering

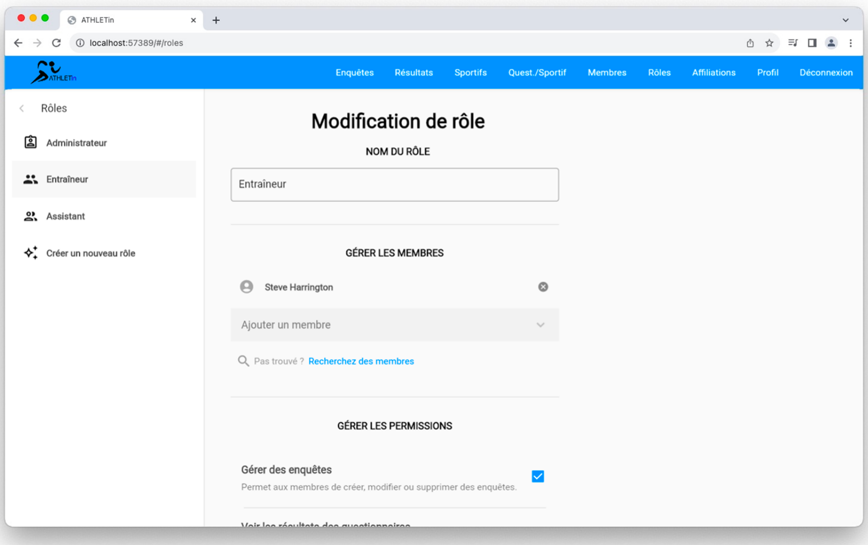
To add permissions to the new role, a `CheckboxListTile` [40] is available. Each selected permission will affect the experience for members assigned to this role. The following figure illustrates the current list of *ATHLETin* permissions and their impact on the user experience.


GÉRER LES PERMISSIONS

Gérer des enquêtes Permet aux membres de créer, modifier ou supprimer des enquêtes.	✓	enable/disable widgets aiming to handle studies, questionnaires &questions
Voir les résultats des questionnaires Permet aux membres de voir un récapitulatif des résultats aux questionnaires.	☐	show/hide the Results section
Gérer les sportifs Permet aux membres de créer, modifier ou supprimer des sportifs.	☐	show/hide widgets aiming to manage athletes in the Athletes section
Gérer les membres Permet aux membres de créer, modifier ou supprimer des membres.	✓	show/hide widgets aiming to manage members in the Members section
Gérer les rôles Permet aux membres de créer de nouveaux rôles et de modifier ou de supprimer les rôles inférieurs à leur rôle le plus élevé.	☐	show/hide the Roles section & enable/ disable widgets aiming to manage roles

Figure 8.5: Permissions list & their impact on the user experience

Role modification page



Modification de rôle

NOM DU RÔLE

Entraîneur

GÉRER LES MEMBRES

Steve Harrington

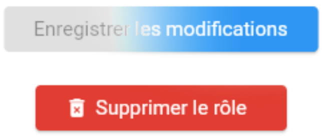
GÉRER LES PERMISSIONS

Gérer des enquêtes ✓
Permet aux membres de créer, modifier ou supprimer des enquêtes.

Voir les résultats des questionnaires

Figure 8.6: Role modification page

The role modification page fundamentally follows the same structure as the creation page. Here, the label, selected members, and chosen permissions are initially pre-filled with corresponding data. The initial state of the role (label, members, and permissions) is stored to detect any changes in the role and enable the 'Save Changes' button. The aim is to offer the ability to send an update request to the server only when a modification has been made. It's also possible to delete the role.



Administrator role page

The admin role modification page allows only viewing its members and adding new members if the logged-in member is an admin of the affiliation. It's not possible to remove members, modify the role name, alter its permissions (all), or delete the role.

Implementation details

In order to minimize interactions with the `REST` server, caching was implemented. Requests for necessary data for the page are made only once, prior to any other actions.

Consequently, the lists of roles, members, and permissions are retrieved from the start. This methodology enhances the page's responsiveness, albeit at the cost of a slightly longer initial loading time, which we consider an acceptable trade-off.

8.3 Affiliation pages

The Affiliation pages essentially consist of a **main menu page** for the *ATHLETin* application, a page providing an **overview of the affiliation**, and a page to **add an affiliation** from the selection menu (create or join). The main menu of an affiliation displays all the application modules as sections, with the affiliation overview being one of them. The affiliation overview page enables members to review affiliation information and administrators to manage it. Specifically, an administrator can upgrade a member to an administrator, remove a member or an athlete from the affiliation, modify the affiliation's name, or delete it.

Affiliation main menu page

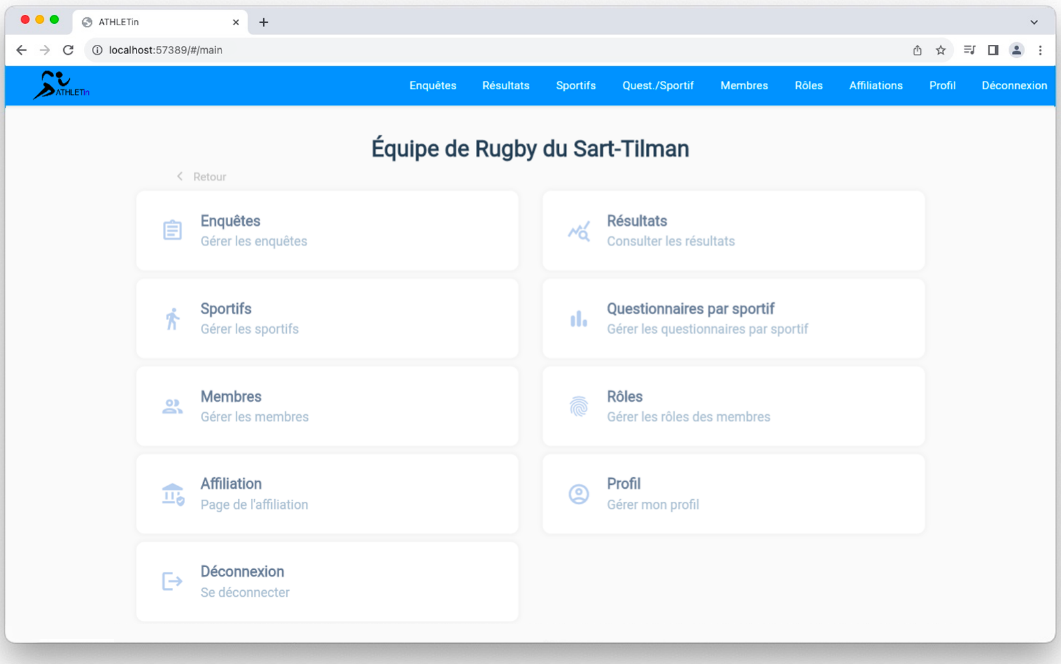
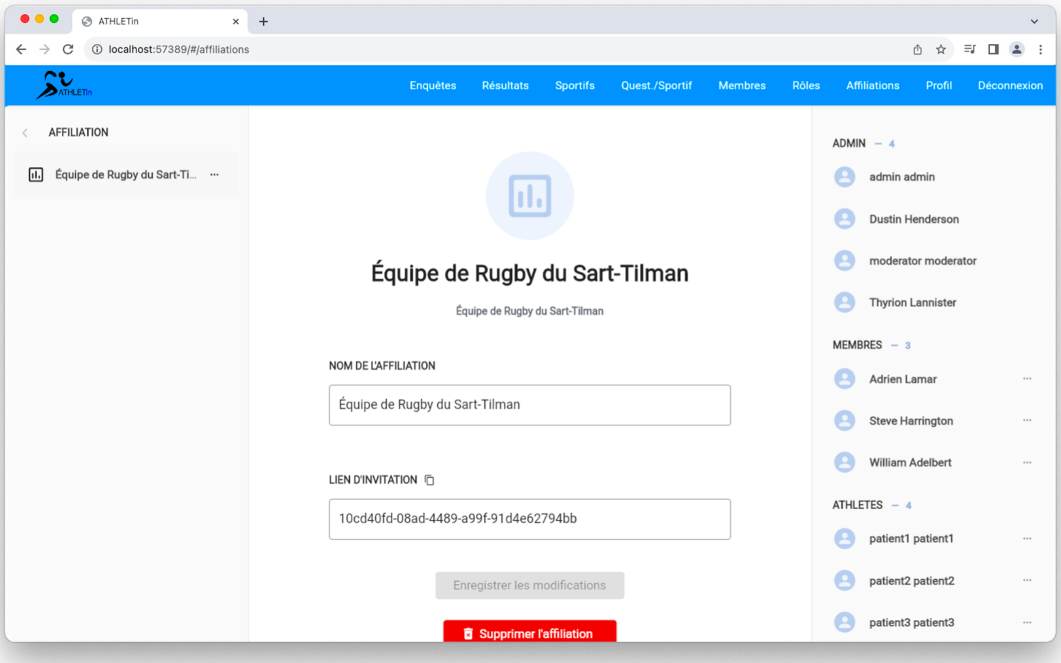


Figure 8.7: Affiliation main menu page

The main menu page of an affiliation is simply a classic home menu displaying the current affiliation's name and all *ATHLETin* modules as sections. All these sections are also accessible via the application bar. Screenshots of the sections that are not covered here can be found in the appendix B.

Affiliation overview page



Équipe de Rugby du Sart-Tilman

Équipe de Rugby du Sart-Tilman

NOM DE L'AFFILIATION

Équipe de Rugby du Sart-Tilman

LIEN D'INVITATION

10cd40fd-08ad-4489-a99f-91d4e62794bb

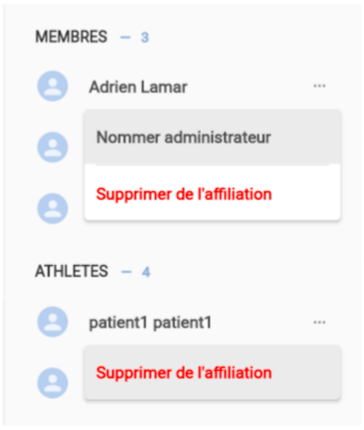
ADMIN — 4
admin admin
Dustin Henderson
moderator moderator
Thyrion Lannister

MEMBRES — 3
Adrien Lamar
Steve Harrington
William Adelbert

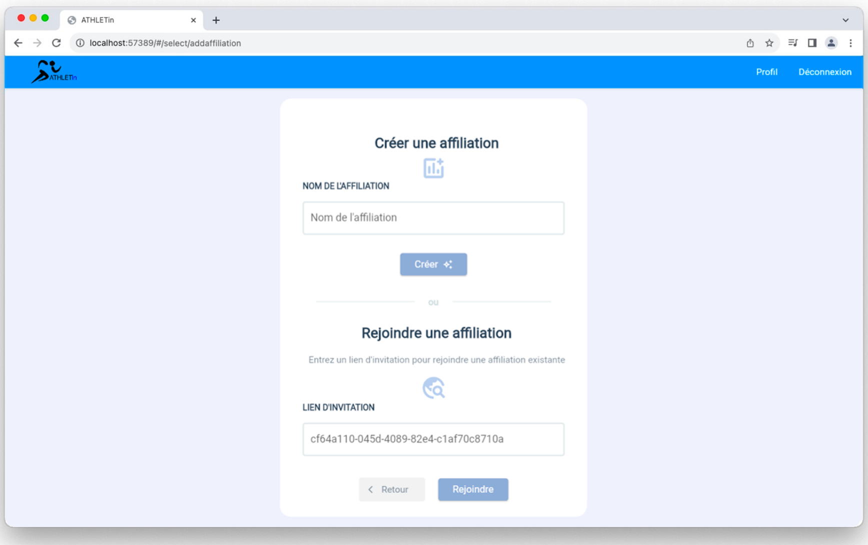
ATHLETES — 4
patient1 patient1
patient2 patient2
patient3 patient3

Figure 8.8: Affiliation overview page

The affiliation overview page allows the viewing member to perform various actions. The member can exit the affiliation using the 'more' button in the top left corner. If the member is an administrator, they can modify the affiliation's name or delete it. They can also upgrade a regular member to an administrator or remove them from the affiliation. Similarly, they can also remove an athlete from the affiliation.



Add an affiliation page

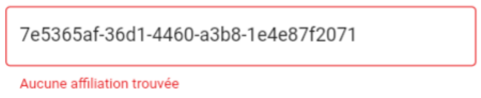


cf64a110-045d-4089-82e4-c1af70c8710a

Figure 8.9: Add affiliation page

The Add affiliation page allows members to create a new affiliation or join an existing one. Members access this page when they select the 'Add an affiliation' option from the affiliation/group selection page (the starting page). If a member creates a new affiliation, the request is processed when they press the creation button. Similar to roles, the affiliation name cannot be empty or exceed 64 characters, ensured beforehand by a validator.

If a member decides to join an existing affiliation by submitting an invitation link (the ID) of the affiliation, a request to join the affiliation is sent to the server. The `REST` server receives the request and successfully attempts to process it. However, if no affiliation with that ID is found, it returns an error message to the web server, which is then displayed on the screen via a validator.



7e5365af-36d1-4460-a3b8-1e4e87f2071

Aucune affiliation trouvée

If the invitation link is valid, the member becomes a member without default permissions and is directed to the main menu page of the affiliation. Therefore, they won't see the 'Results' and 'Roles' sections on the main menu and in the application bar. They also won't have the option to modify/delete the affiliation on the overview as they are not an administrator. Both pages from the perspective of a member without permissions are illustrated below.

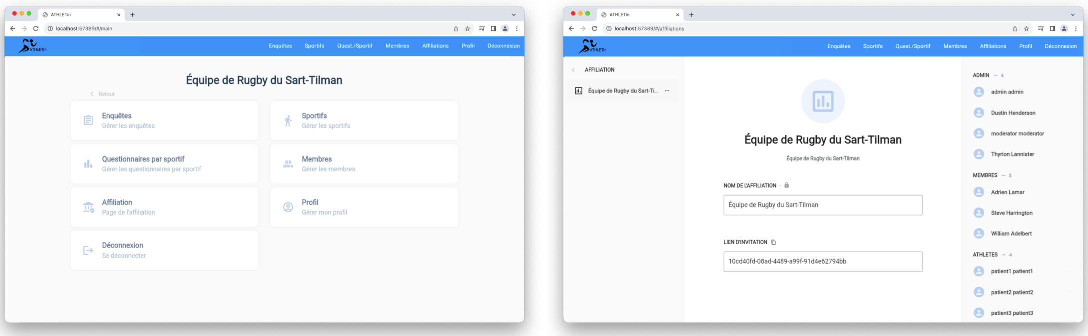


Figure 8.10: Affiliation main menu & overview page as a member with no permission

Implementation details

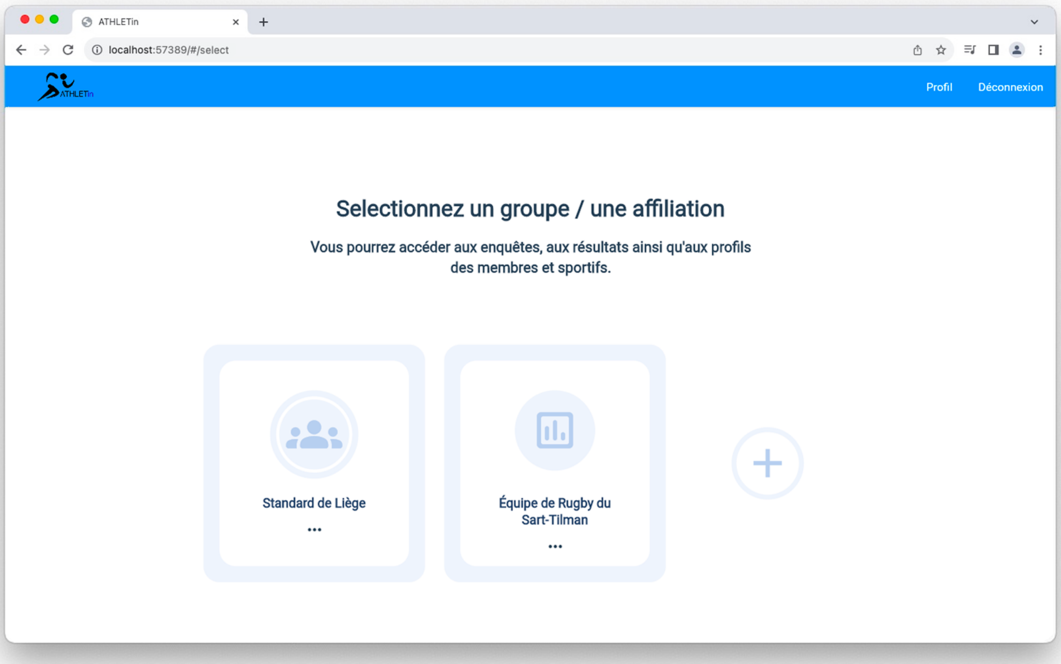
In order to minimize interactions with the REST server, caching was implemented. Requests for necessary data for the pages are made only once, prior to any other actions. Consequently, the lists of administrators, members, and athletes are retrieved from the start.

The use of `ChangeNotifierProvider` and `Consumer` has also been extensively employed to minimize and target widget rebuilds when necessary. When a request modifying the affiliation has been successfully executed, the method `notifyListeners()` enables the reconstruction of only those parts dependent on this change. Hence, if there's a modification in the affiliation's name, the consumers of the `AffiliationChangesNotifier` are notified (which includes all widgets displaying the affiliation's label).
`MembersChangesNotifier` and `UsersChangesNotifier` are used similarly.

8.4 Group pages

The group pages allow members to organize and navigate through a completely dynamic and flexible topology. They consist of a **selection page**, a **group menu page**, a **group overview page**, and two pages summarizing the activity of a sought **member or athlete** within the topology.

Selection page



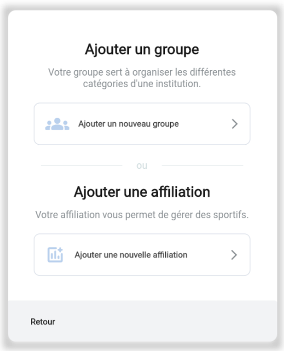
Selectionnez un groupe / une affiliation

Vous pourrez accéder aux enquêtes, aux résultats ainsi qu'aux profils des membres et sportifs.

Figure 8.11: Selection page

The selection page is the first page a member lands on just after logging in. It presents the member with their ’root’ groups and affiliations, as well as a button to add a group or an affiliation. A ’root’ group simply means it is the highest group in the topology to which the member belongs. It is possible to select a group to explore its topology or an affiliation to directly access the main menu of that affiliation. Additionally, one can leave the group/affiliation via the ’more’ button below the entity’s name.

The group/affiliation addition button brings up a dialog that allows the member to choose whether to add a group or an affiliation. Both the group and affiliation addition pages are essentially the same, offering the option to create or join using an invitation link.

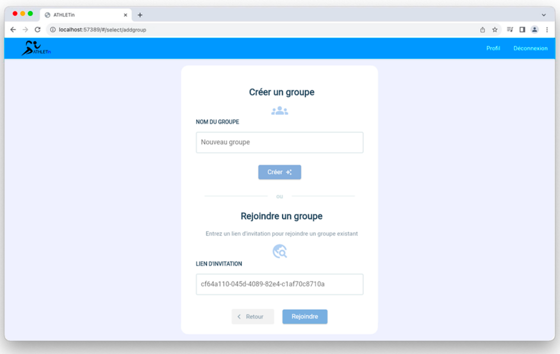


Ajouter un groupe

Votre groupe sert à organiser les différentes catégories d'une institution.

Ajouter une affiliation

Votre affiliation vous permet de gérer des sportifs.



Créer un groupe

NOM DU GROUPE: Nouveau groupe

Rejoindre un groupe

Entrez un lien d'invitation pour rejoindre un groupe existant

LIEN D'INVITATION: cf64a110-045d-4089-82e4-c1af70c8710a

Group menu page

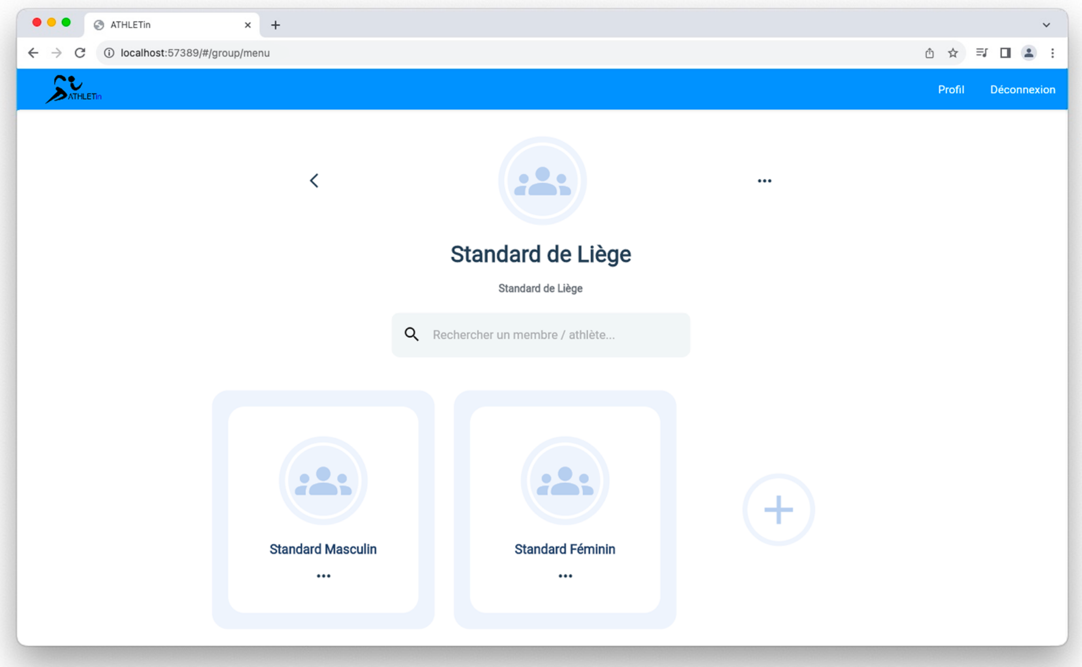


Figure 8.12: Group menu page

The group menu page is where a member lands when choosing a group from the selection page or a subgroup from the group menu page. It enables the member to see the subgroups and sub-affiliations within the selected group. It also provides access to the overview of the currently selected subgroup via the 'more' button at the top right. The page also features a search bar, further explained below. If the member is an administrator, it's also possible to add a subgroup/sub-affiliation via the '+' button. An important point to note here is that the administrator member can only create subgroups/affiliations and cannot join them via an invitation link inside a group.

The search bar allows the logged-in member to search for other members or athletes throughout the entire topology (starting from the currently selected group). It provides a filterable list based on whether one is searching for a member or an athlete. It also automatically filters with each keyboard input. Once a member or athlete is selected, the logged-in member lands on the dedicated member/athlete page.

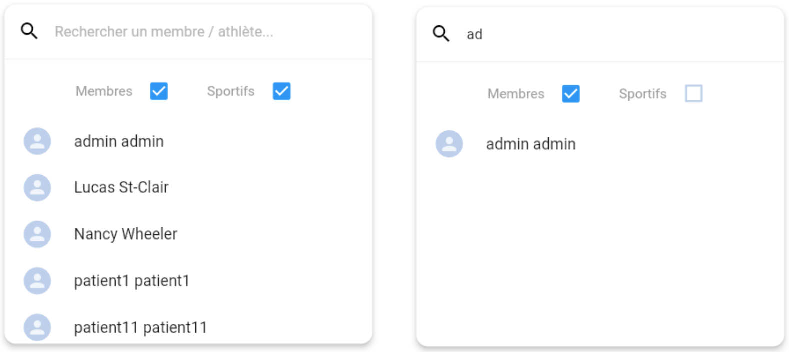


Figure 8.13: Group search bar

Group overview page

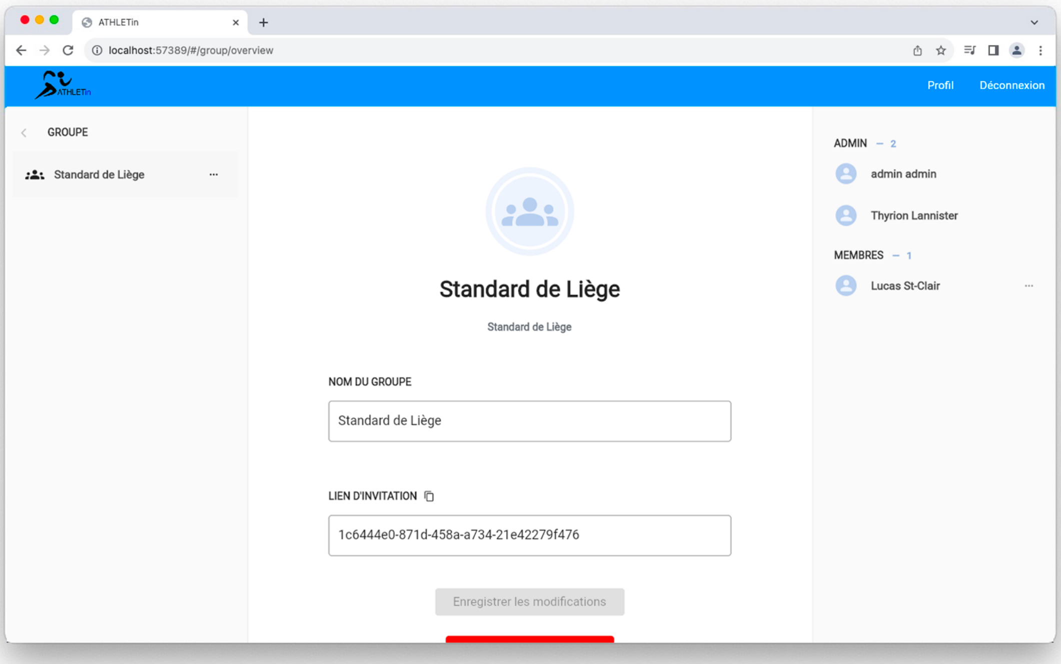
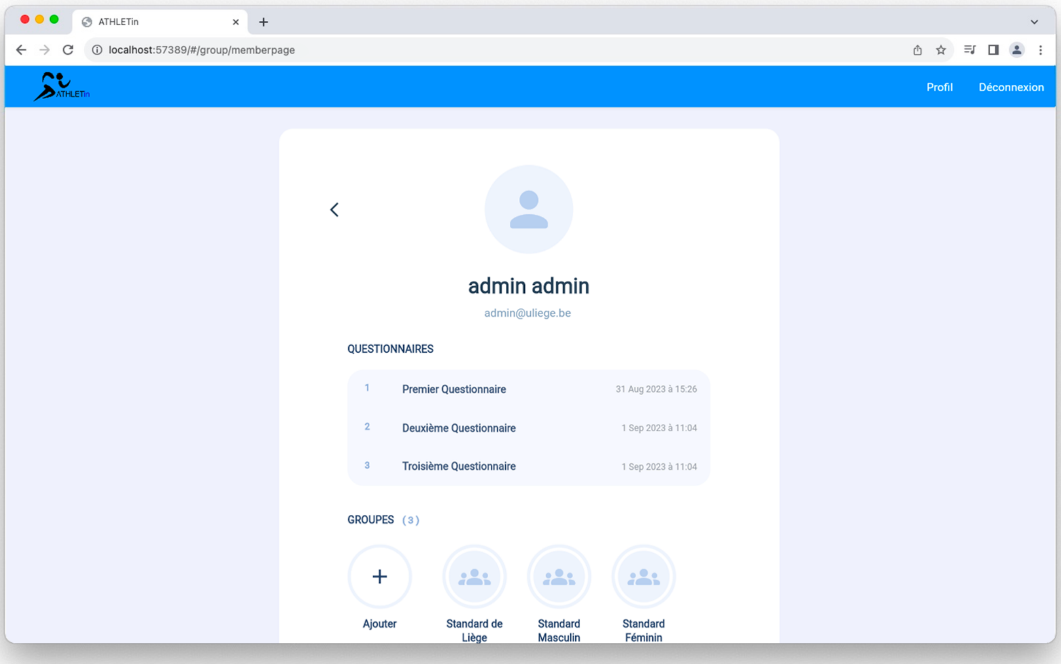


Figure 8.14: Group overview page

Fundamentally similar to the affiliation overview page, the group overview page allows members to view the group's information. If the member is an administrator, they can edit the name, remove or upgrade a member to an administrator. They can also delete the group.

Member page



admin admin

admin@uliege.be

QUESTIONNAIRES

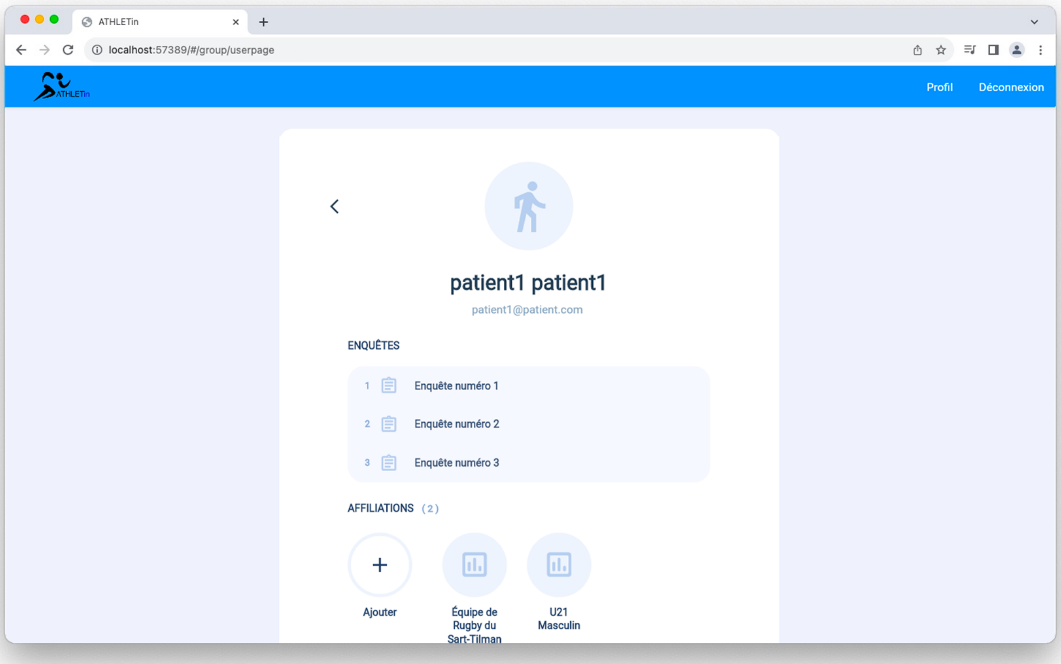
1	Premier Questionnaire	31 Aug 2023 à 15:26
2	Deuxième Questionnaire	1 Sep 2023 à 11:04
3	Troisième Questionnaire	1 Sep 2023 à 11:04

GROUPES (3)

Figure 8.15: Member page

The member page allows the connected member to view various pieces of information about a member within the topology. They can see the list of questionnaires created by that member along with their creation dates. Additionally, they can view the groups and affiliations to which this member belongs. If the connected member is a group administrator, there is a '+' button available that allows them to add groups/affiliations to this member via their ID. If the connected member is not an administrator, they can only view those information.

Athlete page


ATHLETin
localhost:57389/#/group/userpage
Profil
Déconnexion
Ajouter
Équipe de Rugby du Sart-Tilman
U21 Masculin

patient1 patient1

patient1@patient.com

ENQUÊTES

1 Enquête numéro 1

2 Enquête numéro 2

3 Enquête numéro 3

AFFILIATIONS (2)

Figure 8.16: User page

The athlete page is essentially the same as the member page. There is the list of studies in which the athlete is registered. There is no list of groups since an athlete can only belong to affiliations and not groups.

Implementation details

In order to minimize interactions with the `REST` server, caching was implemented. Requests for necessary data for the pages are made only once, prior to any other actions. Consequently, the lists of subgroups, sub-affiliations, members, and athletes are retrieved from the start.

The use of `ChangeNotifierProvider` and `Consumer` has also been extensively employed to minimize and target widget rebuilds when necessary. Hence, if there's a modification in the group's name, the consumers of the `GroupChangesNotifier` are notified (which includes all widgets displaying the group's label).
`SearchChangesNotifier`, `MembersChangesNotifier` and `UsersChangesNotifier` are used similarly.

Chapter 9

Testing & Performance tools

In this chapter, we will explore the testing tools used to verify the correctness of implemented functionalities. Testing is a crucial phase in software development as it ensures the quality and reliability of applications. This chapter covers the different types of tests applied to the project.

9.1 Unit Tests

Unit tests are designed to independently test small units of code to ensure that each element performs its intended function. These small units can be functions, classes, or isolated methods. This type of testing is well-suited for testing REST API methods. Regarding the web server, it is also well-suited for testing business logic, algorithms, and data processing operations. Because they operate on small portions of code, they are straightforward to implement and run efficiently.

9.2 Widget Tests

Widget tests in Flutter are designed to test the behavior and appearance of individual widgets within the application. They focus on testing user interface components, allowing developers to simulate user interactions. This involves testing widgets, their interactions, and the overall responsiveness of the interface. Widget tests ensure that interface components render correctly and maintain the expected state throughout the application's lifecycle. In summary, they ensure a smooth and reliable user experience in Flutter applications.

9.3 Integration Tests

Integration tests assess how different parts of the application interact and come together. They're the most complete type of automated tests, aiming to verify smooth collaboration among various widgets, modules, or features of the application. Specifically, an integration test checks communication between components, data consistency, and responsiveness to application interactions. In practice, they validate data flow, navigation, API integration, and the overall application behavior. Integration tests contribute to ensuring the robustness and overall performance of the Flutter application. However, due to their complexity, they are slow to execute.

9.4 Manual Tests

During this thesis, a complete set of automated tests unfortunately was not implemented. Instead, emphasis was placed on manual testing for quick and targeted verifications at the expense of comprehensive case coverage. This approach allowed successful testing of all actions achievable on the web application. Additional tests from a more malicious perspective were also conducted to verify that a logged-in member only has access to resources granted by their permissions. Access to unauthorized pages and unauthorized requests were tested and covered successfully. As the application includes many recurring code structures (e.g., member lists or group widget implementations), some manual tests could be performed once to validate multiple parts of the module.

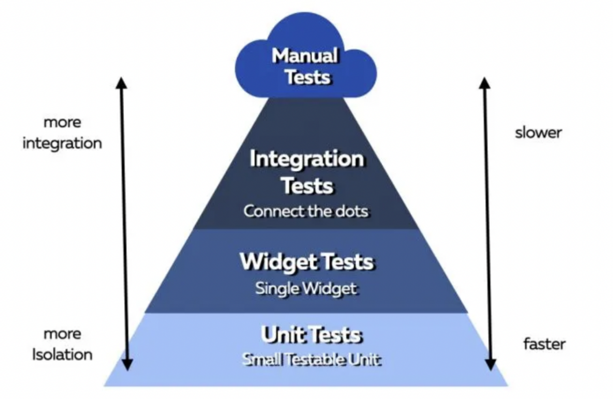


Figure 9.1: Test types relationships [41]

9.5 Performance Tools

Regarding the backend part, query performance is enhanced through indexes in tables. They expedite the search and retrieval of records. Specifically, indexes optimize searches by rapidly locating records that meet specific search criteria, reducing the time required to execute `SELECT` queries. Indexes also ensure the uniqueness of values in a column or set of columns and are often used for primary keys or unique constraints. They additionally speed up joins between multiple tables and data filtering. Indexes have been defined for all new implemented tables, which can be found in Appendix A.1.

Regarding the frontend part, `Dart` and Flutter provide tools allowing to get better insights on the way the app is run. The `Dart DevTools` [42] is a list of features that allows for profiling and debugging `Dart` and Flutter applications. Within the tools accessible for web application, the following are included:

- The *Flutter Inspector* helps visualize and explore Flutter widget trees, aiding in understanding existing layouts and diagnosing layout issues [43].
- The *Flutter Debugger* allows pausing the application at breakpoints, executing the app step by step, and inspecting variables [44].

Other features were implemented by Flutter, such as the CPU and Network profiler, but they were not available for web applications in profile mode.

Browser tools

For deeper insights into the app's performance, the browser can be utilized to directly scrutinize the module's behavior. `Chrome DevTools` [45] were employed while examining the app for CPU profiling, network profiling, and frame rate analysis. For an overall assessment of the website, Lighthouse [46] was employed. This open-source tool, initiated by Google, aids developers in optimizing their websites. It provides a rating ranging from 1 to 100 across various aspects of the website, including:

1. Performance: Measures factors influencing loading speed, such as interactivity and time to load. It was not able to score the module because the performance should be handled internally by Flutter during compilation to JavaScript code.
2. Accessibility: Checks for elements that affect how easily users with disabilities can navigate and understand the content.
3. Best Practices: Assesses adherence to web development best practices, including secure browsing, use of modern technology, and avoiding deprecated practices.
4. SEO (Search Engine Optimization): Evaluates how well the website can be discovered by search engines, assessing metadata, keywords, and other factors.

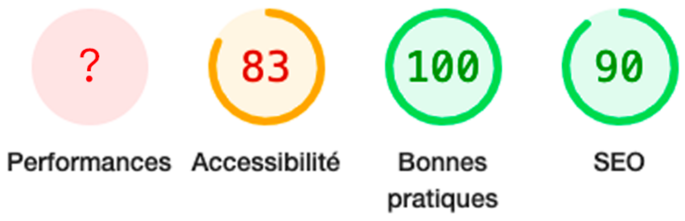


Figure 9.2: Lighthouse score for the `Select Page`

Chapter 10

Deployment

After completing the app development, the next phase involves deployment. This chapter will outline *ATHLETin*'s module structure and detail the method employed to consolidate all modules, aiming to streamline the user experience.

10.1 Docker to encapsulate the modules

The various components of the *ATHLETin* project, such as the medical and calendar modules, were enclosed within containers to manage their respective dependencies. Each module is encapsulated separately, allowing for independent dependency management. `Docker` [28] was utilized to create these containers. The module separation achieved enables running them on distinct dependency versions aligned with their development requirements. Ultimately, `Docker`'s usage in encapsulating these modules facilitates faster and simpler deployment procedures.

`Docker` serves as a widely-used containerization tool, enabling developers to package applications along with their dependencies into self-contained containers. In the context of this app, `Docker` simplifies containerization. Once the dependencies are accurately listed in the `Dockerfile`, building the `Docker` image becomes straightforward. This resulting image can be executed on any environment equipped with `Docker`.
The image generation process involved two main steps:

1. Setting up the environment: Starting with a `Debian` [47] image, it configures and installs the essential dependencies required for running Flutter.

2. Generating the runtime image: The compiled web application is then transferred into an `Nginx` container (further details below).

10.2 Nginx as a reverse proxy

ATHLETin consists of multiple standalone modules, each needing access within a unified web server. `Nginx` [48] was employed to create this server, configured as a reverse proxy to distribute users across the different modules depending on the `URI` they send. It routes users to the appropriate module based on their request. These modules are differentiated by their unique port numbers while sharing the same domain.

Part IV

Conclusion

Chapter 11

Conclusion

Throughout this thesis, we've detailed our implementation of a `REST` server, a database, and an administrative web module for *ATHLETin*. The administrative module comprises a role system for flexible permission granting, an affiliation system to group and manage members and athletes, and a group system to organize sports organizations into teams. The aim of this module is to provide coaches and health specialists with a flexible, user-friendly platform for managing their athletes. This thesis builds upon a project initiated by Professor Laurent Mathy and his team, motivating the decision to develop modular components to ease future integration.

Before delving into project implementation, it was crucial to analyze the current system specifications, particularly the technologies employed and architectural decisions. The backend of the existing system consisted of a `REST` server in `Go` coupled with a `PostgreSQL` database. This server communicated with the database using `GORM`, an `ORM` specifically tailored for `Go`. Regarding the frontend, `Dart` language and the Flutter framework were chosen. Flutter, being a cross-platform `SDK`, enables 'write once, deploy everywhere' functionality. This property might prove beneficial in the future if Professor Mathy intends to repurpose parts of the web module for other application projects.

The development of the solution was divided into 3 phases. Firstly, the role system was designed and implemented. Secondly, the affiliation system was implemented, consequently completing the role system. Thirdly, the grouping system was implemented on top of the other two. This approach allowed the creation of modular components that align with an already existing and advanced project.

This thesis focused on achieving high performance while ensuring user-friendly interactions, which posed a considerable challenge. It was necessary to devise the most optimal way to implement the three systems on top of an already ongoing project. To enhance the efficiency of the `REST` server, we optimized the database tables by minimizing the number of tables to only the essential ones and incorporating indexes. These indexes play a vital role in maintaining reasonable response times, especially as the database scales. We also optimized the `REST` server by implementing requests to minimize the database workload. In addressing web performance issues, unnecessary page rebuilding has been identified as a significant challenge. To tackle this, we adopted the provider state management pattern. This pattern facilitates selective component rebuilding based on user interactions, preventing unnecessary rebuilding of unchanged parts. Moreover, the web platform also minimizes interactions with the `REST` server through its caching mechanisms.

Throughout the development of the administration module, various types of tests were created to verify the entirety of the implemented functionalities. Once the development phase concluded, the subsequent steps involved deployment. Generalities about the project were introduced, along with specific details regarding its case.

In conclusion, the objective of this thesis was to create a solution for *ATHLETin*'s issues. After multiple design and implementation phases, the result meets the clients' requirements successfully. The enhanced administrative module comprising role, affiliation, and group systems empowers coaches and healthcare professionals to flexibly and effectively manage a sports organization according to their preferences. Regarding the backend and frontend implementation, the components were designed in a modular manner to facilitate integration within Professor Mathy's architecture. Finally, the solution includes documentation to ease future reuse.

Chapter 12

Future work

To conclude this thesis, this final chapter below lists possible new features that could be added to enhance the current system capabilities and user experience.

12.1 Responses to questionnaires in athlete page

Alongside personal information, studies and affiliations, it would be relevant to showcase an athlete's questionnaire responses when a member views their researched athlete page (see 8.16). This could provide an accessible summary of a athlete's responses within a single view.

12.2 More languages

The modules currently offer a French interface. Incorporating additional languages could not only broaden the app's audience but also assist non-French-speaking athletes using the app when answering questions. Expanding the range of supported languages could make the *ATHLETin* solution accessible to a larger user base.

12.3 Instant messaging

In Section 1.2.5, we highlighted the need for a communication feature enabling members to exchange messages in an instant messaging format similar to Messenger or WhatsApp. Although this functionality hasn't been implemented at present, as a recommendation for future project handling, it could be realized using web sockets.

12.4 Question creation

While studies and questionnaires can be generated, the feature for creating questions has not been implemented. Although a simple menu could suffice, considering that questions will occupy a significant portion of the database, efficiency in their creation becomes crucial. Efficiency, in this context, aims to prevent generating numerous similar questions that might result in redundant duplicates in the database. Utilizing existing features like keywords could help by initially searching if a question already exists before its creation.

12.5 Event visualization

Each questionnaire is active based on its event's time period. Implementing a visualization feature would aid management by providing a clear view of the number of active questionnaires during specific time periods. This can prevent the setup of an excessive number of questionnaires simultaneously, preventing athletes from being overloaded with an excessive amount of questions to answer.

12.6 Improve responsiveness

The application currently has limited responsiveness features since its original design targeted computer usage exclusively. While certain Flutter widgets adjust their display when screen size changes, enhancements are needed to enhance the platform's usability across various devices.

Appendix A

Backend Appendix

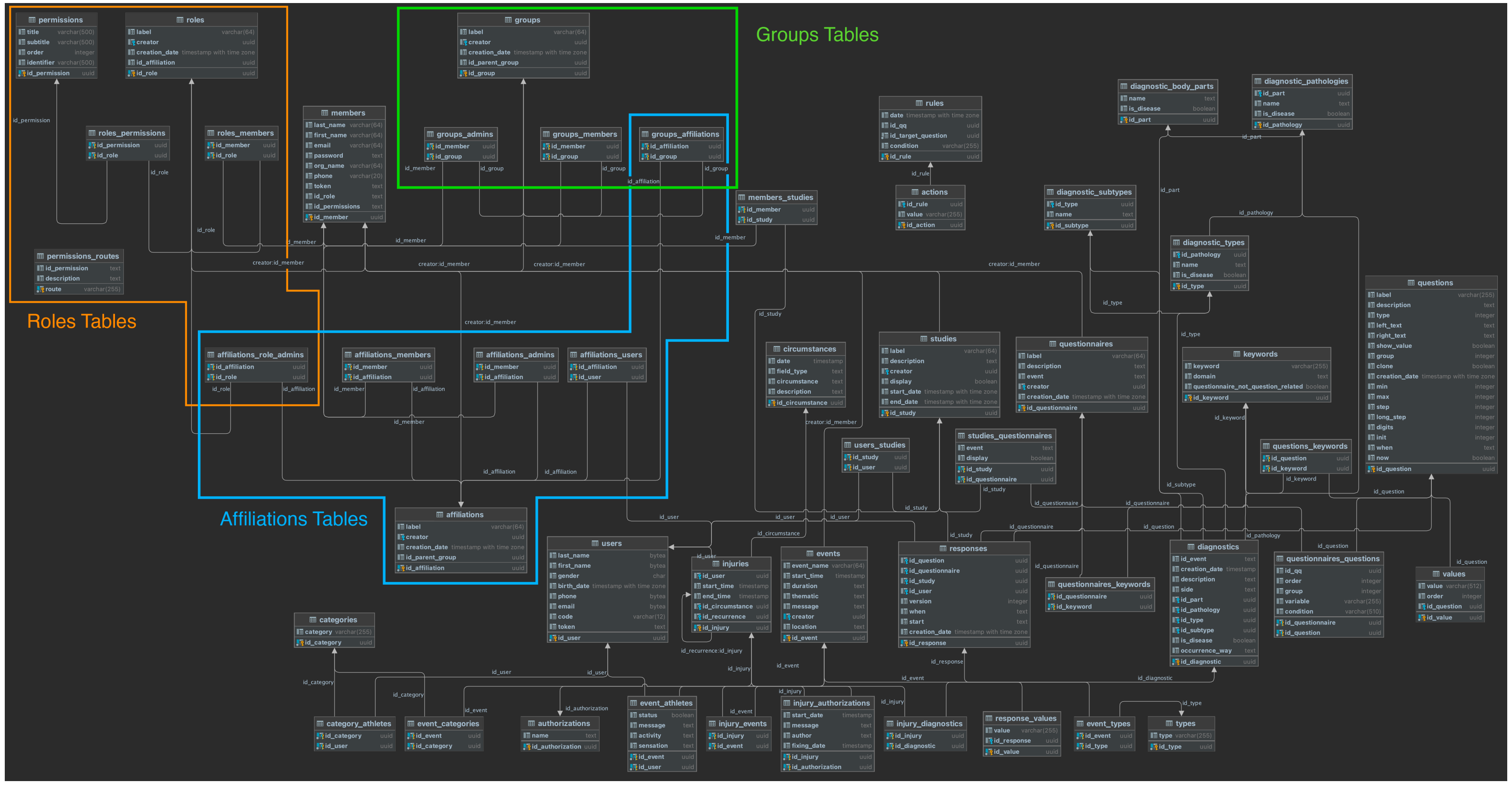
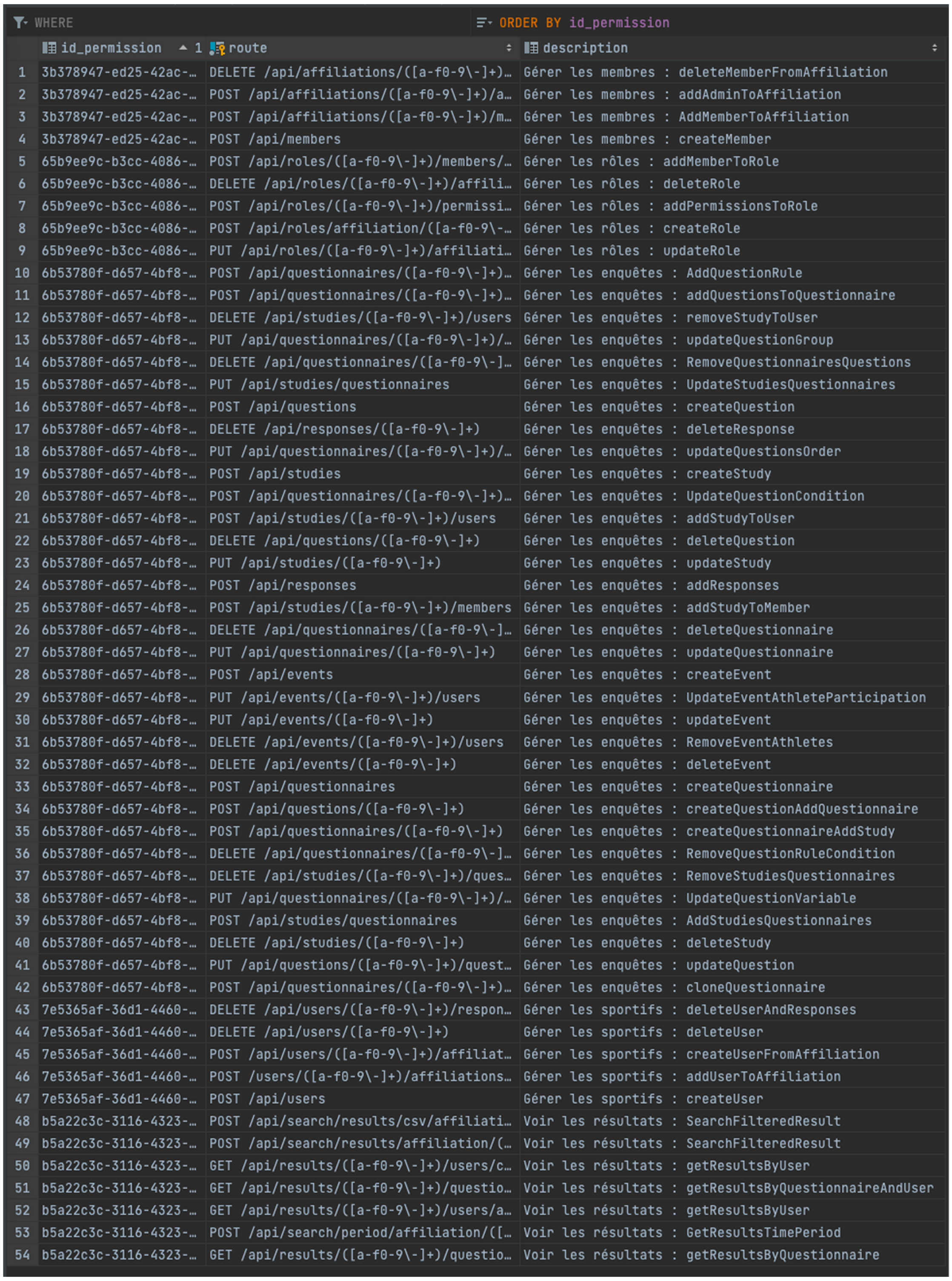


Figure A.1: Complete structure of the database



WHERE ORDER BY id_permission

	id_permission	route	description
1	3b378947-ed25-42ac-…	DELETE /api/affiliations/([a-f0-9\-]+)…	Gérer les membres : deleteMemberFromAffiliation
2	3b378947-ed25-42ac-…	POST /api/affiliations/([a-f0-9\-]+)/a…	Gérer les membres : addAdminToAffiliation
3	3b378947-ed25-42ac-…	POST /api/affiliations/([a-f0-9\-]+)/m…	Gérer les membres : AddMemberToAffiliation
4	3b378947-ed25-42ac-…	POST /api/members	Gérer les membres : createMember
5	65b9ee9c-b3cc-4086-…	POST /api/roles/([a-f0-9\-]+)/members/…	Gérer les rôles : addMemberToRole
6	65b9ee9c-b3cc-4086-…	DELETE /api/roles/([a-f0-9\-]+)/affili…	Gérer les rôles : deleteRole
7	65b9ee9c-b3cc-4086-…	POST /api/roles/([a-f0-9\-]+)/permissi…	Gérer les rôles : addPermissionsToRole
8	65b9ee9c-b3cc-4086-…	POST /api/roles/affiliation/([a-f0-9\-…	Gérer les rôles : createRole
9	65b9ee9c-b3cc-4086-…	PUT /api/roles/([a-f0-9\-]+)/affiliati…	Gérer les rôles : updateRole
10	6b53780f-d657-4bf8-…	POST /api/questionnaires/([a-f0-9\-]+)…	Gérer les enquêtes : AddQuestionRule
11	6b53780f-d657-4bf8-…	POST /api/questionnaires/([a-f0-9\-]+)…	Gérer les enquêtes : addQuestionsToQuestionnaire
12	6b53780f-d657-4bf8-…	DELETE /api/studies/([a-f0-9\-]+)/users	Gérer les enquêtes : removeStudyToUser
13	6b53780f-d657-4bf8-…	PUT /api/questionnaires/([a-f0-9\-]+)/…	Gérer les enquêtes : updateQuestionGroup
14	6b53780f-d657-4bf8-…	DELETE /api/questionnaires/([a-f0-9\-]…	Gérer les enquêtes : RemoveQuestionnairesQuestions
15	6b53780f-d657-4bf8-…	PUT /api/studies/questionnaires	Gérer les enquêtes : UpdateStudiesQuestionnaires
16	6b53780f-d657-4bf8-…	POST /api/questions	Gérer les enquêtes : createQuestion
17	6b53780f-d657-4bf8-…	DELETE /api/responses/([a-f0-9\-]+)	Gérer les enquêtes : deleteResponse
18	6b53780f-d657-4bf8-…	PUT /api/questionnaires/([a-f0-9\-]+)/…	Gérer les enquêtes : updateQuestionsOrder
19	6b53780f-d657-4bf8-…	POST /api/studies	Gérer les enquêtes : createStudy
20	6b53780f-d657-4bf8-…	POST /api/questionnaires/([a-f0-9\-]+)…	Gérer les enquêtes : UpdateQuestionCondition
21	6b53780f-d657-4bf8-…	POST /api/studies/([a-f0-9\-]+)/users	Gérer les enquêtes : addStudyToUser
22	6b53780f-d657-4bf8-…	DELETE /api/questions/([a-f0-9\-]+)	Gérer les enquêtes : deleteQuestion
23	6b53780f-d657-4bf8-…	PUT /api/studies/([a-f0-9\-]+)	Gérer les enquêtes : updateStudy
24	6b53780f-d657-4bf8-…	POST /api/responses	Gérer les enquêtes : addResponses
25	6b53780f-d657-4bf8-…	POST /api/studies/([a-f0-9\-]+)/members	Gérer les enquêtes : addStudyToMember
26	6b53780f-d657-4bf8-…	DELETE /api/questionnaires/([a-f0-9\-]…	Gérer les enquêtes : deleteQuestionnaire
27	6b53780f-d657-4bf8-…	PUT /api/questionnaires/([a-f0-9\-]+)	Gérer les enquêtes : updateQuestionnaire
28	6b53780f-d657-4bf8-…	POST /api/events	Gérer les enquêtes : createEvent
29	6b53780f-d657-4bf8-…	PUT /api/events/([a-f0-9\-]+)/users	Gérer les enquêtes : UpdateEventAthleteParticipation
30	6b53780f-d657-4bf8-…	PUT /api/events/([a-f0-9\-]+)	Gérer les enquêtes : updateEvent
31	6b53780f-d657-4bf8-…	DELETE /api/events/([a-f0-9\-]+)/users	Gérer les enquêtes : RemoveEventAthletes
32	6b53780f-d657-4bf8-…	DELETE /api/events/([a-f0-9\-]+)	Gérer les enquêtes : deleteEvent
33	6b53780f-d657-4bf8-…	POST /api/questionnaires	Gérer les enquêtes : createQuestionnaire
34	6b53780f-d657-4bf8-…	POST /api/questions/([a-f0-9\-]+)	Gérer les enquêtes : createQuestionAddQuestionnaire
35	6b53780f-d657-4bf8-…	POST /api/questionnaires/([a-f0-9\-]+)	Gérer les enquêtes : createQuestionnaireAddStudy
36	6b53780f-d657-4bf8-…	DELETE /api/questionnaires/([a-f0-9\-]…	Gérer les enquêtes : RemoveQuestionRuleCondition
37	6b53780f-d657-4bf8-…	DELETE /api/studies/([a-f0-9\-]+)/ques…	Gérer les enquêtes : RemoveStudiesQuestionnaires
38	6b53780f-d657-4bf8-…	PUT /api/questionnaires/([a-f0-9\-]+)/…	Gérer les enquêtes : UpdateQuestionVariable
39	6b53780f-d657-4bf8-…	POST /api/studies/questionnaires	Gérer les enquêtes : AddStudiesQuestionnaires
40	6b53780f-d657-4bf8-…	DELETE /api/studies/([a-f0-9\-]+)	Gérer les enquêtes : deleteStudy
41	6b53780f-d657-4bf8-…	PUT /api/questions/([a-f0-9\-]+)/quest…	Gérer les enquêtes : updateQuestion
42	6b53780f-d657-4bf8-…	POST /api/questionnaires/([a-f0-9\-]+)…	Gérer les enquêtes : cloneQuestionnaire
43	7e5365af-36d1-4460-…	DELETE /api/users/([a-f0-9\-]+)/respon…	Gérer les sportifs : deleteUserAndResponses
44	7e5365af-36d1-4460-…	DELETE /api/users/([a-f0-9\-]+)	Gérer les sportifs : deleteUser
45	7e5365af-36d1-4460-…	POST /api/users/([a-f0-9\-]+)/affiliat…	Gérer les sportifs : createUserFromAffiliation
46	7e5365af-36d1-4460-…	POST /users/([a-f0-9\-]+)/affiliations…	Gérer les sportifs : addUserToAffiliation
47	7e5365af-36d1-4460-…	POST /api/users	Gérer les sportifs : createUser
48	b5a22c3c-3116-4323-…	POST /api/search/results/csv/affiliati…	Voir les résultats : SearchFilteredResult
49	b5a22c3c-3116-4323-…	POST /api/search/results/affiliation/(…	Voir les résultats : SearchFilteredResult
50	b5a22c3c-3116-4323-…	GET /api/results/([a-f0-9\-]+)/users/c…	Voir les résultats : getResultsByUser
51	b5a22c3c-3116-4323-…	GET /api/results/([a-f0-9\-]+)/questio…	Voir les résultats : getResultsByQuestionnaireAndUser
52	b5a22c3c-3116-4323-…	GET /api/results/([a-f0-9\-]+)/users/a…	Voir les résultats : getResultsByUser
53	b5a22c3c-3116-4323-…	POST /api/search/period/affiliation/([…	Voir les résultats : GetResultsTimePeriod
54	b5a22c3c-3116-4323-…	GET /api/results/([a-f0-9\-]+)/questio…	Voir les résultats : getResultsByQuestionnaire

Figure A.2: `permissions_routes` – All 54 routes that requires authorization

Appendix B

Modules screenshots

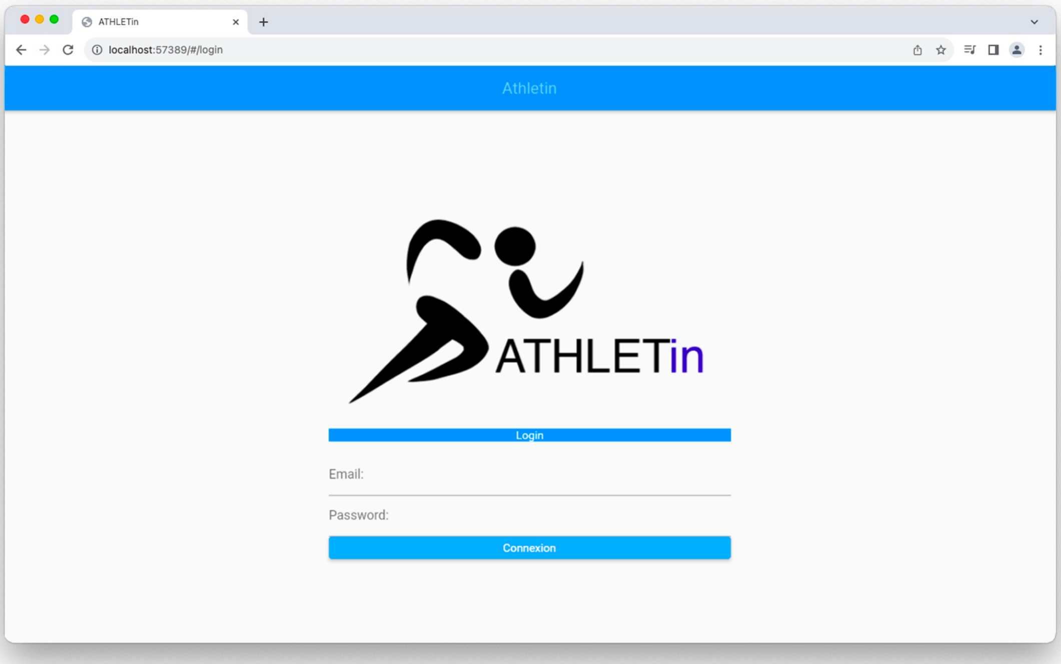
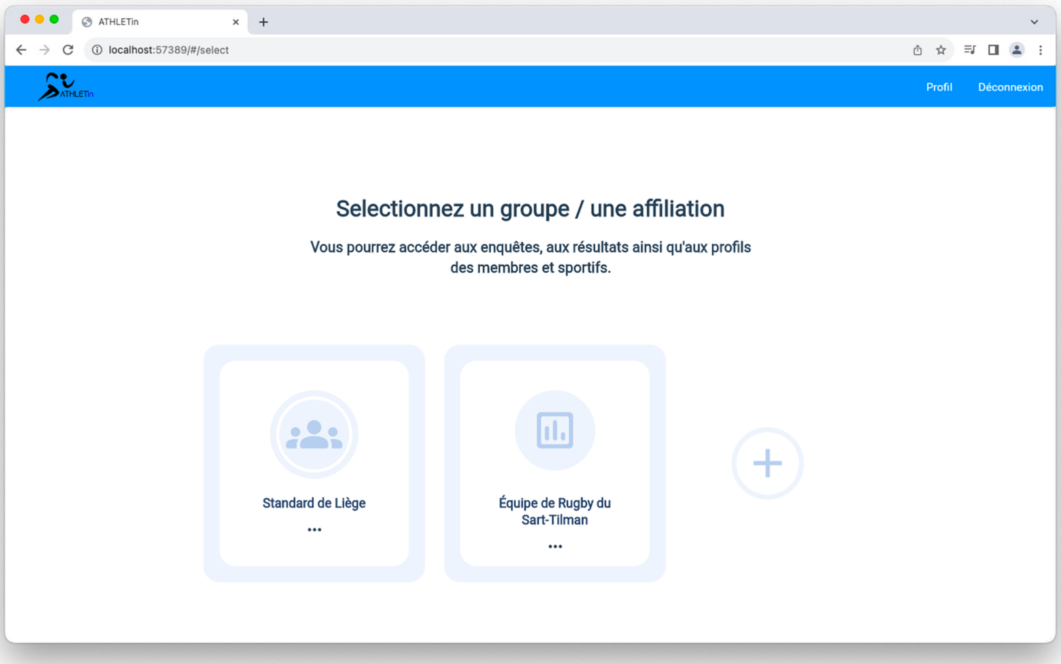


Figure B.1: Login page



Selectionnez un groupe / une affiliation

Vous pourrez accéder aux enquêtes, aux résultats ainsi qu'aux profils des membres et sportifs.

Figure B.2: Selection page

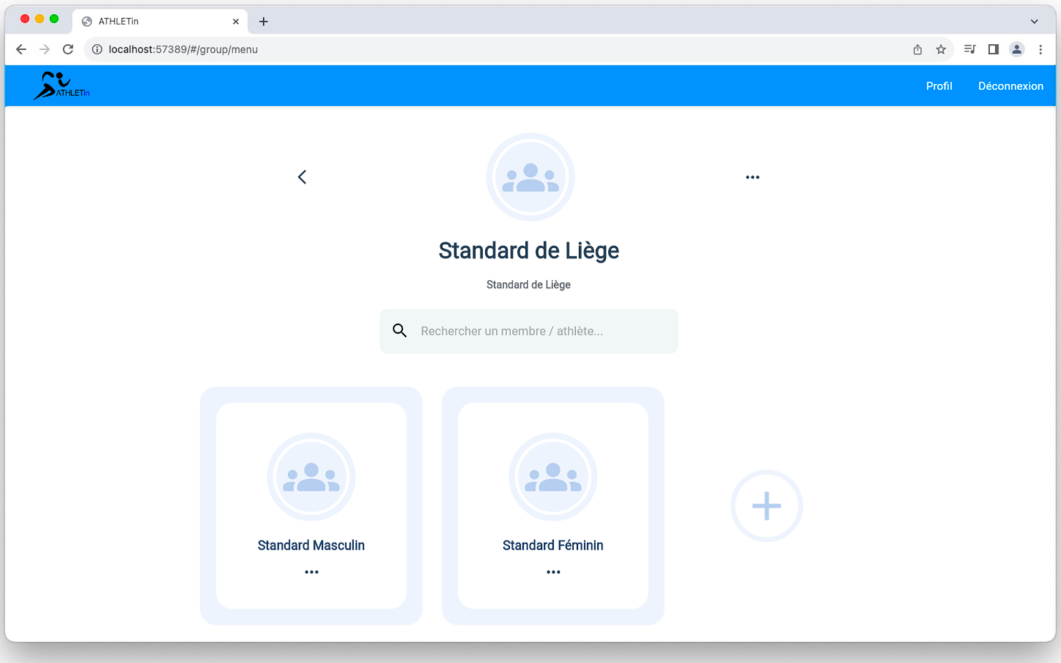


Figure B.3: Group menu page

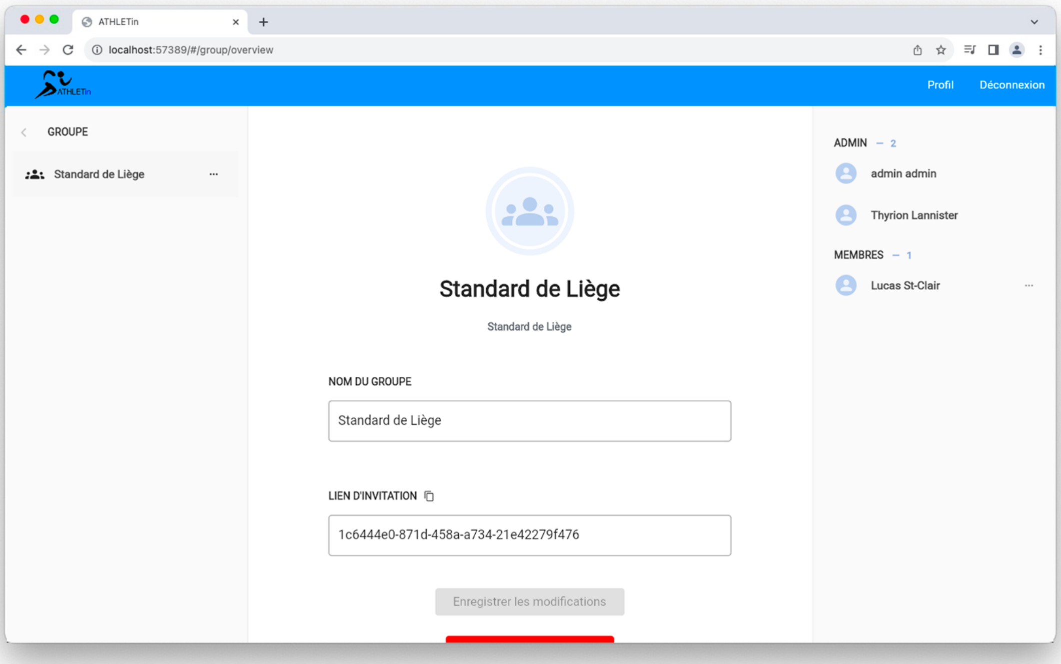


Figure B.4: Group overview page

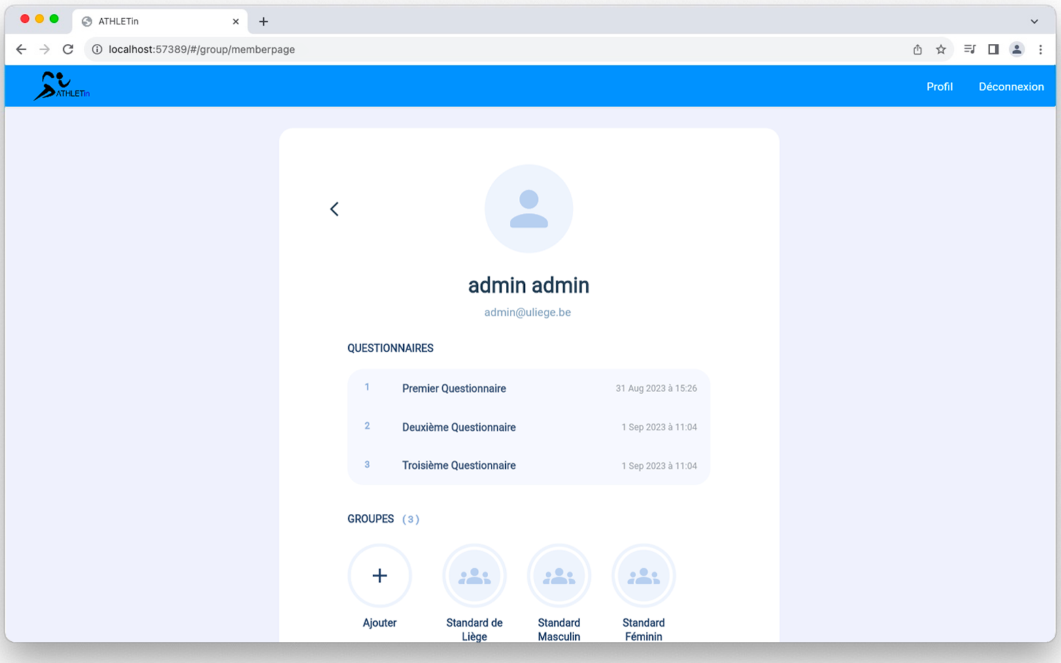

ATHLETin
localhost:57389/#/group/memberpage
Profil
Déconnexion
admin admin
admin@uliege.be
QUESTIONNAIRES
1 Premier Questionnaire 31 Aug 2023 à 15:26
2 Deuxième Questionnaire 1 Sep 2023 à 11:04
3 Troisième Questionnaire 1 Sep 2023 à 11:04
GROUPES (3)
Ajouter
Standard de Liège
Standard Masculin
Standard Féminin

Figure B.5: Member page

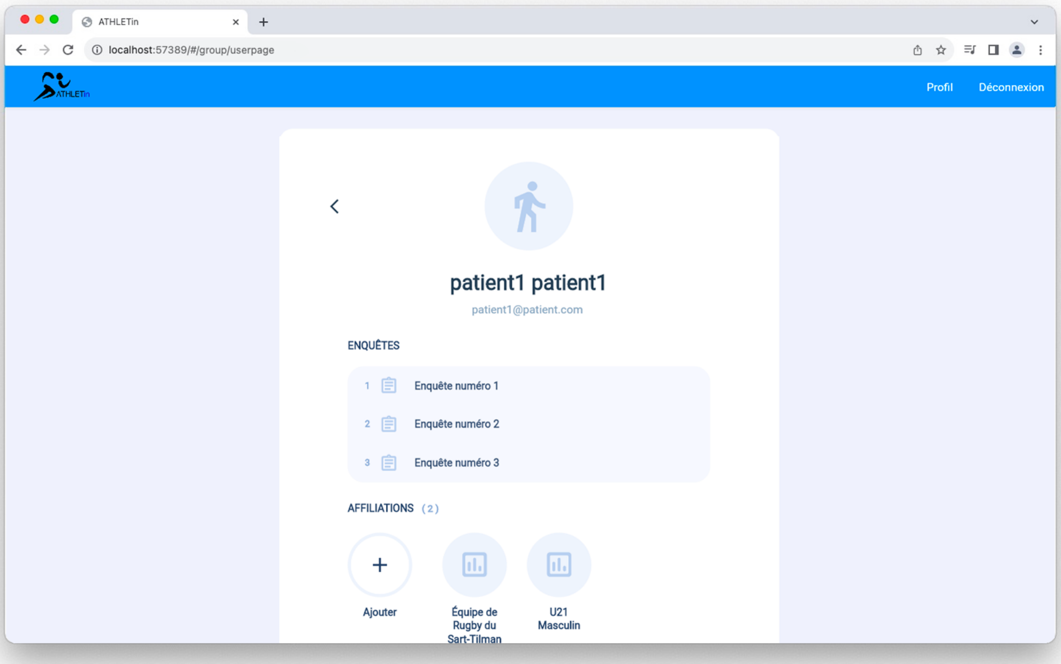


Figure B.6: User page

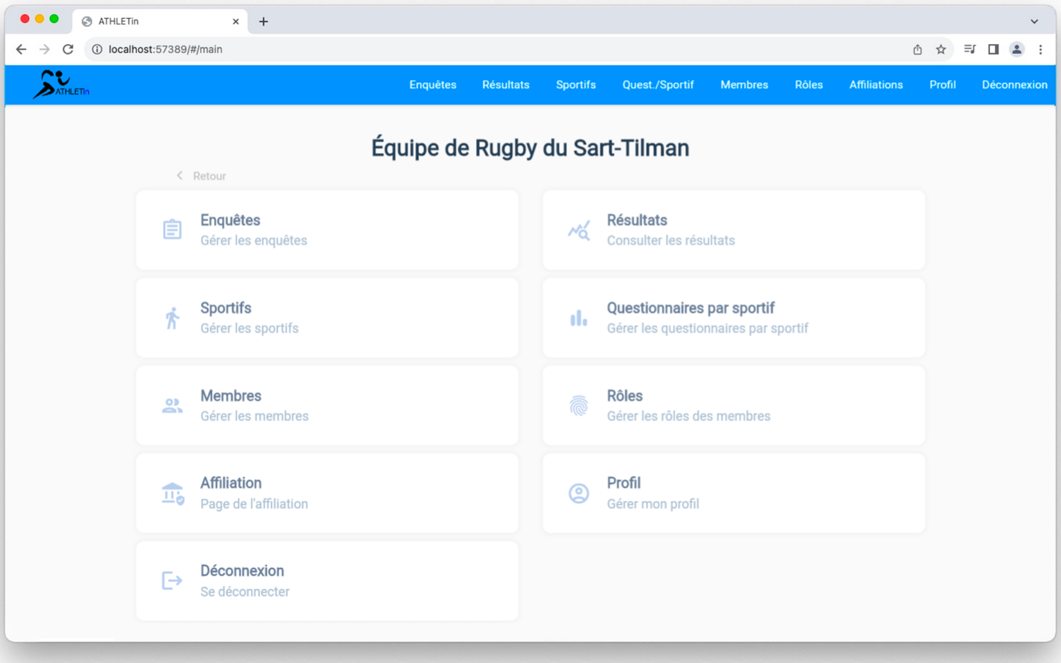
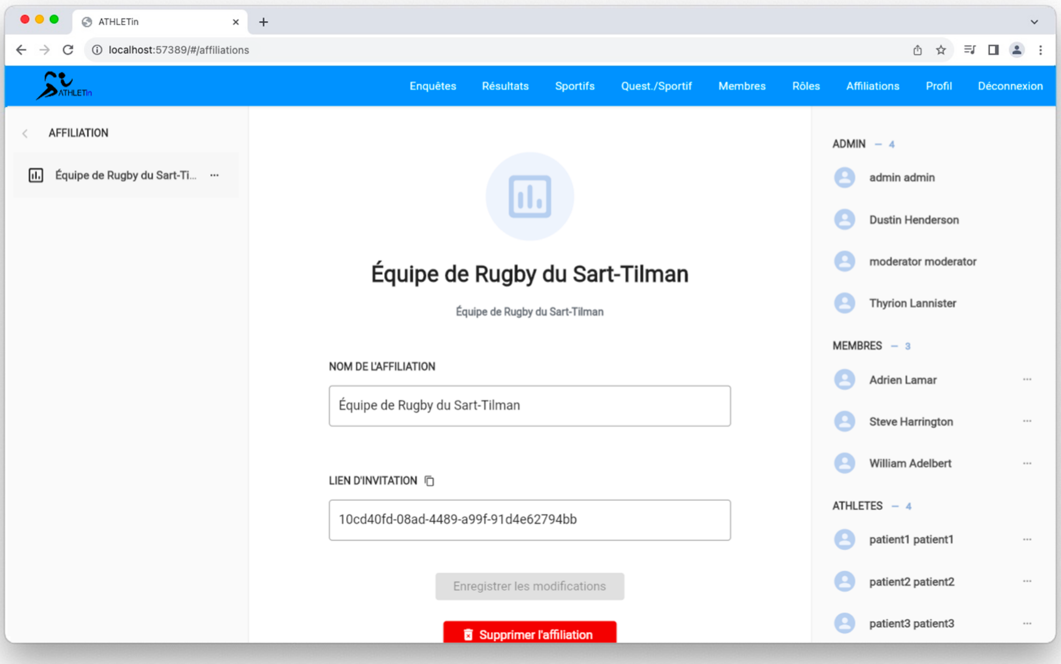


Figure B.7: Affiliation main menu page



Équipe de Rugby du Sart-Tilman

Équipe de Rugby du Sart-Tilman

NOM DE L'AFFILIATION

Équipe de Rugby du Sart-Tilman

LIEN D'INVITATION

10cd40fd-08ad-4489-a99f-91d4e62794bb

ADMIN – 4

admin admin
Dustin Henderson
moderator moderator
Thyrion Lannister

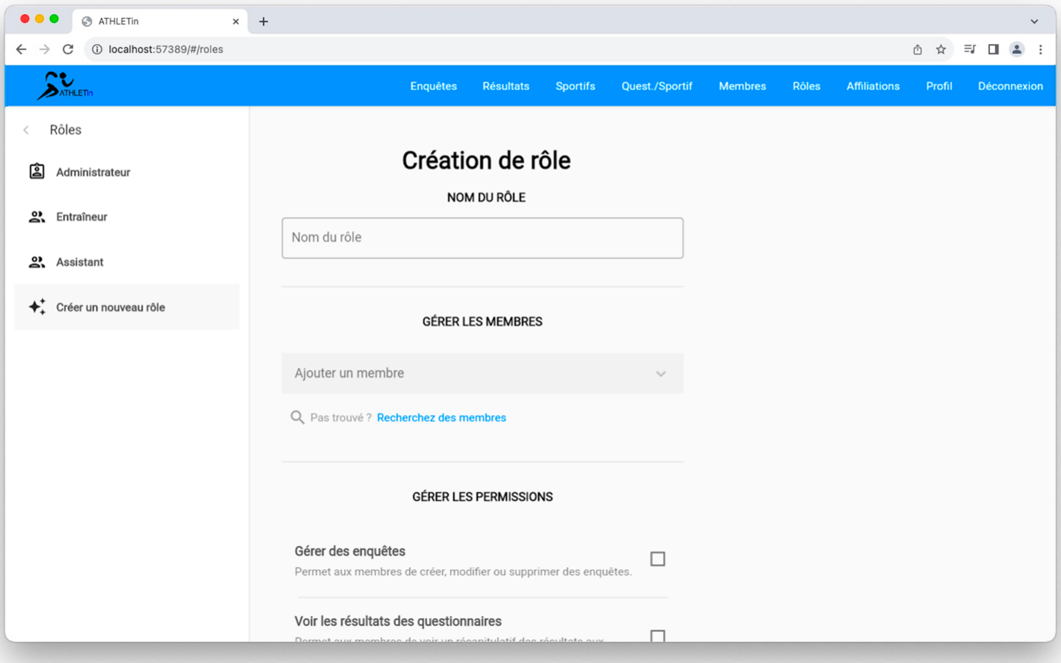
MEMBRES – 3

Adrien Lamar
Steve Harrington
William Adelbert

ATHLETES – 4

patient1 patient1
patient2 patient2
patient3 patient3

Figure B.8: Affiliation overview page



Création de rôle

NOM DU RÔLE

Nom du rôle

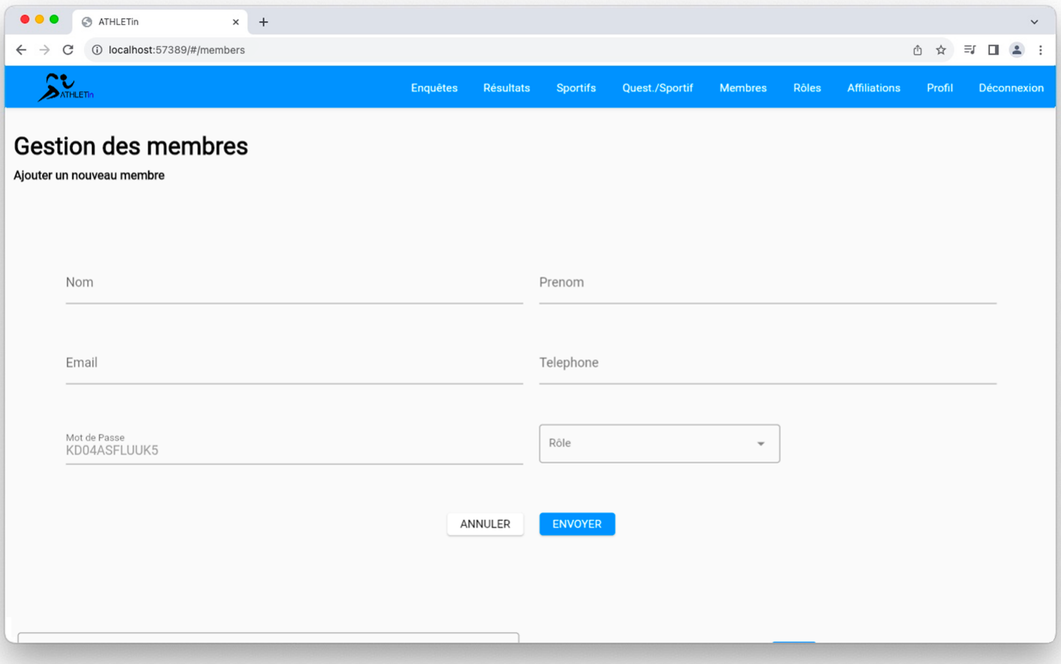
GÉRER LES MEMBRES

GÉRER LES PERMISSIONS

Gérer des enquêtes ☐
Permet aux membres de créer, modifier ou supprimer des enquêtes.

Voir les résultats des questionnaires ☐

Figure B.9: Role creation page

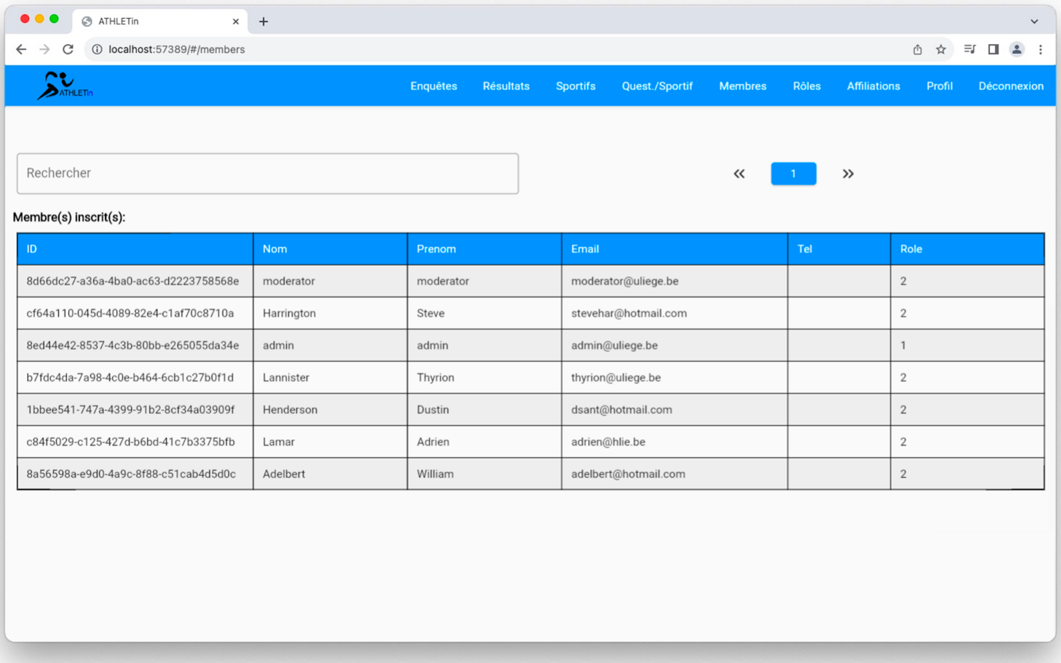


Gestion des membres

Ajouter un nouveau membre

Nom	Prenom
Email	Telephone
Mot de Passe KD04ASFLUUK5	Rôle

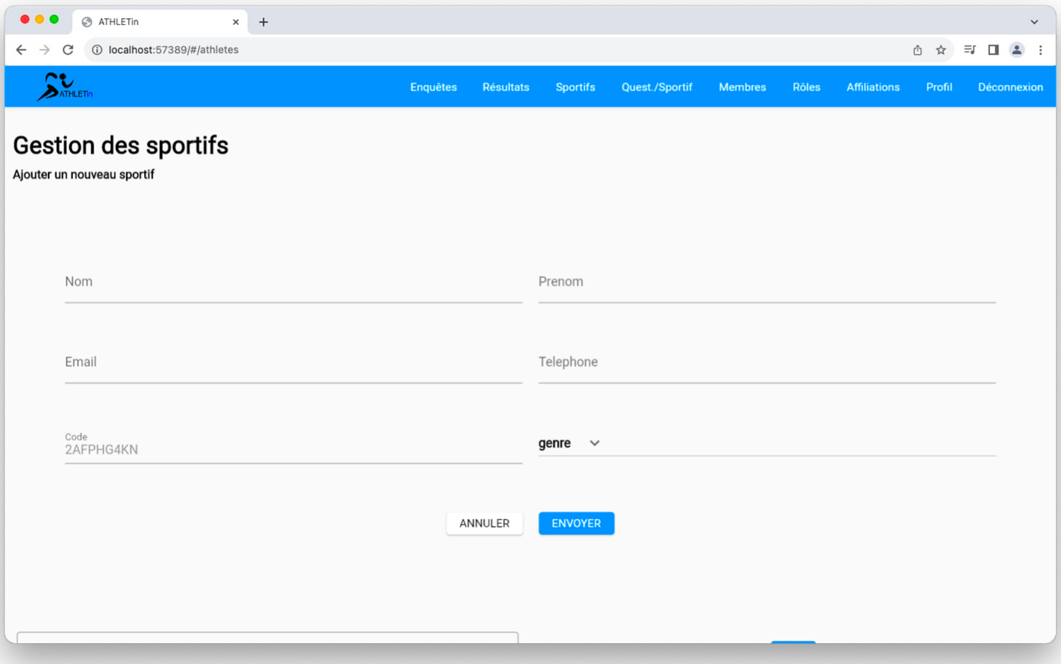
Figure B.10: Members page 1/2



Membre(s) inscrit(s):

ID	Nom	Prenom	Email	Tel	Role
8d66dc27-a36a-4ba0-ac63-d2223758568e	moderator	moderator	moderator@uliege.be		2
cf64a110-045d-4089-82e4-c1af70c8710a	Harrington	Steve	stevehar@hotmail.com		2
8ed44e42-8537-4c3b-80bb-e265055da34e	admin	admin	admin@uliege.be		1
b7fdc4da-7a98-4c0e-b464-6cb1c27b0f1d	Lannister	Thyrion	thyrion@uliege.be		2
1bbee541-747a-4399-91b2-8cf34a03909f	Henderson	Dustin	dsant@hotmail.com		2
c84f5029-c125-427d-b6bd-41c7b3375bfb	Lamar	Adrien	adrien@hlie.be		2
8a56598a-e9d0-4a9c-8f88-c51cab4d5d0c	Adelbert	William	adelbert@hotmail.com		2

Figure B.11: Members page 2/2

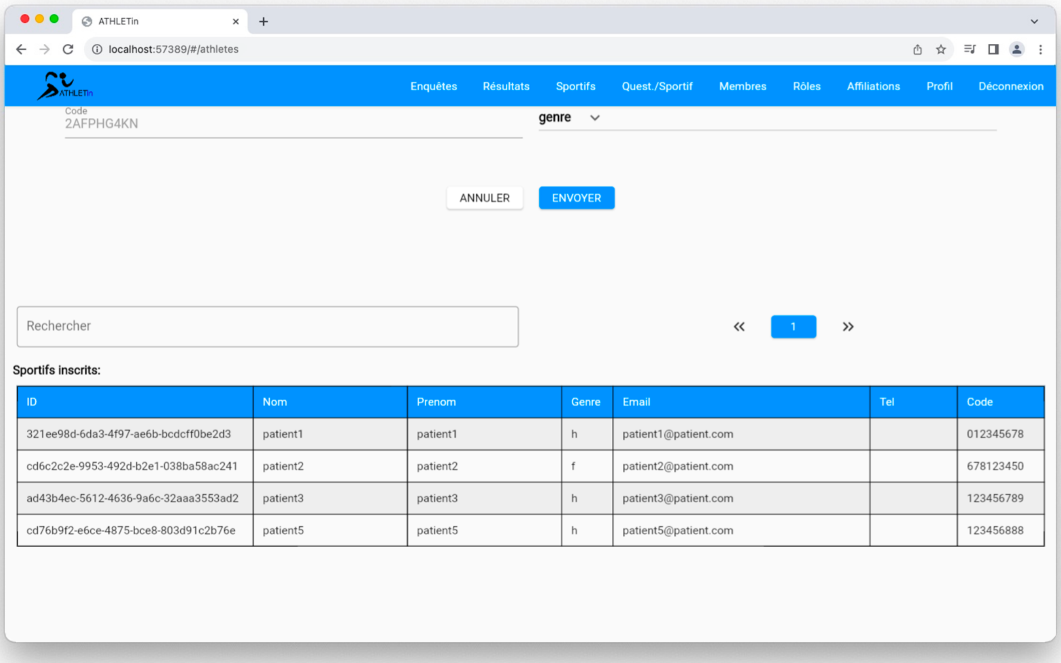


Gestion des sportifs

Ajouter un nouveau sportif

Nom	Prenom
Email	Telephone
Code 2AFPHG4KN	genre

Figure B.12: Athletes page 1/2

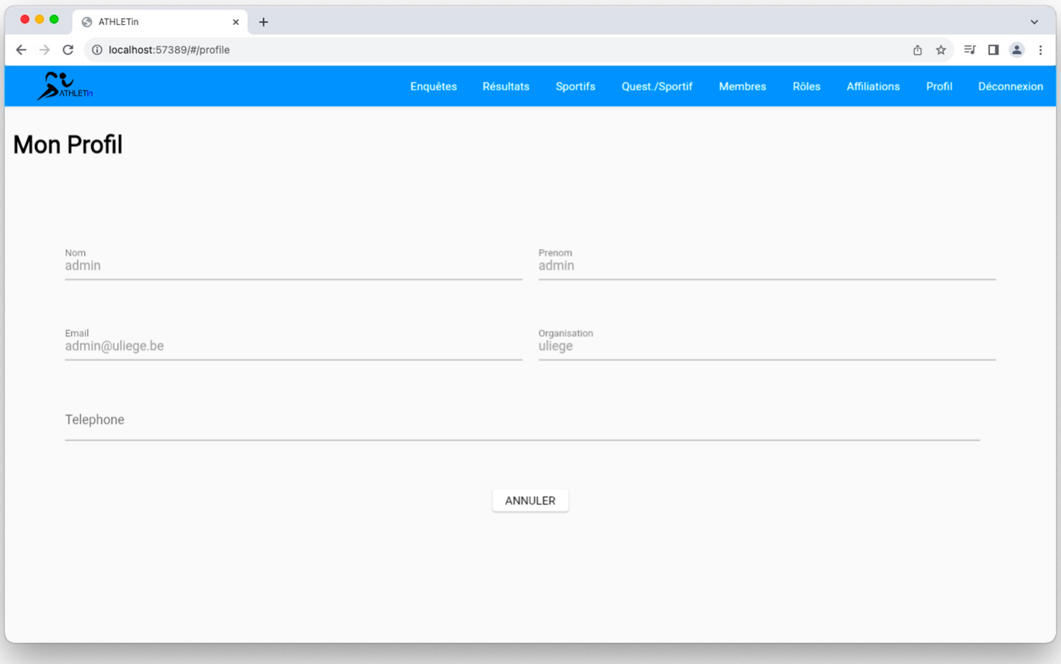


Code 2AFPHG4KN genre

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
cd76b9f2-e6ce-4875-bce8-803d91c2b76e	patient5	patient5	h	patient5@patient.com		123456888

Figure B.13: Athletes page 2/2



Mon Profil

Nom
admin

Prenom
admin

Email
admin@uliege.be

Organisation
uliege

Telephone

Figure B.14: Profile page

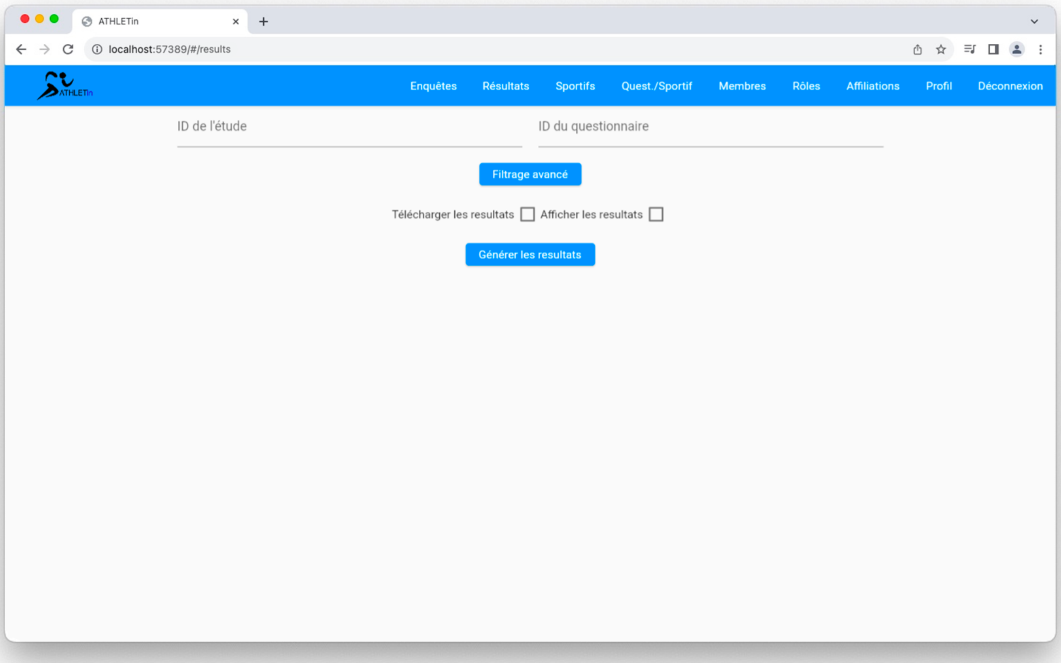
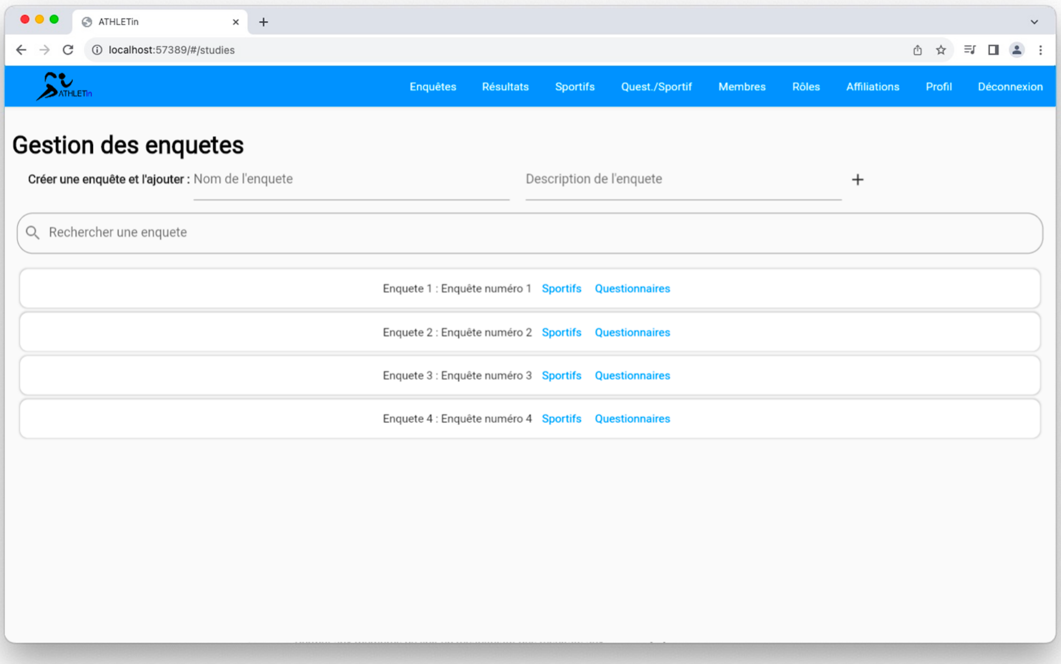


Figure B.15: Results page



Gestion des enquetes

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

Rechercher une enquete

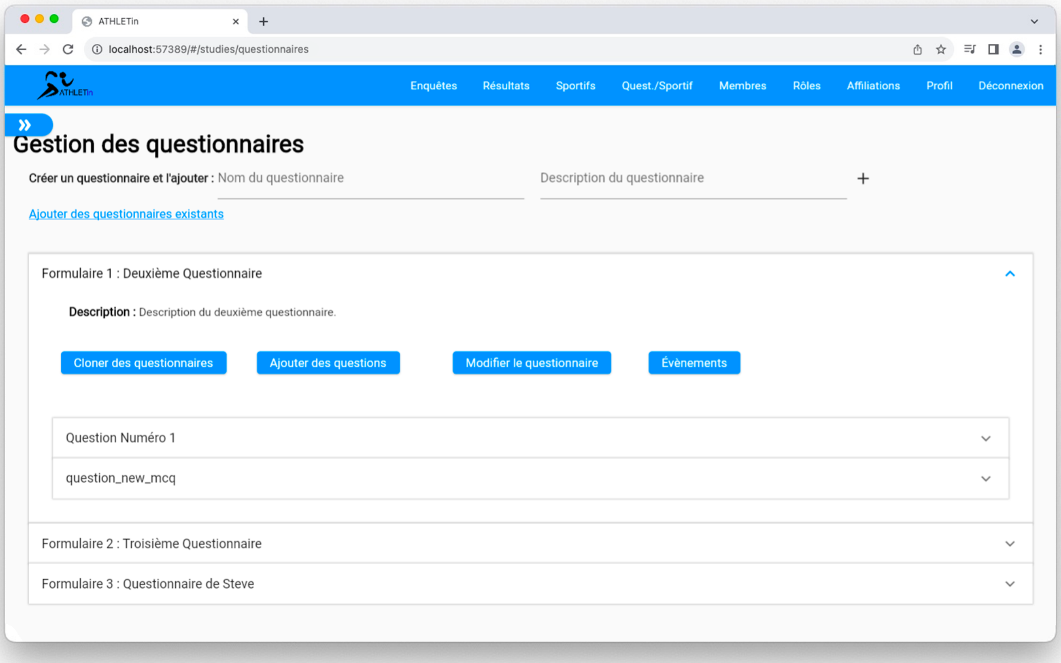
Enquete 1 : Enquête numéro 1 Sportifs Questionnaires

Enquete 2 : Enquête numéro 2 Sportifs Questionnaires

Enquete 3 : Enquête numéro 3 Sportifs Questionnaires

Enquete 4 : Enquête numéro 4 Sportifs Questionnaires

Figure B.16: Studies page



Gestion des questionnaires

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

Ajouter des questionnaires existants

Formulaire 1 : Deuxième Questionnaire

Description : Description du deuxième questionnaire.

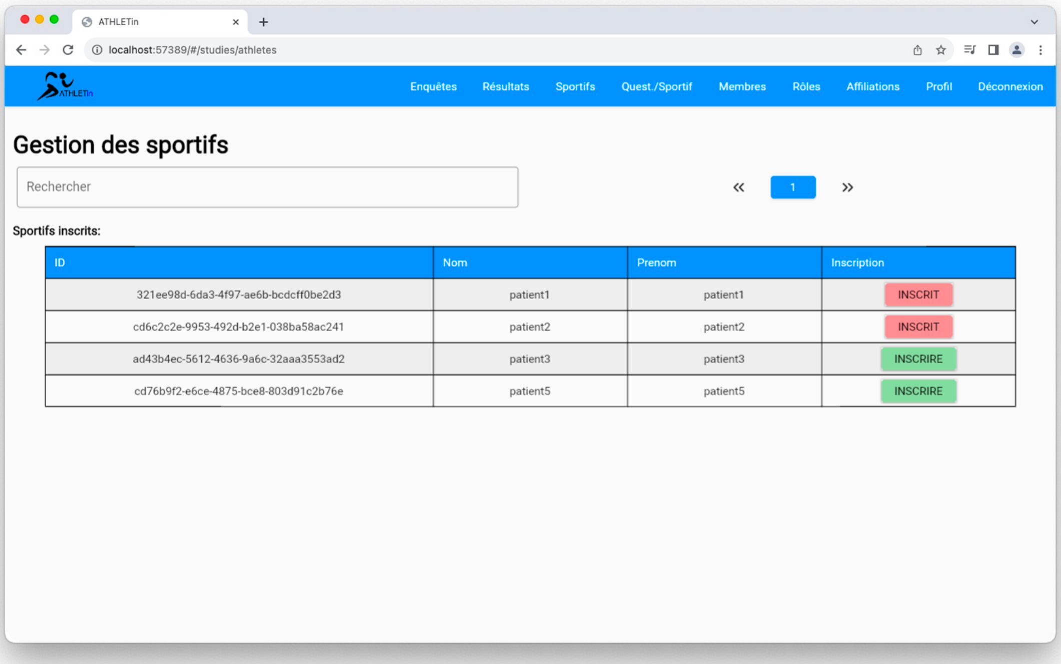
Question Numéro 1

question_new_mcq

Formulaire 2 : Troisième Questionnaire

Formulaire 3 : Questionnaire de Steve

Figure B.17: Studies questionnaires page



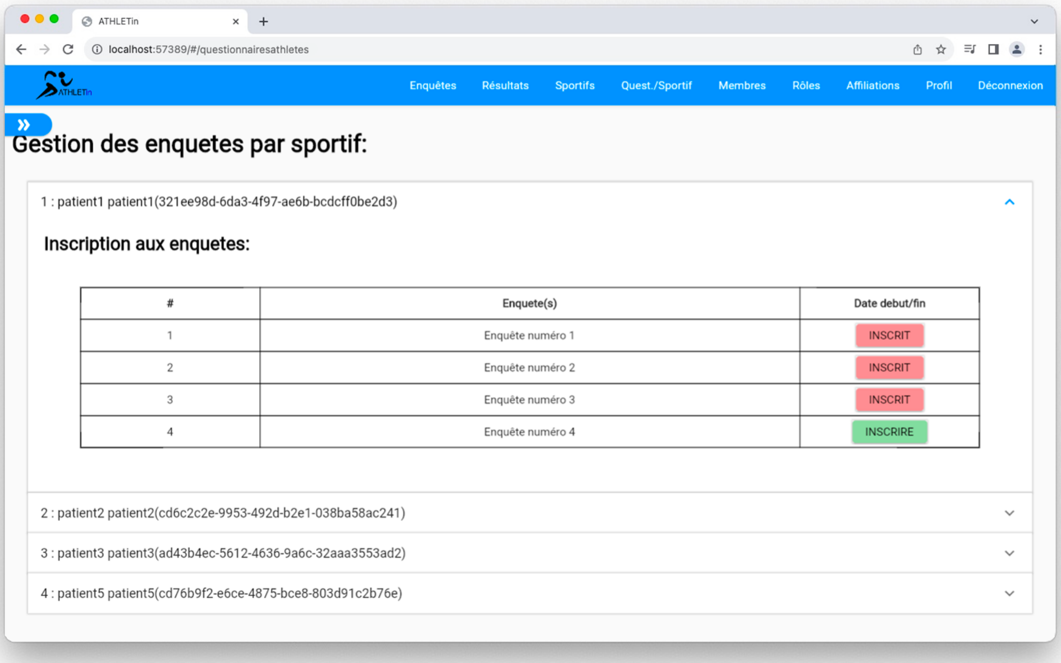
Gestion des sportifs

Rechercher

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
cd76b9f2-e6ce-4875-bce8-803d91c2b76e	patient5	patient5	INSCRIRE

Figure B.18: Studies athletes page



Gestion des enquetes par sportif:

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

Inscription aux enquetes:

#	Enquete(s)	Date debut/fin
1	Enquête numéro 1	INSCRIT
2	Enquête numéro 2	INSCRIT
3	Enquête numéro 3	INSCRIT
4	Enquête numéro 4	INSCRIRE

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

3 : patient3 patient3(ad43b4ec-5612-4636-9a6c-32aaa3553ad2)

4 : patient5 patient5(cd76b9f2-e6ce-4875-bce8-803d91c2b76e)

Figure B.19: Questionnaires athletes page

Bibliography

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

[2] Alakhir Ahmed. *Master thesis : ATHLETin: Development of Administration module for ATHLETin.*

[3] *PostgreSQL database management system.* URL: `https://www.postgresql.org/docs/current/`.

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

[5] *GORM - The fantastic ORM library for Golang.* URL: `https://gorm.io/index.html`.

[6] *Object Relational Mapping.* URL: `https://medium.com/@emccul13/object-relational-mapping-9d84807f5536`.

[7] *What is REST.* URL: `https://restfulapi.net/`.

[8] *Getting Started With Rest API Testing.* URL: `https://testsigma.com/blog/rest-api-testing/`.

[9] *What is Postman.* URL: `https://www.postman.com/`.

[10] *Dart programming language.* URL: `https://dart.dev/`.

[11] *Dart : Sound null safety.* URL: `https://dart.dev/null-safety`.

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

[13] *Dart : Asynchronous programming: futures, async, await.* URL: `https://dart.dev/codelabs/async-await`.

[14] *Dart overview | Dart.* URL: `https://dart.dev/overview`.

[15] *Core libraries | Dart.* URL: `https://dart.dev/guides/libraries`.

[16] *Flutter - Build apps for any screen.* URL: `https://flutter.dev/`.

[17] *Flutter architectural overview.* URL: `https://docs.flutter.dev/resources/architectural-overview`.

[18] *Web renderers | Flutter.* URL: `https://docs.flutter.dev/platform-integration/web/renderers`.

[19] *Material Components widgets | Flutter.* URL: `https://docs.flutter.dev/ui/widgets/material`.

[20] *Cupertino (iOS-style) widgets | Flutter.* URL: `https://docs.flutter.dev/ui/widgets/cupertino`.

[21] *Flutter - List of state management approaches*. URL: https://docs.flutter.dev/data-and-backend/state-mgmt/options.

[22] *Flutter - Differentiate between ephemeral state and app state*. URL: https://docs.flutter.dev/data-and-backend/state-mgmt/ephemeral-vs-app.

[23] *Flutter - setState method*. URL: https://api.flutter.dev/flutter/widgets/State/setState.html.

[24] *Flutter - InheritedWidget class*. URL: https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html.

[25] Daniel Herrera Sánchez. *Flutter Provider: What is it, what is it for, and how to use it?* URL: https://medium.com/bancolombia-tech/flutter-provider-what-is-it-what-is-it-for-and-how-to-use-it-47d6941860d7.

[26] *Hot reload | Flutter*. URL: https://docs.flutter.dev/tools/hot-reload.

[27] *Navigation and routing | Flutter*. URL: https://docs.flutter.dev/ui/navigation.

[28] *Docker overview*. URL: https://docs.docker.com/get-started/overview/.

[29] *What is UUID?* URL: https://www.techtarget.com/searchapparchitecture/definition/UUID-Universal-Unique-Identifier#:~:text=A%20UUID%20(Universal%20Unique%20Identifier,UUID%20generated%20until%20A.D.%203400..

[30] *MVC Design Pattern*. URL: https://www.geeksforgeeks.org/mvc-design-pattern/.

[31] *JSON Web Tokens*. URL: https://auth0.com/docs/secure/tokens/json-web-tokens.

[32] *Go by Example: Regular Expressions*. URL: https://gobyexample.com/regular-expressions.

[33] *Preorder Traversal of Binary Tree*. URL: https://www.geeksforgeeks.org/preorder-traversal-of-binary-tree/.

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

[35] *Swagger*. URL: https://swagger.io/.

[36] *Singletons | Flutter by Example*. URL: https://flutterbyexample.com/lesson/singletons.

[37] *Stack Data Structure: Practical Applications Operations*. URL: https://medium.com/swlh/stack-data-structure-practical-applications-operations-e4e308008752.

[38] *Navigation drawer*. URL: https://m2.material.io/components/navigation-drawer/flutter.

[39] *AlertDialog class*. URL: https://api.flutter.dev/flutter/material/AlertDialog-class.html.

[40] *CheckboxListTile class*. URL: https://api.flutter.dev/flutter/material/CheckboxListTile-class.html.

[41] *Testing In Flutter*. URL: https://medium.flutterdevs.com/testing-in-flutter-fd0f82ecddc7.

[42] *DevTools | Flutter*. URL: https://docs.flutter.dev/tools/devtools/overview.

[43] *Using the Flutter Inspector*. URL: https://docs.flutter.dev/tools/devtools/inspector.

[44] *Using the debugger*. URL: `https://docs.flutter.dev/tools/devtools/debugger`.

[45] *Devtools Presentation | Chrome*. URL: `https://developer.chrome.com/docs/devtools/overview?hl=fr`.

[46] *Google Lighthouse*. URL: `https://www.semrush.com/blog/google-lighthouse/`.

[47] *Debian*. URL: `https://www.debian.org/index.en.html`.

[48] *Ngynx*. URL: `https://www.nginx.com/`.