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

Master thesis : ATHLETin: Web module for the management of athletes' training calendar and medical appointments

Auteur : Lodrini, Guillaume
Promoteur(s) : Mathy, Laurent
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en informatique, à finalité spécialisée en "computer systems security"
Année académique : 2021-2022
URI/URL : http://hdl.handle.net/2268.2/14451

Avertissement à l'attention des usagers :

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

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

Master thesis

ATHLETin: Web module for the management of athletes' training calendar and medical appointments

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

Guillaume LODRINI

University of Liège
Faculty of Applied Sciences
Academic year 2021-2022

Promoter:
Prof. Laurent MATHY

Jury:
Prof. Bernard BOIGELOT
Prof. Guy LEDUC

Abstract

ATHLETin: Web module for the management of athletes' training calendar and medical appointments

Guillaume LODRINI

Master in computer science and engineering
with specialised orientation in "computer systems security"
University of Liège - Academic year 2021-2022

Promoter: **Prof. Laurent Mathy**

As far as injuries are concerned, we commonly say that "prevention is better than cure". However, prevention usually requires a lot of information. This is where the *ATHLETin* solution wants to help. Its originator, Julien Paulus, is convinced that a better communication between athletes, trainers, and health specialists would reduce the risk of injuries. He notably believes that a better overview of the athletes' training program will prevent inadequate training load, and more interactions between doctors and managers will diminish the risk of injury recurrence.

The *ATHLETin* solution proposes a set of modules, each targeting a particular problem. A training module to focus on providing adapted training sessions to each athlete, a medical part to ease the communications with health specialists, and a shared calendar will allow both managers and athletes to organize themselves. This thesis consisted in the design, development, and deployment of solutions for the calendar and medical parts. The solution proposed is composed of a `REST` server interacting with a database, and two web modules. The `REST` server has been implemented in `Go` and communicates with a `PostgreSQL` database. For the web modules, it has been decided to implement them using the Dart language and the Flutter framework.

The main concern for this project was to provide great performances while keeping the modules ergonomic. To achieve this for the `backend`, we added indexes on the database's tables to speed up queries. This reduces the average queries duration and ensures constant performances when tables' sizes increase. For the `Flutter` modules, efficiency has been reached through a well-designed state management strategy. This strategy is based on the Flutter provider mechanism and allows fine-grained control on an application rebuild to avoid unnecessary operations. This thesis also provides several testing strategies, guidelines to improve them, and a `docker` solution for the deployment of the components. As a conclusion, it reviews the work realized and proposes some features to enhance the current system.

This thesis is part of a project started by the professor Laurent Mathy and his team. It had induced some technical restrictions, notably on the technologies to use. However, we still had the freedom of most design choices. The most important was to develop a solution that complies with its current work to smooth as much as possible the future integration.

Acknowledgements

First, I would like to express my gratitude to my supervisor, Prof Laurent Mathy for giving me the opportunity to work on this project and for the continuous support throughout this year.

I wish to extend my special thanks to Gaulthier Gain for always being accessible and providing me with precious advice, and feedback on this thesis.

I want to use this opportunity to also express my deepest gratitude to the Professors of the Faculty of Applied Sciences for sharing their expertise with me and contributing to my academic success. This thesis, along with my five years at university, were a truly rewarding experience. I have learned and seen things in many fields and it provides me with lots of new skills.

Finally, I would also like to thank my family who surrounded me during my studies and always encouraged me for achieving my goals.

Contents

I Introduction **1**

1 Introduction **2**
1.1 Story of *ATHLETin* . . . 2
1.2 Project definition . . . 2
1.2.1 Calendar module . . . 3
1.2.2 Medical module . . . 3
1.2.3 Training module . . . 3
1.3 List of Constraints . . . 3
1.4 Proposed solution . . . 3

II State of the art **5**

2 Backend technologies **6**
2.1 Go . . . 6
2.2 GORM framework . . . 7
2.2.1 Object-Relational Mapping - ORM . . . 7
2.2.2 GORM specifications . . . 7
2.2.2.1 GORM models . . . 7
2.2.2.2 Auto-migration . . . 7
2.2.2.3 Associations . . . 7
2.2.2.4 CRUD interface . . . 8
2.3 REST API . . . 8
2.4 Postman . . . 9

3 Frontend technologies **10**
3.1 Dart . . . 10
3.1.1 Dart native . . . 11
3.1.2 Dart on web . . . 11
3.1.3 Dart runtime . . . 11
3.1.4 Asynchronous programming . . . 12
3.2 Flutter . . . 13
3.2.1 Flutter architecture . . . 13
3.2.1.1 The Embedder . . . 13
3.2.1.2 The Engine . . . 14
3.2.1.3 The Framework . . . 14
3.2.2 Flutter building blocks : Widgets . . . 15
3.2.2.1 Stateless and Stateful widgets . . . 17
3.2.3 Rendering model . . . 17

3.2.3.1 Native application rendering model . . . 17
3.2.3.2 Other cross-platform frameworks rendering model . . . 17
3.2.3.3 Flutter rendering model . . . 18
3.2.4 Rendering pipeline . . . 19
3.2.5 State management . . . 20
3.2.5.1 The setState method . . . 20
3.2.5.2 The provider method . . . 21
3.2.5.3 The GetIt method . . . 22
3.2.6 Hot reload functionality . . . 23
3.2.7 Packages and Plugins . . . 23

III Implementation **24**

4 Requirements **25**
4.1 Functional requirements . . . 25
4.1.1 Calendar module requirements . . . 25
4.1.2 Medical module requirements . . . 26
4.1.3 Summary diagrams . . . 28
4.2 Platform requirements . . . 29

5 High level architecture and Database design **31**
5.1 Project architecture . . . 31
5.2 Database design . . . 32

6 REST server **37**
6.1 Software architecture . . . 37
6.1.1 Controller-Model pattern . . . 37
6.1.2 Files organization . . . 37
6.2 Performances . . . 38
6.2.1 Indexes . . . 38
6.2.2 Analysis of indexes efficiency . . . 39
6.3 Authentication and Routing . . . 39
6.4 Swagger for documentation . . . 41
6.5 Postman as testing platform . . . 41

7 Flutter modules **42**
7.1 MVVM Architectural Pattern . . . 42
7.2 Interaction with REST server . . . 43
7.3 Files organization . . . 45
7.4 Performance tools . . . 45
7.4.1 Flutter profile mode . . . 46
7.4.2 Dart DevTools . . . 46
7.4.3 Tracking widget rebuilds . . . 47
7.4.4 Performance overlay . . . 47
7.5 Calendar module . . . 48
7.5.1 Calendar page . . . 48
7.5.2 Event details view . . . 49
7.6 Medical module . . . 49
7.6.1 Injuries table . . . 50

7.6.2 Injury details . . . 50
7.6.3 Statistics . . . 51

8 Testing **53**
8.1 REST server testing . . . 53
8.1.1 Go testify package . . . 53
8.1.2 Unit tests . . . 54
8.1.2.1 Testing template . . . 54
8.1.2.2 Test coverage . . . 55
8.1.3 Performance tests . . . 56
8.2 Flutter Web modules testing . . . 57
8.2.1 Widget testing . . . 58
8.2.2 Integration testing . . . 58

9 Deployment **59**
9.1 Docker to encapsulate the modules . . . 59
9.2 Nginx as a reverse proxy . . . 60
9.3 Starting scripts . . . 61

IV Conclusion **62**

10 Conclusion **63**

11 Future work **65**
11.1 Add pagination . . . 65
11.2 Improve responsiveness . . . 65
11.3 Upgrade it to Flutter 3.0 . . . 65
11.4 Internationalizing the web modules . . . 65
11.5 Improve the current role system . . . 65
11.6 Instant messaging in medical module . . . 66

A Backend part **67**
A.1 Database architecture . . . 68
A.2 Full Database overview . . . 69
A.3 Databases indexes . . . 70
A.4 Test coverage results . . . 71

B Flutter web modules **74**
B.1 Calendar module screenshots . . . 74
B.2 Medical module screenshots . . . 78

List of Figures

2.1 ORM mapping . . . 7

3.1 `Dart` platform compilation [12] . . . 11
3.2 Asynchronous programming: callback method . . . 12
3.3 Asynchronous programming: async/await method . . . 13
3.4 `Flutter` layers [18] . . . 14
3.5 Widget lifecycles [19] . . . 16
3.6 JavaScript based application rendering model compared to `Flutter` application rendering model [19] . . . 18
3.7 From initial widget tree to the render tree [18] . . . 19
3.8 `Flutter` constraints and sizes propagation in the tree [19] . . . 20

4.1 Calendar module use cases . . . 28
4.2 Medical module, statistics use cases . . . 28
4.3 Medical module, injuries use cases . . . 29

5.1 High level view of the interactions between the different components . . . 32
5.2 Internal structure of the database, Calendar part (Primary keys are colored in yellow and Foreign keys in blue) . . . 33
5.3 Internal structure of the database, Medical part (Primary keys are colored in yellow and Foreign keys in blue) . . . 33

6.1 MC pattern . . . 37
6.2 Result of the performance test on the update events route . . . 40
6.3 Result of the performance test on the get event athletes route . . . 40

7.1 MVVM pattern . . . 43
7.2 APIresponse object . . . 44
7.3 Loading indicator while waiting for a request to finish . . . 44
7.4 Alert dialog because of a bad authentication token . . . 45
7.5 Flutter widget rebuild tracking tool . . . 47
7.6 Flutter frames rendering time tracking tool . . . 47
7.7 Calendar module agenda, overview (event day display) . . . 48
7.8 Calendar module event details, overview . . . 49
7.9 Medical module overview . . . 50
7.10 Medical module injury details, overview . . . 51
7.11 Medical module statistics overview . . . 52

8.1 Coverage result for the event model . . . 55
8.2 Result of the performance test on the get member events route (Mr.Paulus estimations) . . . 57

8.3 Result of the performance test on the get member events route (Our estimations) . . . 57

9.1 Organization of the project files . . . 60
9.2 High level overview of the deployment case. Nginx used as a reverse proxy to handle different routes . . . 61

A.1 Internal structure of the database (Primary keys are colored in yellow and Foreign keys in blue) . . . 68
A.2 Whole database overview . . . 69
A.3 Covering percentage for models . . . 72
A.4 Covering percentage for controllers . . . 73

B.1 Calendar module agenda, overview (event list display) . . . 74
B.2 Calendar module agenda, overview (event month display) . . . 75
B.3 Calendar module agenda, multi-select filter . . . 75
B.4 Calendar module event details, data view . . . 76
B.5 Calendar module event details, status view . . . 76
B.6 Calendar module event details, modify event view . . . 77
B.7 Calendar module add a new event 1 . . . 77
B.8 Calendar module add a new event 2 . . . 78
B.9 Medical module injury details, diagnostics . . . 78
B.10 Medical module injury details, add a new diagnostic (wound) . . . 79
B.11 Medical module injury details, add a new diagnostic (wound) 2 . . . 79
B.12 Medical module injury details, add a new diagnostic (disease) . . . 80
B.13 Medical module modify a diagnostic . . . 80
B.14 Medical module injury details, circumstances . . . 81
B.15 Medical module injury details, consultations . . . 81
B.16 Medical module injury details, add a new consultation . . . 82
B.17 Medical module injury details, modify a consultation . . . 82
B.18 Medical module injury details, authorizations . . . 83
B.19 Medical module injury details, add new authorizations . . . 83
B.20 Medical module injury details, modify injury dates . . . 84

List of Tables

3.1 High-level feature comparison among `Dart`, Java, and JavaScript 10

5.1 Architecture of the database, tables of the Calendar module 34
5.2 Architecture of the database, tables of the Medical module 36

6.1 Files organization of the Go server 38

7.1 Files organization of the Flutter modules 46

A.1 List of the indexes defined in the database 71

List of Abbreviations

AOT Ahead Of Time.

API Application Programming Interface.

ARM Advanced RISC Machines.

CPU Central Processing Unit.

CRUD Create Read Update Delete.

DB DataBase.

GDPR General Data Protection Regulation.

HTML HyperText Markup Language.

HTTP HyperText Transfer Protocol.

ID IDentifier.

IDE Integrated Development Environment.

JIT Just In Time.

JS JavaScript.

JSON JavaScript Object Notation.

JWT JSON Web Token.

LBFR Ligue Belge Francophone de Rugby.

MVVM Model-View-View Model.

ORM Object-Relational Mapping.

OS Operating System.

REST REpresentational State Transfer.

RFC Request For Comments.

SDK Software Development Kit.

SQL Structured Query Language.

SSL Secure Sockets Layer.

UI User Interface.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

UUID Universally Unique Identifier.

VM Virtual Machine.

Part I

Introduction

Chapter 1

Introduction

This introductory chapter goes over the *ATHLETin* story, from the context where the idea of *ATHLETin* was born up to the current state of the project. This introduction also reviews the *ATHLETin* project definition and the few constraints on the project, to finally expose the proposed solution.

1.1 Story of *ATHLETin*

The story of *ATHLETin* started about 10 years ago. At this time, Julien PAULUS, the originator of *ATHLETin*, is the strength and conditioning coach of athletes at the rugby training center in the Sart-Tilman. As a physical trainer, Mr.PAULUS has to prepare the training sessions for different players. The problem is that players have multiple "training contexts". Indeed, most of the players train during the week at the training center, and have a rugby match with their own team during the weekend, coupled with possible training sessions or competitions with the national team. This makes the management and the following of each athlete very difficult.

To solve this management issue, a co-worker of Mr.PAULUS developed a web application called *MyLBFR*. Initially designed for the Sart-Tilman training center, the application grew and became more and more complete. However, sportive administrations in Belgium refused to invest in the project. This lack of financial support ultimately led the designer of the platform to stop managing it.

However, Mr.PAULUS is convinced that bad training load management is the number one cause of injuries, and he believes that a well-designed platform would tackle this problem. Lacking the necessary skills to develop the solution himself, Mr.PAULUS gets in touch with Xavier PICARD, asking him to become the head of the commercial part of the project. The latter, contacted the professor Laurent MATHY to handle the development part. This is where we are today, the project is currently being developed at the *ULiège* university by both the professor Laurent MATHY, his team, and some students including myself.

1.2 Project definition

As explained in the section 1.1, the application mainly targets the management of athlete training sessions. On top of that, the application should handle the review and manage-

ment of athlete injuries. Finally, an agenda should also be available to allow managers to see their athletes' programs, and athletes to see their events.

1.2.1 Calendar module

The calendar module, or agenda module, must provide a way for both athletes and managers to manage events. The athletes should be able to visualize the events they are involved in and notify their eventual absence. From the managers' perspective, they should have the possibility to create events, attach athletes to them, and visualize all the events related to the athletes they supervise.

1.2.2 Medical module

The medical part is intended to be used by health professionals (doctors, physiotherapists ...). Currently, doctors are using shuttle sheets to keep information about athlete injuries. The medical module aims at the digitization of those sheets. In essence, this module should allow doctors to create injuries when athletes get hurt, attach them diagnostics, and define the authorized activities depending on the state of the injury. On top of that, doctors must also have the possibility to create appointments with injured athletes from this platform. Finally, some statistics should be computed from the injuries stored to analyze recurrent problems and find solutions.

1.2.3 Training module

After each training session, athletes will be asked to fulfill a questionnaire linked with the session completed. This module aims at displaying the results and analyzing them through some automatic statistical computation. On top of that, it should also allow for adding data from performance capacity tests to know the training zone of each athlete.

1.3 List of Constraints

Mr.Paulus had very few constraints concerning the development of the project. The most important things for him were that the platforms should be ergonomic, and easy to understand.

The professor Laurent Mathy already started the project when my thesis began. He asked me to reuse the same technologies to ease the future integration. Thus, the web platforms and the mobile application are implemented in Dart (see section 3.1) with the Flutter (see section 3.2) framework. Concerning the `REST` server, it is implemented in Go (see section 2.1) and the database management system is PostgreSQL.

1.4 Proposed solution

To implement all those different modules, the proposed solution is to develop

- A web platform for each module. Those platforms will be used by the managers to fulfill their tasks.
- A mobile application for the athletes, from which they can see their agenda and fulfill their questionnaires.

This thesis is about the calendar and medical modules. Therefore the implementation part III will only cover those two modules.

> After this small introduction about *ATHLETin* to set the context in which this thesis fits, we will expose the two main parts of this thesis. The first one will describe the tools used throughout the project while the second will deal with the implementation details of the proposed solution.
>
> The following chapter provides a list of the `backend` technologies before having a look at what has been used for the `frontend` part.

Part II

State of the art

Chapter 2

Backend technologies

In software engineering, the term `backend` refers to the data access layer of a piece of software, the physical infrastructure, or the hardware. Said otherwise, we could consider the `backend` as the server in a client-server architecture. In our case, this corresponds to the database and the `REST API`. In this chapter, the different `backend` technologies used throughout the development of the project are described, starting from the language used up to the platform allowing to test requests.

2.1 Go

The `Go`[1] programming language has been described as "the C for the 21st century"[2]. It is a compiled and statically typed language with `C`-like structures that yet has the convenience of garbage collection and feels like a dynamically typed, interpreted language [3]. `Go` has been conceived fairly recently (in late 2007) by Google based on an idea from Robert Griesemer, Rob Pike (former Bell Lab researcher), and Ken Thompson (one of the Unix creators) as an answer to some of the problems seen by Google's software developers [4].

This language was not designed as a breakthrough research language but more as a "software engineering" language [5]. It has been created with a willingness to make things efficient and simple. This makes `Go` an easy to understand and learn language with as key functionalities:

1. A strict error checking policy (it is an error to import unused packages or declare unused variables)
2. Type inference
3. No implicit type conversion
4. Built-in currency mechanism through `goroutine` and channels
5. Simplified syntax with few keywords (semicolon removed, no parentheses for function arguments, possible to return multiple values at once, etc)

Moreover, some frameworks, such as `GORM`, exist to ease the use of Go as a language to design databases. This is discussed in section 2.2.

2.2 GORM framework

2.2.1 Object-Relational Mapping - ORM

Object-Relational Mapping is a programming technique that allows communication and interaction with a database using the Object-Oriented paradigm. In other words, an `ORM` sets the mapping between objects written in any Object-Oriented language to the corresponding `SQL` queries. This allows for better readability and understanding of code, leading to easier maintenance. The only drawback is that the level of abstraction over the way the data is being stored and the request are made can sometimes make the debugging of the code slightly harder. However, using a logger, it is possible to visualize the requests perform by the `ORM` and understand what is wrong.

Figure 2.1: ORM mapping

2.2.2 GORM specifications

`GORM`[6], as the name suggests, is a full-featured `ORM` designed for the `Go` programming language. `GORM` provides official support for `sqlite`, `mysql`, `postgres`, and `sqlserver` [6]. In the following, I will develop the features that I used during the project, but of course, the framework allows many more than that.

2.2.2.1 GORM models

Models are normal `structs` with basic `Go` types, pointers/alias of them or custom types [6]. `GORM` favors conventions over configurations and will convert `Go struct` names and their variables to snake_cases respectively as tables and columns name.

2.2.2.2 Auto-migration

`GORM` auto-migration functionality handles the creation of tables, missing foreign keys, constraints, columns, and indexes. Said otherwise, it is enough to create the database tables, to declare the `Go structs` and use the auto-migration.

2.2.2.3 Associations

`GORM` allows defining associations between tables. This is especially useful in the `many-to-many` case. Combined with the auto-migration functionality, this ease the creation of relationships between tables. Indeed, to create a relation, one can simply define in one

of the two object structures a table of the other object. The framework will handle all the details to create the corresponding mapping table. It also provides methods to automatically loads associations when reading records.

2.2.2.4 CRUD interface

`GORM` allows for `CRUD` [1] operations.

To create a record, it lets the possibility to create it from an object, from an object but inserting into the database only a subset of the object fields, or from a `map[string]interface{}`[2]. The last case is especially useful to create records from data in `JSON` format. In case the record created involves associations, the corresponding values are upserted and the hook methods[3], if any, are invoked.

To read, `GORM` provides a way to query the database for a particular record using any of its field(s) as filtering condition. It also provides a way to recover all the records of a table or all the records of a table matching a particular condition.

To update a record, `GORM` makes it possible to update all the fields of a record or only specific columns. As for creation, this can be performed from either an object or a `map[string]interface{}`.

To delete, it allows performing a delete operation using the primary key as the matching condition or a batch delete where all the records matching a condition are deleted. As for creation, `GORM` lets developers define hooks methods to perform operations just before or just after deletion.

2.3 REST API

A `REST API` (also known as `RESTful API`) is an application programming interface (`API`) that conforms to the constraints of `REST` architectural style and allows for interaction with `RESTful` web services. `REST` stands for representational state transfer [7].

`REST` is a set of architectural constraints, not a protocol or a standard. This means that when developing an `API`, one can implement `REST` in approximately the way he likes. The main guidelines are the following:

- Requests have to be sent using `http` and response must follow the `http` status code.
- Data must be formatted in a specific format. Most of the time the `JSON` (Javascript Object Notation) format is favored[4].
- Authorization is contained in the header of requests.
- Stateless client-server communication, meaning that no info about the client is kept from request to request. Each request is isolated from the others.

[1] Create, Read, Update, Delete
[2] `interface{}` means any type, including custom type
[3] Methods that implements operations to perform before/after creating/saving a record
[4] In the context of this thesis, we use `JSON` format.

2.4 Postman

Postman[8] is an API client that makes it easy for developers to create, share, test and document APIs. This is done by allowing users to create, save, and send simple or complex HTTP/s requests, as well as read their responses [9]. Postman is a very complex and complete tool, but in the scope of this thesis, we used it as a testing platform to test that my API routes and endpoints were working as expected.

> This section presented the list of tools used for the implementation and testing of the REST server which aims at being easily maintainable but also at providing efficiency. Those are the main reasons why the Go language has been selected.
>
> The technologies used for the frontend will be listed in the next chapter.

Chapter 3

Frontend technologies

The term `frontend` involves what is linked with the presentation layer. When speaking about `frontend` we usually denote everything which is related to the client in a client-server model. In the scope of this thesis, this corresponds to the Calendar and Medical modules. This chapter describes the two technologies used to build the `frontend` part of the project. First, it reviews the `Dart` programming language. Then it analyses the `Flutter` framework in details.

3.1 Dart

`Dart`[10] is an open-source, structured programming language. It has been released by Google in 2011 but is still in active development with regular updates. It is a purely object-oriented, class-based language that supports dynamic typing (with both static and dynamic checks), asynchronous operations, and multiple inheritance thanks to `mixins` [11]. `Dart` also offers null safety, meaning that variables can't be null unless this was explicitly stated. This allows for protection from null exceptions at runtime through static code analysis. Finally, it also provides a garbage collector that reclaims unused memory [12].

`Dart` language is syntactically very close to `Java`, and `JavaScript` languages, making it very easy to pick up for developers used to one of those two languages [13].

	`Dart`	`Java`	`Javascript`
Type system	Optional, dynamic	Strong, static	Weak, dynamic
Classes	Yes, single inheritance	Yes, single inheritance	Prototypical
Interfaces	Yes, multiple interfaces(`mixin`)	Yes, multiple interfaces	No

Table 3.1: High-level feature comparison among `Dart`, Java, and JavaScript

As it can be read on the dart web page, "`Dart` is a client-optimized language for developing fast apps on any platform. Its goal is to offer the most productive programming language for multi-platform development, paired with a flexible execution runtime platform for app frameworks". Said otherwise, `Dart` is a language that has been thought and made for multi-platform software development. In addition, it also comes with a large set of core libraries providing most of the fundamentals for the development of web and

mobile applications [12].

Finally, `Dart`'s compiler technology lets you run code in different ways.

Figure 3.1: `Dart` platform compilation [12]

3.1.1 Dart native

For apps targeting mobile and desktop devices, `Dart` includes both a `Dart VM` with just-in-time (`JIT`) compilation and an ahead-of-time (`AOT`) compiler for producing machine code [12]. During development phase, the `JIT` compilation together with incremental compilation allows what we call hot-reloading (see section 3.2.6). Once the app has to be released, the `Dart AOT` compiler enables ahead-of-time compilation to native `ARM` or `x64` machine code. This has the advantage of providing better performances (code executed instead of interpreted) with consistent and shorter startup time (frequently used code has already been compiled and doesn't have to be recompiled every time the app is launched).

3.1.2 Dart on web

For apps targeting the web, `Dart` includes both a development time compiler (dartdevc) and a production time compiler (dart2js). Both compilers translate `Dart` into JavaScript [12]. Then the `JavaScript` code needs to be run in a browser (e.g. V8 engine interprets `JavaScript` for Chrome browser). The dartdevc compiler supports incremental compilation and emits modular JavaScript [14] enabling a fast developer cycle. On the other hand, dart2js compiler tries to compile `Dart` code into fast, compact, and deployable JavaScript using techniques such as dead-code elimination [12].

3.1.3 Dart runtime

Note that, regardless of the platform (except web), or compilation method being used, executing the dart code requires a `Dart VM` that provides the `Dart` runtime which is among other things in charge of managing memory by applying the garbage collector policy. On native platforms, the Dart runtime is automatically included inside self-contained executables [12]. Note that the name "`Dart VM`" is historical. `Dart VM` is a virtual machine

in the sense that it provides an execution environment for a high-level programming language, however, it does not imply that `Dart` is always interpreted or `JIT`-compiled, when executing on `Dart VM` [15].

3.1.4 Asynchronous programming

It is a common situation in software development to perform network calls or disk accesses and wait that those *long tasks* finish before going on. `Dart`, despite being a single-threaded language, provides asynchronous operations to let your program complete work while waiting for another operation to finish [16].

Asynchronous operations provide results as what we call `Futures`. A `Future` has two states: uncompleted, or completed. In the first case, the asynchronous operation is still going on, and the `Future` is waiting for the operation to finish in order to provide a result. In the second case, the operation is finished and the `Future` completed with a value. Note that `Futures` are typed variables, `Future<T>` denotes a `Future` that when completed will return a value of type T.

`Dart` essentially provides two different ways to work with `Futures`. The first method consists in using callbacks. This is what is shown on the figure 3.2. With this method, the callback takes as argument a variable with the same type as the `Future` returned, and the callback method is executed once the `Future` completes.

```
return http.get(url, headers: headers).then((http.Response response) {
  // We assume that an empty response means that the request found no corresponding endpoint.
  // Ideally, the server should be the one to handle this.
  final int statusCode = response.statusCode;
  String source = Utf8Decoder().convert(response.bodyBytes);
  final jsonData = response.body.isNotEmpty
      ? jsonDecode((source))
      : jsonDecode('{"message": "Route not found","status": false}');
  if (statusCode < 200 || statusCode >= 400) {
    return APIresponse(
        error: true,
        errorMessage: "${statusCode.toString()} ${jsonData["message"]}");
  } else {
    return APIresponse(data: jsonData["data"]);
  }
});
```

Figure 3.2: Asynchronous programming: callback method

The second solution is to use the `await` and `async` keywords.

- async: You can use the `async` keyword before a function's body to mark it as asynchronous[16].
- async function: An `async` function is a function labeled with the `async` keyword[16].
- await: You can use the `await` keyword to get the completed result of an asynchronous expression. The `await` keyword only works within an `async` function[16].

An `async` function will run synchronously until the first `await`. Said otherwise, it executes all the synchronous code before the `await`, then waits for the `Future` to complete, and

will restart upon completion. This method provides the exact same functionality as the previous one but has the advantage to be much more readable when multiple asynchronous operations arise in the same method. The figure 3.3 shows the same code as the figure 3.2 rewritten with async/await method.

```
http.Response response = await http.get(url, headers: headers)
// We assume that an empty response means that the request found no corresponding endpoint.
// Ideally, the server should be the one to handle this.

final int statusCode = response.statusCode;
String source = Utf8Decoder().convert(response.bodyBytes);
final jsonData = response.body.isNotEmpty
  ? jsonDecode((source))
  : jsonDecode('{"message": "Route not found","status": false}');

if (statusCode < 200 || statusCode >= 400) {
  return APIresponse(
    error: true,
    errorMessage: "${statusCode.toString()} ${jsonData["message"]}");
} else {
  return APIresponse(data: jsonData["data"]);
}
```

Figure 3.3: Asynchronous programming: async/await method

3.2 Flutter

`Flutter`[17] is an open source framework based on `Dart` released by Google in 2017 for building natively compiled, multi-platform applications from a single codebase. Just as `Dart`, `Flutter` is still under development and updates are published more or less regularly [18]. `Flutter` is special in that it makes it truly possible to "write once, and deploy everywhere". Initially, it was possible to deploy `Flutter` apps to Android, iOS, and ChromeOS. On top of that, `Flutter` 2.0 (released in march 2021) and `Flutter` 3.0 (released in may 2022) made it possible to run `Flutter` apps as web apps and desktop apps on all major operating systems [19][20].

From a `UI` development point of view, `Flutter` is a reactive, declarative, and composable view-layer library that has some similarities with tools such as `ReactJS` or `SwiftUI`. In short, developers create `UI` by assembling groups of smaller components, the widgets.

3.2.1 Flutter architecture

`Flutter` has been designed into three extensible layers. Those layers exist as independent libraries and each rely upon the underlying one. No layer has privileged access to the lower layer, and every part of the framework level is optional and can be replaced[18].

3.2.1.1 The Embedder

To the underlying operating system `Flutter` applications are managed in the same way as any other native application. The `Flutter` platform-specific embedder provides a way to coordinate with an `OS` for accessing specific services such as rendering surfaces, or input. As it is a multi-platform framework, `Flutter` has to provide an appropriate embedder for

Figure 3.4: `Flutter` layers [18]

each platform it supports. Currently, embedders are written in `Java` and `C++` for Android, `Objective-C/Objective-C++` for iOS and macOS, and `C++` for Windows and Linux. This way, one can embed `Flutter` code as a module to another application or use its code as a complete application[18].

3.2.1.2 The Engine

The engine layer is written in `C/C++`. In short, it takes care of file and network requests and handles the task of (re)rendering whenever a frame needs to be painted. `Flutter` uses `Skia`, an open-source 2D graphics library that abstracts platform-specific graphics `API`, as its rendering engine. The engine is exposed to the framework via the `dart:ui` library which wraps the corresponding `C/C++` code in dart classes so that it can be accessed by developers easily if needed[18].

3.2.1.3 The Framework

This is the classic way developers interact with `Flutter`. The framework is written in `Dart` and provides modern and reactive interactions. This part is composed of a rich set of platform, layout, and foundational libraries organized into different layers. This could be described from bottom to top as

1. Primary fundamental classes and building blocks including animation, gesture de-

tection, and painting.

2. A rendering layer that brings abstraction to build a tree of renderable objects. The `RenderObject` class is the base class for every object that needs to be painted on screen. It defines an abstract model for layout and painting. Every painted element must necessarily inherit from the `RenderObject` class, and each type of element (Text, Image, Structural element ...) has its `Rendering class`. Notes that a `RenderObject` knows very few about its children other than how to visit them and their constraints.

3. The widgets layer introduces the reactive programming model. Each class in the widget layer corresponds to an object in the rendering layer and thus can be displayed on screen. On top of that, the layer allows, to define custom combinations of classes. This is discussed in more detail in the section 3.2.2.

4. Finally, the `Material` and `Cupertino` libraries offer an all-inclusive set of `Flutter` widgets that respectively implement the Material and iOS design languages. These two libraries are the ones that developers usually import when designing applications in `Flutter`.

Still, the `Flutter` framework is relatively small and many high level features that developers might need are implemented as packages (see section 3.2.7)[18][21].

3.2.2 Flutter building blocks : Widgets

In `Flutter`, "everything is a widget", said otherwise every element displayed on screen is a widget. Behind the name, a widget is simply a `Dart` class that knows how to represent itself. The structure of an element is defined with the widget, together with its style, animations and anything that someone may think of that is being part of UI [19]. The central idea is that widgets are the building blocks of the UI. `Flutter` provides a rich catalog of built-in widgets [22] that developers can reuse to create their own widgets to eventually describe a unique UI.

Concerning the built-in widgets provided by `Flutter`, most of them can be categorized in two parts :

- Organizational widgets. Those widgets describe how elements should be organized on screen or the style of elements, e.g: Row(), Column(), Stack(), Container(), Padding() ...

- Data widgets. Those widgets represent some data, or object, that should be displayed on screen, e.g : Image(), Text(), ElevatedButton() ...

To create complex designs, developers organize widgets in a tree with parent-child relationships. The process of composing widgets together into a tree is done by telling widgets that their child(ren) are more widgets [19]. Usually, the widget you find at the top of the tree are organizational widgets and the leaves are data ones.

Concerning widgets methods and object members, they have only a few of both. The *main method* is the `build()` method. "Every widget" should declare this method. This method describes the view associated with the widget by returning a tree made of lower-level widgets. The other life-cycle methods are only available to Stateful widgets. The

`State.initState` method, is the method responsible to initialize all data needed by a widget before `Flutter` tries to paint it on screen. The `dispose` method is the opposite of the `initState` one, it is used to tell an object to stop using particular resources. This method is being called when "State object will never build again" [23]. This is the final method of the widget life-cycle. Finally, the `setState()` method should be called whenever a State object is modified. This method aims at signaling the framework to update the `UI` by calling the State's `build` method again [19] [18].

Figure 3.5: Widget lifecycles [19]

Pay attention, "everything is a widget" can be misleading. This doesn't mean that a `Flutter` program includes only widgets. It means that every element that appears on-screen maps to a widget but developers might also use `Dart` objects to manage widgets' state or program logic.

3.2.2.1 Stateless and Stateful widgets

When creating a new application, developers commonly author new widgets that are subclasses of either `StatelessWidget` or `StatefulWidget`. The difference between the two lies in the name. Stateful widgets keep information about their internal state whereas a Stateless doesn't have an internal state. One could say that a Stateful widget is dynamic whereas a Stateless one is immutable. The first one can be modified over time whereas the second is built once and for all.

In the 3.2.2 part we said that every widget should declare its build method. However, for a Stateful widget, the way it works is a bit different than simply implementing the method in the widget object. Every Stateful widget is associated with its corresponding state object which does implement the build method. One could see the pair Stateful widget/state object as a single component. Actually, a Stateful widget just like a Stateless one is immutable, and it is the state object that holds all the mutable information[19].

Note that it is faster to build a Stateless widget than a Stateful one since fewer methods are invoked for the build. In addition, by using a constant Stateless widget instead of Stateful we reduce the number of components to be analyzed when having to rebuild (since it knows that Stateless are immutable) which ultimately fastens the rebuild.

We don't insist further on the difference between those two types of widgets because, as we will discuss in section 3.2.5, in practice it is possible to use Stateless widgets only and just rebuild specific parts of widgets once mutable information is modified.

3.2.3 Rendering model

The rendering model is the way `Flutter` manages to transform a tree of widgets to their actual representation on screen. This section, although it is not essential to understand `Flutter` or to be a `Flutter` developer, is interesting to see and understand what is going on behind the hood. Furthermore, this section also shows how a cross-platform framework is optimized to provide efficient performances that are comparable to what we could get from a native application.

3.2.3.1 Native application rendering model

Before looking at how `Flutter` deals with rendering it is useful to start by thinking about how traditional native applications work. If we take the example of an Android application, when rendering, it first calls the `Java` code of the Android framework. The Android system libraries come up with components that allow them to draw themselves to a Canvas object. Then, Android uses Skia which will handle the remaining work to render elements on the device. We would like `Flutter` to be as close as possible to this rendering scheme to get performances that are closed to one of native applications [18].

3.2.3.2 Other cross-platform frameworks rendering model

Concerning cross-platform frameworks, usually, they work by adding an abstraction layer on top of the native application. This new layer tries to abstract the differences that may exist between the different platforms supported. The application code is usually written in an interpreted language such as JavaScript which must in turn deal with a platform-specific language, Java/Kotlin for Android applications or Objective-C/Swift

for iOS ones. All this creates overhead due to what we call "the JavaScript bridge" which is the major bottleneck to mobile frameworks in JavaScript. Indeed since the application is not compiled to machine code, to interact with a specific platform, JavaScript code must be compiled on the fly while the app is running to the targeted platform language. This creates additional work to perform before rendering objects on the device, which can be significant, particularly when there is a lot of interaction between the UI and the app logic (because the UI has to be re-rendered regularly) [18]. Furthermore, this might also make debugging more difficult in case of runtime error since the error will have to be traced back across the bridge up to the right JavaScript piece of code [19].

3.2.3.3 Flutter rendering model

`Flutter` tries to minimize abstractions to reduce as much as possible the overhead. To achieve this, it bypasses the platform-specific UI widget libraries in favor of its own widget set. In addition, the `Dart` code that paints `Flutter`'s visuals is compiled into native code from the start. This is an example of `AOT` performed by `Flutter` which prevents from having to compile on the fly some `Dart` code. Finally, `Flutter` also embeds its own copy of the graphical engine Skia as part of its engine layer (see section 3.2.1, Engine layer). This has the main advantage of allowing applications developers to upgrade their app to keep up on the latest `Skia` performance updates event if the phone on which the app runs hasn't been updated to the latest version of its operating system [18].

Figure 3.6: JavaScript based application rendering model compared to `Flutter` application rendering model [19]

3.2.4 Rendering pipeline

Concerning the rendering pipeline, `Flutter` starts by building the widget tree of the widget that has to be built. In this step, `Flutter` might introduce in the widget tree new widgets, as necessary, based on the widgets' state. For instance, if the widget tree includes a Container() with a color, `Flutter` will introduce in the tree a ColoredBox representing the color. This is why the widgets identified by `Flutter` inspector might have a structure much deeper than what the original `Flutter` code has [18].

Once this task is done, `Flutter` translates the widget tree into its corresponding element tree. This tree has the same structure as the tree built previously where each widget has been mapped to one of the two following elements :

- ComponentElement, a host for other elements.
- RenderObjectElement, an element that participates in the layout or painting phases.

The RenderObjectElement are intermediaries with the RenderObject (see section 3.2.1, Framework, rendering layer). During the build phase, `Flutter` creates or updates an object that inherits from RenderObject for each RenderObjectElement in the element tree [18].

Figure 3.7: From initial widget tree to the render tree [18]

Once the render tree has been established, `Flutter` starts thinking about how the widgets should be organized. To establish the layout, `Flutter` explores the render tree in a depth-first traversal. On the way down, constraints are passed from parent to child. Then, according to the constraints received, a child can decide about its size and responds by passing up a size within the constraints to the parent. Thus `Flutter` has a O(N) way to decide about object layout [18]. However, this is not because `Flutter` has an efficient rendering pipeline that it is not beneficial to take advantage of some mechanism in code to reduce even more the time necessary to render the UI (see section 3.2.5).
Once all this has been performed, `Flutter` can finally paint objects on screen.

Figure 3.8: `Flutter` constraints and sizes propagation in the tree [19]

3.2.5 State management

State management, with respect to `Flutter`, denotes the activity of taking care of the state of widgets[1]. A widget can see its state being modified by many different actors such as another widget, or network, and we would like to make sure that modifications are handled correctly and visible on screen.

Concerning the states, we distinguish two types. On one hand, an ephemeral state is a state that can neatly be contained in a single widget. On the other hand, a state that is not ephemeral is a state that is going to be shared across many parts of your app, and that might be needed to be saved across user sessions. We call this an application state[25].

In the following, we expose three state management techniques, The provider and GetIt techniques that we use in this thesis and the setState technique which we don't use but which is the most basic technique. For each technique, we also argue why/when it is a good or bad approach.

3.2.5.1 The setState method

This method consists in using a Stateful widget and saving the state as a/some variable(s) inside the state object related to the widget. Then, upon modification of state variables, one should use the `setState()` method to perform a rebuild. This approach is by far better suited for ephemeral states rather than for application states. To use this method to handle application states we have two main solutions. A first solution would be to share the widget state objects as global variables so that they can be modified from anywhere in the project. The other solution would be to declare the state variables in the closest common widget ancestor between the widgets that must share a state. Then, we can pass callback methods as arguments through the widget tree to be able to modify the state variables from "outside". In summary,

+ Easy to use and implement for ephemeral state cases.

− With the global variable solution we might end up with a lot of dependencies and tracking them manually can quickly become cumbersome and error-prone when the number of actors increases.

[1] According to `Flutter`, State is information that (1) can be read synchronously when the widget is built and (2) might change during the lifetime of the widget [24]. In other words, the state is any information related to the `UI` that you might need to rebuild the app.

- The common ancestor solution might require passing callbacks and values as arguments through a very long list of widgets, which again is cumbersome and error-prone.
- Mix logic and UI code.
- Difficult to have fine-grained control over what is rebuilt.

3.2.5.2 The provider method

The provider approach is a wrapper around `InheritedWidget`[2]. In a few words, an `InheritedWidget` is a widget that makes a resource directly accessible to all its children in the widget tree. Thus, all the children of this widget inherit the property it exposes. The provider method provides `InheritedWidget` capabilities through four concepts,

- `ChangeNotifier`
- `ChangeNotifierProvider`
- `Consumer`
- `Provider.of`

The `ChangeNotifier` is a simple class[3] included in the `Flutter SDK` which provides change notification to its listeners. Said otherwise, if a class is a `ChangeNotifier`, any other class can subscribe to it and listen to its changes. With the provider method, `ChangeNotifier` is a way to encapsulate your application state. Compare to the setState method this has the advantage of separating the state and the logic from the `UI`. For very simple apps, you get by with a single `ChangeNotifier`. In complex ones, you'll have several models, and therefore several `ChangeNotifiers` [27].

The `ChangeNotifierProvider` is a widget that provides an instance of a `ChangeNotifier` to its descendants. It means that it is possible to create multiple instances of the same `ChangeNotifier` at different locations in your project, to manage multiple independent ephemeral states. But it also allows sharing the same instance between several listeners, to deal with an application state. This widget has been implemented in the provider package. The `ChangeNotifierProvider` widget must be placed above the widgets that need to access it but not higher than necessary to avoid polluting the scope [27].

The `Consumer` is a widget that listens to a particular instance of a `ChangeNotifier` exposed by a `ChangeNotifierProvider`. A `Consumer` must specify the type of model it wants to access. This way the provider package will go up the widget tree and find the closest `ChangeNotifierProvider` that corresponds to the type requested. The `Consumer` widget must provide a builder as argument for its constructor. The builder is a function that returns a widget and is called whenever the related `ChangeNotifier` changes [27].

The `Provider.of<T>` is a method that allows obtaining the nearest `Provider` of type `<T>` in the widget tree. Sometimes, you don't need the data in the model to change the

[2]The implementation of this state management technique is a community achievement that has been approved and adopted by the `Flutter` development team. This package is open source and can be found on the *pubdev* website [26]

[3]`Dart` class, not widget.

`UI` but you still need to access it [27]. For instance, in an E-commerce website removing the content of the cart doesn't require displaying it, it just requires calling a method (e.g `clear()`) that reset the content in the cart state. This could be done with a `Consumer` but this would lead to an unnecessary rebuild of the `UI`.

In summary, it is possible to maintain a state with the `ChangeNotifier`, expose it with the `ChangeNotifierProvider`, and listen to its change with the `Consumer`.

+ Allows to manage both ephemeral and application states easily
+ Clear distinction between `UI` code and application logic
+ Fine grained control over what is rebuild through the `Consumer` builder
+ Approved and recommended by `Flutter`
+ Scale well
+ Allows to use Stateless widgets only
− Learning curve a bit steeper than setState method for `Flutter` beginners

3.2.5.3 The GetIt method

First, note that the `GetIt` package in itself is not a full state management solution as the Provider solution is. `GetIt` is a locator solution that allows retrieving objects from different locations in a program[28]. We decided to explain it in this section as it can be used to build a state management solution but we simply didn't use it for this purpose.

As we just discussed, `GetIt` is a service locator that has a central registry where it is possible to register classes and then obtain instances of those classes. The first step is to create the central registry. This central must be easily and quickly accessible from everywhere. We achieve this by declaring a global `GetIt` variable. Then we must register our classes. To do so `GetIt` offers several methods.

- `registerFactory`, every time the service is accessed the method returns a new instance of the corresponding class.
- `registerSingleton`, creates a singleton[4] of a class. Then returns always the same instance on request. To use this method one must provide an instance of the class to the `registerSingleton` method.
- `registerLazySingleton`, same behavior as the `registerSingleton` method except that the instantiation of the class is delayed to the time the object is first time requested. Here one can simply indicate the class of interest, it is not necessary to provide an instance of it. This has the advantage to save resources in case a class is never used or if a process takes time to create and it is preferable to not start it at the launch of the application.

Once the classes have been registered, it is possible to access them in O(1) via the global `GetIt` variable and the name of the class of interest. To sum up,

+ Easy and fast access to classes (O(1)).

[4]Design pattern that restricts the instantiation of a class to one "single" instance

+ Easy to learn, understand and put in place.

− Requires additional material to be used as a complete state management solution.

3.2.6 Hot reload functionality

`Flutter`'s hot reload feature helps to quickly and easily experiment, build `UIs`, add features, and fix bugs. Hot reload works by injecting updated source code files into the running `Dart` Virtual Machine (`VM`). After the `VM` updates classes with the new versions of fields and functions, the `Flutter` framework automatically rebuilds the widget tree, allowing you to quickly view the effects of changes [29]. In other words, `Flutter` hot reload functionality allows developers to modify their applications and see the changes almost live. This functionality is one of the key functionalities of `Flutter`. This increases the development speed since it is not required to recompile the application at every modification to see the resulting `UI`. The tools that allow this functionality are described in the 3.1.1 and 3.1.2 sections.

3.2.7 Packages and Plugins

As previously said, `Flutter` still lacks some high-level features to support operations such as HTTP requests, or interactions with hardware. To overcome this problem, `Flutter` supports the use of shared packages written by other developers. More than making some high-level features available to the community, this also allows building applications without having to recreate everything from scratch which speeds up the development phase [30]. Existing packages can be found on the *pubdev* website [26] which allows to search for packages providing specific functionalities and gives each package a grade over 130 telling if the package follows `Flutter` conventions, provides documentation, passes static tests and so on.

> This chapter has reviewed in detail the most important concepts of the Dart language and its associated framework Flutter. This concludes the state-of-the-art part, and the reader should now be aware of all the technologies used in the context of this thesis.
>
> The next part will go through the implementation details of the different components constituting this project.

Part III

Implementation

Chapter 4

Requirements

This is the first chapter on the implementation of the solution. However, before truly diving into implementation details, it is important to describe the requirements that will guide the development phase. This is the purpose of this chapter. It will describe the features required by each platform without concern for technical details, and provide summary diagrams of the possible use cases for each module.

4.1 Functional requirements

4.1.1 Calendar module requirements

The calendar module aims to be used by managers. It has the following specifications:

- When accessing the calendar, a manager should be able to:
  - **Visualize** all the events he is managing,
  - **Create** new events,
  - **Delete** existing events,
  - **Modify** existing events,
  - **Access** events' details easily.
- An event should be composed of:
  - a **Date and hour**: starts of the event,
  - a **Duration**,
  - a **Title**: name of the event,
  - a **Thematic**: description of the event,
  - a **Location**,
  - a **Message** (optional): additional information for the athletes,
  - a **List of athletes** registered for the event,
  - a **List of categories**[1],
  - a **Type** (selected from a list): describes the purpose of the event (e.g. training, medical appointment...).

[1] In the context of this application, a category is another word for a team. Said otherwise, a category is simply a group of athletes.

- It should be possible to **switch** between different calendar styles: **day**, **week**, **month**, **list**.
- It should be possible to **filter the events displayed** by, **athletes**, **categories**, and **types**.
- The details of an event should be divided into 4 parts:
  - An **overview part** should display a **summary** of the event,
  - A **data part** should **list the athletes** registered in the event, and make it possible to **manage** them. This includes **adding** and **removing** athletes, **modifying** their status[2], **getting information** about what each athlete did during the event and **visualizing** athletes' response to **post training questionnaires**[3],
  - A **status part** should **display** the present and missing athletes ergonomically,
  - A **modify part** should allow to modify the event.

4.1.2 Medical module requirements

The medical module aims to be used by health specialists. It has the following specifications,

- When accessing the medical module the user should be able to:
  - **Visualize** a table that lists **all the injuries** of the athletes he supervises,
  - **Create a new injury** for a specific athlete and select if it is a **recurrent** injury or not,
  - **Delete** an injury,
  - **Access** the details of an injury,
  - **Visualize** some **statistics** about the injuries.
- The injury table should display for each injury,
  - The **injured athlete**,
  - **Date** when the injury started,
  - **Current authorizations**,
  - **Next authorizations**,
  - **Diagnostic** of the injury.
- It should be possible to **filter** the injuries by, **Status**[4], **Athletes**, **Diagnostics**, and **Authorizations**.
- The details of an injury should be splitted in 7 different parts:
  - An **overview part** should recap the injury details,

[2]Present or Missing

[3]The two last points are outside the scope of this thesis since they concern the mobile application and the training module

[4]Injury closed or in progress

  - A **diagnostic part** should allow to manage the diagnostic(s) of the injury. This includes, **creating**, **removing**, and **modifying** diagnostics.
  - A **circumstances part** should allow to fulfil the circumstances where the injury occurred,
  - A **consultation part** should allow to **manage** the consultations related to the injury. This includes **creating** new consultations, or **modifying** existing ones,
  - An **authorization part** should allow to **add**, **remove**, and **modify** authorizations,
  - A **discussion part** should allow doctors and managers to exchange messages in a instant messaging similar to *Messenger* or *WhatsApp*,
  - A **closing part** should allow to **modify** the starting and ending date of the injury.

- Concerning the **diagnostics**, they can either describe a **wound** or a **disease** and they should be composed of:
  - a **Body part** (selected from a list)
  - a **Side** (selected from a list)
  - a **Pathology** (selected from a list)
  - a **Diagnostic** (selected from a list)
  - a **Sub-diagnostic** (optional) (selected from a list)
  - an **Occurrence way**
  - the **Consultation** (event) where the diagnostic was made
  - a **Description**

- A circumstance should be composed of the following fields,
  - A **Date**
  - A **Circumstance** (selected from a list)
  - A **Field type** (selected from a list)
  - A **Description**

- When **adding** a new **authorization** or **modifying** an existing one, the **participation** to events of the athlete concerned by the authorization should be **updated** accordingly.

- Concerning the statistics part, it should display four circular charts which describe **percentage of injuries** occurring per: **body part**, **side**, **pathology**, and **circumstance**. Then, for each circular chart, according to the value selected, it should be possible to visualize the average and the median value of the number **unavailable days**, the number of **missed training sessions**, and the number of **consultations**. Finally, it should be possible to select the athlete for which we would like to see statistics.

4.1.3 Summary diagrams

The use cases regarding the calendar module and the medical module have been summarized in diagrams, respectively the figure 4.1 for the calendar, and the figures 4.2 and 4.3 for the medical module.

Figure 4.1: Calendar module use cases

Figure 4.2: Medical module, statistics use cases

Figure 4.3: Medical module, injuries use cases

4.2 Platform requirements

As stated in the introduction, the modules developed will be integrated in the platform created by the professor Laurent Mathy, and his team. Thus, they should be implemented in `Dart` and `Flutter`. On top of that, some parts of the web modules might be directly integrated into the mobile application. Hence, to ease the integration, all the packages used should be compatible with mobile devices. Finally, the code must run on the latest stable `Dart` and `Flutter` release. This implies that the code should comply with the null safety mechanism.

Concerning the `backend` technologies, the professor Laurent MATHY and his team have already designed a database. Therefore, the database developed in this thesis should fit into the existing design and try to reuse as much as possible existing content to ease the future interaction between new modules and existing functionalities.

> In this chapter, we have seen the list of features that should be implemented as well as the interactions between a user and a module thanks to the use case diagrams.
>
> Now that the specifications of the project have been clearly defined, we will review its organization.

Chapter 5

High level architecture and Database design

This chapter aims to provide the reader with a better understanding of how the project is organized. It starts with a description of the different components of the project and shows how they interact together. Then, it reviews the database design and describes the purpose of the created tables.

5.1 Project architecture

The figure 5.1 represents the high-level architecture of the project. It depicts the different components of the architecture and shows the interactions between them.

1. The **REST Server in Go** is the interface that allows interactions with the database. This server exposes a `REST API` for other components to retrieve stored information. It interacts with the database through `SQL` queries. Those `SQL` queries are generated by the `ORM` (`GORM`) used to implement the `REST` server (see section 2.2). By allowing only a single trusted entity to communicate with the database, we decrease the risk of security issues. The chapter 6 covers in more detail the different sections of this component.

2. The **PostgreSQL database** stores information for both the mobile application and the web modules. The design of the database is depicted and explained in the section 5.2

3. The **Web server** entity consists in the web modules describes in the section 1.2. Those modules are implemented in `Flutter` and interact with the `REST` server through `HTTP` requests. All the modules are independent in the sense that they are hosted at different addresses. This has the benefit of distributing the users across different platforms according to their needs, which reduces the risk of overloading a platform. Still, the platforms can exchange information through the browser `LocalStorage`. Information stored there is permanent unless it is manually removed, information can be stored in a `JSON` format, and the number of stored entries is unlimited.

4. The **Mobile application**, is also implemented in `Flutter` and interacts with the `REST` server through `HTTP` requests. This component is outside the scope of this thesis. It is managed by the professor Laurent Mathy, and his team. Nevertheless,

the mobile application is intended to be used by athletes for viewing their agenda, advertising their possible absence, and filling the post-training questionnaires. The results from the questionnaires are not directly sent to the corresponding administration platform, but instead, results are stored in the DB and fetched by the platform on request.

Figure 5.1: High level view of the interactions between the different components

5.2 Database design

According to Microsoft, a correct design is essential for achieving your goals in working with a database. A properly designed database provides you with access to up-to-date, and accurate information [31]. However, designing an efficient database is far from being trivial. The designing phase consists in deciding how to organize the data into tables, what fields to use to represent data, and what kind of relationships should exist between the tables. On top of that, as the database should be integrated into existing architecture, the design should take current tables into account and try to reuse them as much as possible to avoid storing twice the same information.

In the context of this thesis, the designing phase was done once for each module. The calendar module database, as being a requirement of the medical module, was designed first. The tables related to the medical part were added on top once the calendar part was completed. This approach has eased the creation of highly modular datasets, and each module can be seen as being "independent" from the other. The interest in bringing modularity between the different sets of tables is to ease the understanding of modules and the integration of the tables in a different context. As the interactions between the two modules have been minimized, so is the amount of changes required to reuse a module in a different environment. The figure 5.2, and 5.3 illustrate the final database design. The global overview is in appendix A.2. On top of that, the tables 5.1 and 5.2 describe in more details the database tables.

Figure 5.2: Internal structure of the database, Calendar part (Primary keys are colored in yellow and Foreign keys in blue)

Figure 5.3: Internal structure of the database, Medical part (Primary keys are colored in yellow and Foreign keys in blue)

Table Name	Description
events	Models an appointment. The calendar module is organized around this table. The records, once loaded on the corresponding web module, are displayed inside a calendar.

types	Stores all the different event types (e.g competition, (para)medical session, video analysis, or training session with opposition). This table can be read from the modules but not written.[1]
users	Represents a user (e.g athlete) who is going to be involved in events and will answer post-training questionnaires from the mobile application. To respect the GDPR, personal fields (e.g., name, first name, and email address) are ciphered in the database. This table was part of the database design before the start of my thesis.
categories	Represents a team of users. The categories table is used to store the list of possible teams just as the types table stores the list of event types. This table can be read from the modules but not written.
event_athletes	Contains the mapping between event(s) and athletes(s). The `status` field is used to store the athlete's participation in the event. If the `status` is true, the athlete will be present, otherwise, he will be missing. The `message` field is supposed to be set only if the `status` is false and informs about the reason why the athlete is missing. The fields `activity` and `sensation` are meant to be used by the training module to store details about training sessions and questionnaires.
event_categories	Contains the mapping between event(s) and category/categories.
event_types	Contains the mapping between event(s) and type(s).
category_athletes	Contains the mapping between category/categories and athletes(s).

Table 5.1: Architecture of the database, tables of the Calendar module

Table Name	**Description**
injuries	Represents an athlete injury. Each injury is related to exactly one athlete, this is modeled by the field `id_user` inside the table. The `id_recurrence` field tells if the injury is a recurrence of a previous one. The injury referenced by this field must necessary involve the same athlete as the current injury. This allows to create links between injuries of the same athlete.

[1] A general note for all the "read-only" table. One could decide to remove those tables and list all the different possibilities directly on the web modules, but using a table allows for better readability, and flexibility.

diagnostics	Models a diagnostic made by a health specialist. A diagnostic is composed of several information modeled by the tables `diagnostic_body_parts`, `diagnostic_pathologies`, `diagnostic_types`, and `diagnostic_subtypes`. The information stored inside a diagnostic is organized in a particular hierarchy which could be modeled as a tree. A body part is associated with a set of pathologies, which in turn is associated with a set of types, which finally defines a list of possible subtypes. The `is_disease` field distinguishes if the diagnostic describes a wound or a disease. This field is reused in the `diagnostic_body_parts`, `diagnostic_pathologies`, `diagnostic_types`, and `diagnostic_subtypes` with the same purpose.
diagnostic_body_parts	Represents a body part. This table stores a list of predefined body parts and its content is meant to be read but not written.
diagnostic_pathologies	Represents a pathology. Each pathology is attached to a specific body part, this is modeled by the `id_part` field. This table stores a list of predefined pathologies and its content is meant to be read but not written.
diagnostic_types	Represents a diagnostic. Each diagnostic is attached to a specific pathology, this is modeled by the `id_pathology` field. This table stores a list of predefined diagnostics and its content is meant to be read but not written.
diagnostic_subtypes	Represents a sub-diagnostic. Each sub-diagnostic is attached to a specific diagnostic, this is modeled by the `id_type` field. This table stores a list of predefined sub-diagnostics and its content is meant to be read but not written.
circumstances	Models an injury circumstance. Each injury is attached to one circumstance.
authorizations	Represents a sporting activity authorization. The table lists the set of possible authorizations for an injured athlete regarding training, e.g biking, running, upper body strength training, or adapted technical-tactical training. Each authorization is associated with a level and depending on this level adding an authorization to an injured athlete may modify his status for training sessions as being missing or present. This table can be read from the modules but not written.
injury_events	Contains the mapping between injury/injuries and event(s). It is used to store the list of medical appointments realized during an injury. The medical module could work with any kind of event modelization as long as the events have a unique id. One would simply have to use this id in the current table instead of the currently defined `id_injury`.

injury_diagnostics	Contains the mapping between injury/injuries and diagnostic(s).
injury_authorizations	Contains the mapping between injury/injuries and authorization(s).

Table 5.2: Architecture of the database, tables of the Medical module

In this chapter, we have seen how the different project components interact together, and the organization of the database which was thought to be highly modular. In the following, we will review the `REST` server and `Flutter` modules individually to see their inner structures and how they work.

Chapter 6

REST server

From the previous chapters, we have seen the project specifications and its global organization. In this chapter, we will review the technical details of the implementation of the `REST` server.

6.1 Software architecture

6.1.1 Controller-Model pattern

The Controller-Model pattern is the architectural pattern used for the organization of the `REST` server components. This is greatly inspired by the Model-View-Controller Pattern. The figure 6.1 summarizes the logic behind this architecture.

- The **model** part represents the data. This is where the representation of the `SQL` tables can be found. Tables are defined using `Go` structure with `JSON` annotations. Along with the tables' definition, this is where all the logic to interact with tables is implemented. As stated in the section 5.1 (`REST` Server paragraph), all the interactions with the `PostgreSQL` database are performed by `GORM`.
- The **controller** part contains the high-level procedures to perform when a request is received (in path parameter parsing, `JSON` decoding, `UUID` checking ...) before dispatching the remaining tasks to the corresponding model.

Figure 6.1: MC pattern

6.1.2 Files organization

Besides the Controller-Model architectural pattern, files are organized in different folders. This structure aims, as most of this thesis work, at providing modularity and flexibility. The organizational system classifies services based on the similarities between the tasks performed. The table 6.1 lists the folders that come with the controller and model ones.

Package Name	Description
auth	Contains the authentication components of the `REST` server. See section 6.3 for more details
docs	The content of this folder is automatically generated by `Swaggo`. It contains the documentation of the `REST API`. See section 6.4 for further information.
res	Contains sample data used to populated the `DB` in debug mode. On top of that, they are some `Python` scripts used to translate formatted excel files in the corresponding requests to populate the database.
routes	Contains the list of routes exposed by the server.
swaggo	Contains additional structures used for the `REST API` documentation
tests	Contains unit, integration, and performance tests.
utils	Contains utility functions often used in the project (e.g `UUID` checking).

Table 6.1: Files organization of the Go server

6.2 Performances

When someone asked Bruce Lindsay about the 3 most important things in the database world his answer was: "performance, performance, and performance"[32]. This shows the importance to attach to efficiency. Efficiency shouldn't be considered a side task but has to be actively taken into account when designing a database.

6.2.1 Indexes

An index is an auxiliary data structure that is intended to help to find rids[1] of records that meet a selection condition [33]. It usually consists of a B+tree with high fanout. Indexes are used to improve the efficiency of the search algorithm used to find records inside a table. But everything comes with a cost, and indexes are no exception to the rule. Using indexes involves additional costs linked with the management of the indexes' data structures. Thus to maximize efficiency, one should define indexes carefully[2], try to minimize the number of indexes per table and prefer defining indexes on tables that are not frequently modified.

As stated in the last paragraph, indexes help to find records that match a given condition. Said otherwise, an index target a specific field, or set of fields and is usually designed to speed up one or a few queries. Some help for single record fetching, whereas others, will ease range selections on a specific condition. The table A.1, in appendices, describes the list of indexes defined in the database and their benefits. Note that all the `*_pkey` indexes are defined by `GORM`. Those indexes target the primary keys of the tables

[1] Record IDs
[2] The more selective an index the more efficient.

and enforce the uniqueness of the primary keys. Manually added indexes are in bold in the table.

6.2.2 Analysis of indexes efficiency

According to theory, indexes improve performances by providing a logarithmic search algorithm. The higher the fanout, the higher the logarithm base. But what do we have in practice ?

The figure 6.2 illustrates the evolution of the execution time of a query that updates a specific event in the `event` table according to the number of events stored. This query uses the index `events_pkey` to find the record to update. The setup I established to measure the execution time is simple. I started by filling my database with a predefined number of randomized events. Then, I used a pseudo-random number generator to select one event among all the ones created. Finally thanks to a timer provided by the `Go` testing package, I measured the time elapsed between the moment the request is launched up to the moment the response is received. I repeated the two last steps 100 times to get a representative average value. The execution time was measured for tables of sizes 10, 100, 1000, and 10 000. I selected values increasing by a factor of 10 to illustrate the fact that, thanks to the logarithmic algorithm provided by the index, the time to find the record remains almost unaffected even if the table size greatly increases. The graph confirms what the theory says. Indeed, the execution time seems to follow a logarithmic curve with respect to the table size[3]. This comforts us with the idea that our queries will still be efficient when the database will grow.

To illustrate more clearly the improvements bring by an index, the figure 6.3 compares the execution time of a query that fetches all the athletes of a specific event with and without an index. Again the time was measured for tables of 10, 100, 1000, and 10 000 records for the same reason as previously. The methodology used to perform the test is also equivalent. The result here is even more explicit. With an index, the execution time, as in the previous case, follows a logarithmic curve, whereas without an index the query execution time increases at a much faster rate.

6.3 Authentication and Routing

The server uses `JWT` tokens as authentication mechanism. `JSON Web Token` (`JWT`) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a `JSON` object. This information can be verified and trusted because it is digitally signed [34]. Once a user has successfully logged in he receives a dedicated token. Subsequent user requests include the token allowing him to access private routes, services, and resources according to the permission level of the token. Currently, each request includes a "bearer authentication" in the header. The token contains,

- An ID, which is the id of the user/member connected.
- An integer representing the role of the connected user/member. Roles allow for a finer distinction among users and members. This could for instance allow to

[3] I could go on with larger tables than this but my computer already has some trouble in handling 10 000 tuples table so I decided to stop there

Figure 6.2: Result of the performance test on the update events route

Figure 6.3: Result of the performance test on the get event athletes route

represent a hierarchy of users with different permissions at each level.

- A session `ID` representing the current user's session. This is used by the mobile application and goes outside the scope of this thesis.

To manage routing, the database relies on a Github package `Gorilla/Mux`[35]. It implements a request router and dispatcher for matching incoming requests to their respective handler. Incoming requests are matched to a list of registered routes and the package is responsible for calling a handler[4] for the matched rule.

[4]The handler is the corresponding controller method.

6.4 Swagger for documentation

As always in programming, documentation is essential to ease the understanding and maintenance of software. The `REST` server uses `swaggo`[36], a package based on the `Swagger` tool[37], to provide complete and consistent documentation. `Swaggo` allows to convert `Go` annotations (following a specific and predefined scheme) to `Swagger` documentation. All the methods exposed in the `API` (methods in the `controller` folder) are described using the `swaggo` syntax. This allows visualizing in a user-friendly way the whole `API` documentation in a web browser. On top of that, it is possible to extract from the swagger documentation a `JSON` collection that can be used in `Postman` (see section 6.5) to automatically build a collection that test every entry point of the `API`.

6.5 Postman as testing platform

`Postman`[8] offers a user-friendly way to interact with an `API` without having to write some code. This functionality appears to be useful to test an `API`'s methods very quickly. To do so `Postman` allows to define *collections* and *environments*. A *collection* is a set of `HTTP(S)` requests, each targeting a specific `URL`, and having a particular header and body. The *environment* is a list of variables (usually IDs) that can be reused by different requests. The *environment* usually contains IDs of database resources that are accessed by the methods of a specific *collection*.

In the scope of this thesis, on top of the `Swagger` documentation, we also build a `Postman` *collection* and *environment* to test and understand the `API`.

> In this chapter, we have seen how the `backend` part of the project is organized, how indexes were used to improve the efficiency of the requests, the routing and authentication mechanisms, how the `API` documentation was generated, and we concluded with a technique to easily test the `API` routes.
>
> In the next chapter, we will review the other side of the project, the `frontend`.

Chapter 7

Flutter modules

In the last chapter, we discussed the organization of the backend part of the *ATHLETin* solution. This chapter describes how the frontend part of the project is designed. It starts with its design pattern, the interactions with the `REST API`, and the file organization. Then it reviews the different techniques used to make sure that the application performances are satisfactory. Finally, the chapter ends with two sections that describe the functionalities of the modules.

7.1 MVVM Architectural Pattern

As applications are modified, and grow in size and scope, complex maintenance issues can arise [38]. Most of the time, the issues exist due to a blurred distinction between the `UI` display and the business logic that controls it. This simple organizational problem often results in difficulties to understand the code, apply modifications, or unit test the application logic.

The `MVVM` pattern is a well-known design pattern in software engineering. It helps to cleanly separate the business and presentation logic of an application from its user interface (`UI`) [38]. The figure 7.1 shows the interactions that exist between the three core components of the `MVVM` pattern: the model, the view, and the view model. This section will describe them.

First, note that this pattern has many variations and can be interpreted in different ways. As far as this thesis is concerned here is the way each component interacts with the others,

- The **View**, is the component that describes the structure, layout, and appearance of what the user sees on screen [38]. For `Flutter`, a view is composed of widgets only and the structure it defines can be easily modeled by a widget tree. Ideally, it should be independent of the context and could be reused anywhere. This property is not fully enforced in our implementation of the pattern. Rather, we defined one view per page and built it from widgets that may be reusable.

- The **View Model** implements properties and commands to which the view can bind, and react to view notification of any state change by updating its model and notifying the view to rebuild. The properties and commands that the view model provides define the functionality to be offered by the `UI`. However, the view determines how that functionality is to be displayed [38]. In `Flutter`, this can be

Figure 7.1: MVVM pattern

achieved through `ChangeNotifier` (see section 3.2.5.2). Each view is attached to a specific `ChangeNotifier` and listens to its notifications. But, if we look at a lower level, most of the widgets inside a view have their own `ChangeNotifier`. This is done for efficiency reasons. Using a Model View for each widget allows to have fine-grained control of the elements displayed on screen, and to selectively rebuild one element when it is modified instead of rebuilding the whole screen.

- The **Model** classes are non-visual classes that encapsulate the app's data. In our implementation, most of the models define database objects. Therefore, a View Model usually encapsulates several Models plus some variables to control the view behavior.

To conclude this section, I would like to highlight the importance of putting in place such a pattern in a project. More than organizing files, using a pattern have some true benefits. In the context of this thesis, this was especially helpful to settle a clean state management technique as well as to deal with application optimization.

7.2 Interaction with REST server

More than providing an ergonomic `UI`, the different modules must also interact with the database to save information across multiple users' sessions. Interactions with the database are implemented by a set of *Services*, which are `Dart` objects made available everywhere in the project thanks to the `GetIt` method (see section 3.2.5.3).

The core service is the `http_service`. It uses the `http` package [39] to implement the different types of requests that can be sent to the `REST API` (`GET, POST, PUT,` and `DELETE`). Each method receives as argument a route (required parameter) and a body (optional parameter). Then they use the `http` package methods to send requests with as header the bearer token received at login (see section 6.3). Once the request has been sent, we `await` for the response (see section 3.1.4), and upon reception, the body is decoded

into a `JSON` string and the method returns an `APIresponse` object (see figure 7.2). This class is used to encapsulate every `API` response in a generic format that can be reused whatever the content returned by the `REST` server is.

```
/// A model class for any Api response.
///
You, 6 months ago | 2 authors (Guillaume Lodrini and others)
class APIresponse<T> {
  /// Content of the response, can be anything.
  T? data;
  /// Says if an error has occured during request.
  bool error;
  /// Error message sended by the GO REST API in case of error.
  String errorMessage;
  /// Message sended by the GO REST API when no error occured.
  String message;


  /// Creates an [APIresponse].
  APIresponse({this.data, this.errorMessage = "", this.error = false, this.message = "",});
}
```

Figure 7.2: APIresponse object

On top of the `http_service`, I built several *high-level services* to interact with the different "sections" of the `REST` server. This included an `event_service` to manage all the requests targeting events, an `athlete_service` for the athletes, or an `injury_service` to deal with requests fetching or updating injuries. Those services reuse the functions implemented in the `http_service` and translate the `JSON` string stored in the data field of the received `APIresponse` into the corresponding Model object.

During interactions with the database, while the request is in transit, either the whole view or a part of it is replaced by a loading indicator. This allows to advertise the user that something is in progress (see figure 7.3). Otherwise, he might believe that the application is not responding. To notify that the request has ended, either the loading indicator is replaced by the corresponding view or if something went wrong during the request an alert dialog is displayed on screen (see figure 7.4).

Figure 7.3: Loading indicator while waiting for a request to finish

Figure 7.4: Alert dialog because of a bad authentication token

Finally, it is important to note that each module tries to minimize its amount of interactions with the database. For the calendar module, this is done by fetching all the events, and their details when the module is launched for the first time. In the following, when some data needs to be updated in the database, a request is sent to the database and the corresponding data is updated locally without re-fetching it. For the medical module, the idea is identical except that it loads injuries instead of events. This methodology increases the responsiveness of the module at the expense of a longer initial loading time, which in our opinion is an acceptable trade-off.

7.3 Files organization

For better readability, files are organized in different folders. The established structure follows the `MVVM` pattern plus some folders to store specific types of files. The files are first classified based on the tasks they fulfill, then according to the module part they target. As for the `REST` server, this aims at providing modularity and flexibility. The table 7.1 lists the folders used inside the two modules.

7.4 Performance tools

Every Flutter application uses at least 3 threads to run. The `UI` thread is the main one, this is where the `main.dart` runs and the widgets are built. Along with the `UI` thread, there is a raster thread that deals with rendering operations and an I/O thread. Those two threads are used by `Flutter` to perform some specific actions but as a programmer we don't interact with them. Performance issues usually result from heavy work in the UI thread. If the actions performed there take too long, or if the work is too large the application will probably jank[1]. Depending on the issue many solutions could exist. It goes from using `ListView.builder` which will only build visible items for very long lists, to using `Flutter compute()` method to create a new isolate[2] to perform multiple tasks at once without overloading the main thread.

Note that `Flutter` does not recommend trying to over-optimize applications. *Normal* applications should be efficient by default and `Flutter` already provides a good level of optimization. As they said, optimization should only take place if performance issues can be observed [42]. However, to ensure that the application is not doing more work than

[1]Jank refers to sluggishness in a user interface, usually caused by executing long tasks on the main thread, blocking rendering, or expending too much processor power on background processes [40].

[2]Instead of threads, all Dart code runs inside of isolates. Each isolate has its own memory heap, ensuring that none of the states in an isolate is accessible from any other isolate[41].

Package Name	Description
fonts	Contains the fonts used by the text displayed on screen.
images	Contains the images used by the different views.
constants	Contains immutable data the is reused across the project. This includes specific colors, and `REST` server constants.
extensions	Contains some widgets' extensions. A widget extension, as the name suggest, is something that may be added on top of a widget to grant it with additional functionalities (e.g MoveUpOnHover, or ShowCursorOnHover).
models	Contains the models of the `MVVM` pattern.
routing	This folder contains the routing part of the project. In this case routing denotes the activity of switching from views to views inside the module.
services	Contains the services used to interact with the `REST API`.
utils	Contains utility functions often used in the project (e.g `JSON` parsing).
view_models	Contains the view models of the `MVVM` pattern.
views	Contains the views of the `MVVM` pattern.
widgets	Contains the widgets composing the different views of the module.
test	Contains the unit tests testing the services and utils methods.

Table 7.1: Files organization of the Flutter modules

necessary (e.g useless rebuild). Here are some tools that can be used to check applications' performance.

7.4.1 Flutter profile mode

`Flutter`'s profile mode compiles and launches your application almost identically to release mode, but with just enough additional functionality to allow debugging performance problems. For example, profile mode provides tracing information to the profiling tools[43].

7.4.2 Dart DevTools

The `Dart DevTools` is a list of features that allows for profiling and debugging `Dart` and `Flutter` applications. As far as performance is concerned, it allows to,

- Diagnose `UI` jank performance issues
- `CPU` profiling
- Network profiling

However, this feature is currently not available for web applications in profile mode. To circumvent this problem, we decided to use tools available in debug mode. Those are explained in the following sections.

7.4.3 Tracking widget rebuilds

`Dart DevTools` offers the possibility to track widget rebuilds. This tool is very useful to know exactly which pieces on-screen are rebuilt when a specific action occurs. This allows us to see if some widgets are rebuilt more than necessary, and in converse, if widgets that should be rebuilt are. This tool allows to fine-tune the `notifyListeners()` in `ChangeNotifier` to the strict minimum.

Figure 7.5: Flutter widget rebuild tracking tool

7.4.4 Performance overlay

The performance overlay is not directly visible on top of the application for a web application but can still be accessed from the `Dart Devtools`. This tool shows over a time axis the time required by each frame to be rendered. A frame is considered to be janky if it takes more than 16ms to render it for 60fps[3] devices[44]. Thus, one can use its application as a regular user would do and then check the frame rendering time to see if the application is smooth or not.

Figure 7.6: Flutter frames rendering time tracking tool

[3] frame per seconds

7.5 Calendar module

This section describes the available features in the calendar module. Some screenshots of the different views are already provided in the following section, and the rest can be found in the appendices (section B.1).

7.5.1 Calendar page

The calendar page is the main page of the calendar module. This page allows the authenticated user to visualize the events of the athletes he manages. Upon opening of the page, the events are fetched from the database and then displayed inside the calendar, which proposes multiple displays: day, week, month, or list of events. Displayed events can be dragged&dropped and resized to modify respectively their starting time and duration without having to access their details. The month view has the particularity of being divided in two. The upper part displays the calendar while a window below the calendar displays all the events of the selected day. Finally, on clicking on a particular event, whatever the used view, it opens the corresponding event details pop-up (see section 7.5.2).

In addition to the calendar, the left side of the screen provides some filters to ease the search for specific events. The user can filter events according to their types, the categories of players supposed to be present at the event, or the list of players registered in the event. Each filter allows the selection of as many values as they are in the database.

Finally, the dark blue button with circular shape in the bottom right of the screen opens a window that allows to create a new event. This button appears on top of the calendar whatever the calendar view selected.

Figure 7.7: Calendar module agenda, overview (event day display)

7.5.2 Event details view

On clicking on a particular event, a user can access its details. This allows the user to see more information on the event and modify it if necessary. The first thing displayed is an overview of the event that includes all the information about the event except the athletes registered.

The Data section is where the user can see the athlete added to the event. On top of that, the user can also add new athletes thanks to the add button located in the bottom right corner, delete some athletes by clicking on the red trash, and modify the status of athletes by clicking on the pencil in the status column.

The Status part doesn't bring any new information but simply allows us to get an easier overview of who is currently participating in the event and who is not.

Finally, the modify tab allows modifying all the fields of the event, except the athletes. Note that it is not possible to directly modify athletes but athletes might still be modified if the selected categories are. When adding a new category to an event, all the players belonging to that category are automatically added. On the converse, when removing a category, the user has to possibility to select some athletes to keep and the others are removed.

Figure 7.8: Calendar module event details, overview

7.6 Medical module

This section lists the different features of the medical module. Some screenshots of the different views are already provided here, but more can be found in the appendices (section B.2).

7.6.1 Injuries table

This view corresponds to the home view of the injury module. On that page, we can see a table listing athletes' injuries. This table, like all the tables in this module, can be sorted according to any of its columns. The purpose of the table is not to describe in detail the injuries, but it is rather to expose some general information about the different injuries to ease the search for a specific one. Once the injury of interest has been found the user can click on the small blue eye to access its details (see section 7.6.2).

On left, the user can find a list of different filters that can be combined to locate either a specific injury, all the injuries of an athlete, finished injuries, ongoing injuries, injuries with specific diagnostics, or injuries with particular authorizations.

Finally, a button on the bottom of the table allows to create a new injury and add it to the table.

Figure 7.9: Medical module overview

7.6.2 Injury details

The injury details pop-up allows to get all the available information about an injury and to modify them if needed. This pop-up is divided in 7 sections each serving a specific purpose.

The first one, the overview section, as its name suggests, recaps all the information about the injury in a single view. However, nothing can be modified from this part.

The diagnostics section allows us to see all the different diagnostics made for the injury and to see who did them. This is useful to know if the injury has evolved and to compare the opinions of different specialists. Note that each diagnostic can be modified or deleted and it is also possible to add new diagnostics from there.

The circumstances part consists of a list of fields that have to be completed by the injured athlete to describe how he injured himself. Most of the fields propose a list of possible choices but if the *other* choice is made, then a text field will appear allowing the athlete to describe what happened.

The consultation tab lists all the medical appointments that have been done, and to be done by the athlete in the context of this particular injury. Doctors can also create new events or modify existing ones from there. This allows the health specialists to be able to interact with the calendar module without having to give them access to it.

The authorization section allows defining what kind of activity the injured athlete is allowed to perform depending on the evolution of its injury. The athlete's participation in events will be updated accordingly to the authorizations defined. To do so each authorization has been assigned a particular level and each type of event requires a specific authorization level to participate.

Finally the closed tab allows to modify the starting and ending date of the injury.

Figure 7.10: Medical module injury details, overview

7.6.3 Statistics

The last part of the medical module is the statistics page. On this page, the user can visualize statistics about the athletes' injuries. This page is made of 4 pie charts. Those charts allow us to visualize the distribution of the injured body parts, the injured sides, the pathologies diagnosed, and the circumstances. On top of the charts, the right side of the screen displays the average and median values of the number of unavailable days, missed sessions, and medical consultations with respect to the value selected on the corresponding chart.

Figure 7.11: Medical module statistics overview

This chapter, concludes the part about the implementation of the system. It has reviewed most of the components constituting the project and explained how the modules are concretely working.

In the next chapter, we will address the testing strategies that have been implemented, or that could be used to improve the project.

Chapter 8

Testing

This chapter presents testing tools and techniques to ensure the correctness of the implemented functionalities. The chapter is divided into two sections. The first one focuses on the REST server testing strategies, whereas the second reviews testing approaches for the frontend part of the project. This allows distinguishing the differences that exist between frontend and backend regarding testing.

8.1 REST server testing

Providing tests on the REST API was an important part of the creation process of the database. Indeed, the hard task is usually not to design a database but to make sure that its implementation conforms to what has been decided. This is especially true in a REST API where users rely on the specifications to interact with it. Therefore, it was really important to make sure that the system was performing and reacting just as it is supposed to. Tests provide a way to track down bugs and ensure the integrity of the implemented methods, which avoids the burdensome tasks of having to check manually that a function works as intended each time we slightly change it. However, as always with testing, just because you've counted all the trees doesn't mean you've seen the forest. Said otherwise, the tests provided try to be as complete as possible and to check every issue, but it is hard to make sure that all of them have been covered.

8.1.1 Go testify package

Testify[45] is a developer-friendly package with over 16,500 stars on GitHub, and significant community support of almost 200 contributors. It extends the capabilities of the `Go` standard test package in particular by allowing to perform assertion while still providing an easy to read and write syntax.

To create tests with testify, there is nothing fancy. The `REST` server contains a folder named tests (see section 6.1.2) that contains several `Go` files, each targeting a specific table, and importing the testify package. Then, once we imported the package, there is no need to extend a specific class or implement an interface. It suffices to create methods taking as argument a variable of type `assert.TestingT` and to use it to make some assertions on a variable's value, returned status code from the database, or compare some objects. Note that the files' name must end with the "*test*" keyword for `Go` to understand that they are test files.

Finally, to run a specific test, or whole test file the user can either run it from its IDE (GoLand by JetBrains[46] is well suited for Go) or using the go test -v <package> -run <TestFunction> command from the terminal to run a specific function from a specific package. Note that it is the developer's responsibility to make sure that the database state is restored after each test performed. If the created records are not properly deleted after the tests, it will alter the database state.

8.1.2 Unit tests

Unit tests aim at testing small pieces of code independently from each other to make sure that each piece is performing what it is supposed to do whatever the behavior of the others. This kind of test fits perfectly for REST API methods testing.

Ideally, a unit test should be written for each implemented method. However, in this project, we decided to write a unit test for each route exposed by the API. This approach allows testing the behavior of the API rather than its "implementation", in the sense that a test aims at enforcing that the tested route is compliant with its specifications whatever the way it is implemented or used. According to theory, those tests could be considered as integration tests as they involve several components (routing part of the REST server, controller, model, JSON decoding ...) but as each target a single component, we believe they can be viewed as unit tests. We could have written tests for each part individually, but we don't think it would have brought more value to the tests.

8.1.2.1 Testing template

Each test follows a particular testing approach, divided into several specific steps. This allows to make sure that most of the interesting cases have been tested, and that the database state is restored after the test. Here are the steps followed by each test:

1. List all the different objects that will be created for the test. The first object should always lead to a valid execution and the followings aim at testing invalid requests.
2. Declare the route and method tested (e.g path:="/api/events" and method:= "GET").
3. Start by trying to send the request with a wrong JWT and checked that the server returns an error.
4. Send the request with a correct JWT.
5. Check that the HTTP response status complies with what is described in the REST guidelines.
6. Check that the received JSON is well-formatted, and that the objects it describes are what they are supposed to be.
7. If the request is supposed to alter the database, check that the database has indeed been modified.
8. Once all the different cases have been tested, remove all the created records to restore the database state.

8.1.2.2 Test coverage

Test coverage is defined as a metric in Software Testing that measures the amount of testing performed by a set of tests. It gathers information about which parts of a program are executed when running the test suite to determine which branches of conditional statements have been taken[47]. Said otherwise, a covering test tracks which part of the source code are tested by a specific serie of tests and then computes the percentage of tested code. This allows getting an estimation of how far the tests are going. The higher the coverage, the more use cases have been tested.

`Go`, since release 1.2, provides a mechanism to compute the covering percentage and display test coverage results on an `HTML` page. The first step consists in running tests with a specific flag to signal that we are interested in getting coverage information. This is done using the following command `go test -v -coverpkg=./{YOUR PACKAGE(S)} -coverprofile cover.out {./TEST PACKAGE}`. Then, once all the tests have been run, one can visualize the result in his browser using: `go tool cover -html= cover.out -o cover.html | open cover.html`. A sample of what is provided as result is displayed in figure 8.1. In this picture, in green, we have the code that has been covered during the tests, and in red what has not been reached. For the models we get a covering of 78.66% and 76.36% for controllers (the details are provided in the appendix, see figures A.3 and A.4, yellow rectangles highlight the files created for this thesis). Those values are quite good since most of the uncovered code consists of database error checks which are not reached since no internal error occurs during the tests (see figure 8.1).

```
server_go/models/event.go (75.6%)   not tracked   not covered   covered

        injuryEvent := &InjuryEvent{}
        err := GetDB().First(injuryEvent, "id_event = ?", idEvent).Error
        if err != gorm.ErrRecordNotFound {
                return http.StatusForbidden, u.Message(false,
                        fmt.Sprintf("Event \"%s\"is used as consultation for injury \"%s
        }

        if err := GetDB().Model(&event).Association("Users").Clear().
                Error; err != nil {
                return http.StatusInternalServerError, u.Message(false,
                        fmt.Sprintf("Association(s) between %s and users "+
                                "cannot be removed: %s", idEvent, err.Error()))
        }

        if err := GetDB().Model(&event).Association("Categories").Clear().
                Error; err != nil {
                return http.StatusInternalServerError, u.Message(false,
                        fmt.Sprintf("Association(s) between %s and categories "+
                                "cannot be removed: %s", idEvent, err.Error()))
        }

        if err := GetDB().Model(&event).Association("Types").Clear().
                Error; err != nil {
                return http.StatusInternalServerError, u.Message(false,
                        fmt.Sprintf("Association(s) between %s and types "+
                                "cannot be removed: %s", idEvent, err.Error()))
        }

        if err := GetDB().Delete(Event{}, "id_event = ?", idEvent).
                Error; err != nil {
                return http.StatusInternalServerError, u.Message(false,
                        fmt.Sprintf("Event %s cannot be removed : %s"+
                                ". Contact an admin.", idEvent, err.Error()))
        }
```

Figure 8.1: Coverage result for the event model

8.1.3 Performance tests

Database performances have already been introduced in the section 6.2. This section gives details about how long it takes to get all the events of a specific member from the database. Those results will allow us to determine if pagination will be needed or not when the number of events will increase.

To get meaningful performance tests, it was first necessary to know how the platform was going to be used. To do so, I discussed with Julien Paulus to know his expectations concerning the future of the two modules.

In the short term, he hopes to reach around 100 rugby clubs. For the long term, his estimation is between 300 and 1000 clubs. On top of that, each club would provide 2 training sessions per day. Assuming 4.5 weeks per month this would lead, per month, to:

$$100 * 14 * 4.5 = 6300\, events$$

$$300 * 14 * 4.5 = 18900\, events$$

$$1000 * 14 * 4.5 = 63000\, events$$

Those numbers represent the total number of events stored per month, which leads to 63 events per month per club.

On top of his estimations, I decided to create another simulation that would lead to more events per month. This could happen in a situation where the events would target athletes individually instead of teams. My assumptions were the following. The number of clients would be between 10 and 100. As for the previous situation, the clients would be rugby clubs and thus teams of 15 athletes. We could imagine that each club would have 3 different teams. Then each player in the team could be involved in 4 different events per week. Assuming we have 4.5 weeks per month, on average, every month, we get for 10 clubs,

$$10 * 3 * 15 * 4 * 4.5 = 8100\, events$$

For 100 clubs, with the same assumptions, we get 81,000 events, and in both cases, we have 810 events per club per month. On top of those numbers, I also tested performances for 10 events per club, 100, and 400. This allows seeing if the performance's evolution is almost linear or if it is exponential according to the number of events stored.

The graphs 8.2 and 8.3 show the results of respectively Mr.Paulus estimations and our simulation[1]. The first thing we can note from both graphs is that the number of clubs (thus the total number of events stored in the database) doesn't seem to affect the execution time. What matters is the number of events fetched from the table. This can be explained by the index. Then, the second graph shows us that the evolution of the request duration is more or less linear regarding the number of events fetched. Finally, for 810 events, the request lasts around 10 seconds. From this value, we can deduce that if we have to fetch all the events of a club for one year, this would take around 2 minutes.

From all those observations, we can first conclude that the total number of events stored won't lead to performance issues. Second, in a situation where the number of

[1] There is an index `fast_member_events` on the events table to decrease the time necessary to find the events associated with a particular member.

Figure 8.2: Result of the performance test on the get member events route (Mr.Paulus estimations)

Figure 8.3: Result of the performance test on the get member events route (Our estimations)

events per month per client would increase too much[2], pagination might be required to avoid a too-long delay at the launch of the application. Concerning pagination, several approaches could be possible: load events by week, by month, load the next 100 events, and so on. This will have to be discussed when the project will be released. Nevertheless, if the module usage follows the estimations of Mr.Paulus, the current design will be efficient enough.

8.2 Flutter Web modules testing

`Flutter` web user interface testing is probably the weakness of this project. Indeed, the development phase of the two web modules consisted of repetitive phases of development and manual testing. This strategy has the benefit of allowing a quicker development, but at the end of the day, the software is sorely lacking automated tests. Even if usual use cases have been manually tested, those tests are not robust enough and some corner cases

[2]more than 500

may not have been identified. In the section 8.2.1 and 8.2.2, we describe testing strategies dedicated to flutter applications to address this flaw.

On the contrary, the interactions with the `REST API` have been tested. The tests provided verify that the `HTTP` methods used by the different services (see section 7.2) succeed in contacting the `REST API` and that the received response is syntactically correct. However, they do not proceed to any check on the response content, as this is supposed to be handled by the database tests.

8.2.1 Widget testing

A widget test (in other `UI` frameworks referred to as component test) tests a single widget. The goal of a widget test is to verify that the widget's UI looks and interacts as expected[48]. We could view this as unit testing for widgets. In a very large application, with lots of interactions between widgets, this kind of test might not be the most appropriate to verify that the platform is doing what it is supposed to. However, it still might be interesting to verify that some crucial parts of the applications are rendering well whatever the circumstances.

8.2.2 Integration testing

An integration test tests a complete app or a large part of an app. The goal of an integration test is to verify that all the widgets and services being tested work together as expected[48]. In other words, an integration test makes it possible to simulate user interactions and verify that the response from the application is the expected one (e.g when the user taps a button a network request is sent, which induces a loading indicator to be displayed and once it is finished, the view is restored and the table has been updated). On top of the testing capabilities, `Flutter` also provides a visualization tool that allows seeing in live the actions performed by the test. This is especially useful for debugging when a test fails. However, nothing comes for free, and integration tests are no exception to the rule. Indeed, they usually cost a lot in maintenance and are quite slow to execute.

> In this chapter, we have seen how the `REST` server has been tested, and we reviewed two kinds of testing strategies that could be put in place to enhance the test coverage of the `Flutter` web modules.
>
> As a last chapter, we will see how the project has been deployed using `Docker` and `Nginx`.

Chapter 9

Deployment

This chapter reviews the methodology used to deploy the different parts of the project. This goes from the encapsulation of each module with its dependencies in independent containers up to the Nginx configuration to route requests toward the right server. Note that the deployment strategy has been established by the professor Laurent MATHY, and his team. As result, my work regarding the deployment simply consisted in adding, on top of the current configuration, instructions to build the modules created in the scope of this thesis.

9.1 Docker to encapsulate the modules

The first deployment step was to create dedicated environments for each module. It has been done by encapsulating each module[1] in a container and installing in each the required dependencies. This way, the dependencies of each module can be managed independently of the others. The resulting separation between modules allows notably to run them on different dependencies' versions according to what they have been developed for. Finally, using `docker` to encapsulate the modules allows for quicker and easier deployment. Indeed, one can run the project inside a `VM` and the only thing he has to pay attention to is to install `docker` inside that `VM`. The rest is managed by the containers. Without `docker`, it would have been necessary to install the dependencies required by all the modules in the `VM`.

In the context of this thesis, I created three containers, one for the `REST` server, and one for each `Flutter` web application. On top of that, I reused a `Dockerfile` provided by the professor MATHY's team to build a container for the `Nginx` reverse proxy (see section 9.2). The figure 9.1 provides an overview of how the project files are organized.

- The `REST` server container is based on the official Golang image from Dockerhub[49] on which we copy the `Go` files, and install all the dependencies used by the `REST` server such as `GORM` or `testify`. On top of that, we also include the certificates used for `SSL` connection for the mobile application. Once this has been done, we build the executable of the server. Then, we create a new container from a lighter `Linux` image, `alpine`[50], and copy on it the executable, as well as the certificates, and some `SQL` scripts used to fill the database. Finally, all the environment variables used by the server are redefined inside the `alpine` image and the server entry-point is defined as the being the path to the executable.

[1] frontend or backend

- The `Flutter` web module containers are built on top of a `Debian` image. However, `Flutter` requires a series of tools to be available in the environment as installation requirements[51]. Thus, the first step consists in installing those tools. Once it is done, we download the `Flutter SDK` version of interest from the `Flutter` website[52] and copy all the files of the current directory to the container[2]. In the scope of this thesis, we use the `Flutter SDK` version 2.10.5. Once this have been done, using the `Flutter build` command, we ask `Flutter` to produce `JavaScript` files using its `dart2js` compiler. Those files are later copied inside an `Nginx` image which will be the final container and is referenced by the reverse proxy to make the module accessible by users from their browser (see section 9.2).

- The reverse proxy container is based on an `Nginx` image and on which we add our own configuration file.

Figure 9.1: Organization of the project files

9.2 Nginx as a reverse proxy

Once all the modules had been encapsulated, it was necessary to design a web server to make them accessible from the outside world. This has been done with the `Nginx` solution. In a few words, `Nginx` is an event-driven, and non-blocking asynchronous web server released through an open-source software[53]. Unlike `Apache`, one of its major concurrent, which is process-driven, `Nginx` only required a few processes and distribute the load among them. It works by creating worker processes to handle incoming requests, and then managing the events they spawn by adding them to a continuous loop.

In the context of this thesis, Nginx has been configured as a reverse proxy to distribute users across the different modules depending on the URI they send. To do so, all the modules have been assigned to the same domain but each one has a specific port. This way `Nginx` can easily distribute clients on the right server according to the targeted port. The figure 9.2 shows what has been explained here.

[2] `Dockerfile` is located inside a folder holding all the `Flutter` code required to start the module

Figure 9.2: High level overview of the deployment case. Nginx used as a reverse proxy to handle different routes

9.3 Starting scripts

The scripts described in this section are not mandatory but they ease the deployment of the solution by aggregating all the operations to perform to set everything up. They are 3 scripts to use to start the `Flutter` web modules and the `REST` server, they are described here below,

- The `build.sh` script contains the instructions to build all the docker containers. This script must be started first unless the docker containers have already been built.
- The `start.sh` script contains the instructions to start the previously build containers. This script can be used with the argument *resetdb* to reset the database content and fill it with predefined data.
- The `stop.sh` script contains instructions to stop and remove all the docker containers previously started.

> This concludes the implementation part. Through this part, we have seen the requirements of the project, how it was organized globally, the implementation details of each component, the different testing strategies, and finally how the project was deployed. The next chapter will conclude this thesis.

Part IV

Conclusion

Chapter 10

Conclusion

Throughout this master thesis, we describe how we designed and implemented a database, a `REST` server, and two web modules for the *ATHLETin* company. One of the web modules allows the management of events from a calendar. The other aims at providing health specialists with an interface from which they can manage athletes' injuries. This thesis expands a project already started by the professor Laurent MATHY and his team. This has motivated the choice of developing modular components to ease future integration.

Before diving in project implementation, it was necessary to analyze the current system specifications, notably the technologies used and the architectural choices. For the `backend`, the existing system was composed of a `PostgreSQL` database with a `REST` server in `Go`. The latter interacts with the database using an `ORM` dedicated to `Go`, `GORM`. For the `frontend` part, the choice fell on the `Dart` language and the `Flutter` framework. `Flutter` is a cross-platform `SDK` that allows to write once and deploy everywhere. Property that might be useful later on if the professor MATHY wants to reuse web module parts into a mobile application.

The development of the solution happened in two phases. First, we designed and implemented the calendar module part. Then we moved on with the medical module. This approach helped us to produce modular components.

The primary concern with this thesis was to provide both great performances and user-friendly interactions, and it has been challenging. To improve the efficiency of the `REST` server, we have enhanced the database's tables with indexes. Those indexes help to preserve acceptable requests' duration when the database grows up. Concerning the web, performance issues are usually due to unnecessary page rebuilding. To overpass this problem, we used the provider state management pattern. The latter allows selective rebuild of components depending on the user's interactions and thus prevents rebuilding parts that have not been modified.

Once the modules were completed, we created some tests to verify automatically what we did manually throughout the development phase. On top of that, we also designed some performance tests to ensure that the efficiency improvements we made were indeed rewarding. Finally, we created several `docker` containers for the different project parts. This way, one can quickly deploy the modules in any environment as soon as it contains the `docker` technology.

The goal of this thesis was to create a solution for the *ATHLETin* problems. After several designing and implementation phases, the result fits well with the client's requirements. The calendar module provides an ergonomic way to visualize and manage supervised athletes' events. The medical solution successfully digitalizes the shuttle sheets to provide a better follow-up of injuries. As far as the backend is concerned, the tables have been designed in a modular way to ease the integration within the professor MATHY architecture. Concerning efficiency, it was not leftover, and several strategies have been established to ensure stable performances. Finally, the solution comes with documentation and deploying scripts to ease the reuse in the future.

Chapter 11

Future work

To close this thesis, we list here below possible improvements and new features that could be added to enrich the current system capabilities.

11.1 Add pagination

As we have seen in the section 8.1.3 the system has some limitations in terms of loading capabilities. Adding pagination to limit the number of requested events or injuries at the launch of the modules will most probably tackle this issue without restricting the possible use cases of the different platforms.

11.2 Improve responsiveness

The application has only a few responsiveness capabilities, as it was not meant to be used on other devices than computers. Some `Flutter` widgets already adapt their view once the screen size change, but improvements could be made to make the platform usable on other devices.

11.3 Upgrade it to Flutter 3.0

`Flutter` 3.0 has been released on the 11th of May. This release provides new features such as compatibility with macOS and Linux desktop apps. Making the required modifications to deploy the application as a desktop app on the three platforms, Windows, macOS, and Linux could again enhance the application capabilities.

11.4 Internationalizing the web modules

The modules have been developed for French speakers. Enlarging the number of supported languages could make the *ATHLETin* solution available to a greater number of users.

11.5 Improve the current role system

The current application role system doesn't allow for a lot of authorization tuning. Improving the role system could allow making the platforms more suitable for the need of a hierarchical organization.

11.6 Instant messaging in medical module

In the requirement section 4.1.2, we have said that *"A discussion part should allow doctors and managers to exchange messages in an instant messaging similar to Messenger or WhatsApp"*. This functionality has not been implemented yet, but as a suggestion for the future handler of this project, this could be done with web sockets.

Appendix A

Backend part

A.1 Database architecture

Figure A.1: Internal structure of the database (Primary keys are colored in yellow and Foreign keys in blue)

A.2 Full Database overview

Figure A.2: Whole database overview

A.3 Databases indexes

Index name	Table (fields)	Benefits
categories_pkey	categories (id_category)	Checks for the uniqueness of each category.
index_fast_team_athlete	category_athlete (id_category)	Speeds up the fetching of a team's athletes.
circumstance_pkey	circumstances (id_circumstance)	Checks for the uniqueness of each circumstance and speeds up the fetching of the circumstances of an injury.
diagnostic_pkey	diagnostics (id_diagnostic)	Checks for the uniqueness of each diagnostic and speeds up the fetching of a specific diagnostic.
diagnostic_body_parts_pkey	diagnostic_body_parts (id_part)	Checks for the uniqueness of each body part and speeds up the fetching of a specific body part.
diagnostic_pathologies_pkey	diagnostic_pathologies (id_pathology)	Checks for the uniqueness of each pathology and speeds up the fetching of a specific pathology.
diagnostic_types_pkey	diagnostic_types (id_type)	Checks for the uniqueness of each diagnostic type and speeds up the fetching of a specific diagnostic type.
diagnostic_subtypes_pkey	diagnostic_subtypes (id_subtype)	Checks for the uniqueness of each diagnostic subtype and speeds up the fetching of a specific diagnostic subtype.
events_pkey	events (id_event)	Checks for the uniqueness of each event and speeds up the fetching of a specific event.
event_athletes_pkey	event_athletes (id_event, id_user)	Checks for the uniqueness of each event athlete and speeds up the fetching of an athlete in an event.
fast_users_of_events	event_athletes (id_event)	Speeds up the fetching of all the athletes of an event.
event_categories_pkey	event_categories (id_event, id_category)	Checks for the uniqueness of each event category.

event_types_pkey	event_types (id_event, id_type)	Checks for the uniqueness of each event type.
injuries_pkey	injuries (id_injury)	Checks for the uniqueness of each injury and speeds up the fetching of a specific injury.
injury_authorizations_pkey	injury_authorizations (id_injury, id_authorization)	Checks for the uniqueness of injury authorization and speeds up the fetching of a specific injury authorization.
fast_auth_of_injury	injury_authorizations (id_injury)	Speeds up the fetching of all the authorizations of an injury.
injury_diagnostics_pkey	injury_diagnostics (id_injury, id_diagnostic)	Checks for the uniqueness of injury diagnostics and speeds up the fetching of a specific injury diagnostic.
fast_diag_of_injury	injury_diagnostics (id_injury)	Speeds up the fetching of all the diagnostics of an injury.
injury_events_pkey	injury_events (id_injury, id_event)	Checks for the uniqueness of injury events and speeds up the fetching of a specific injury event.
fast_events_of_injury	injury_events (id_injury)	Speeds up the fetching of all the events of an injury.
types_pkey	types (id_type)	Checks for the uniqueness of types events.
fast_member_events	events (creator)	Speeds up the fetching of all the events created by a specific member.

Table A.1: List of the indexes defined in the database

A.4 Test coverage results

Figure A.3: Covering percentage for models

Figure A.4: Covering percentage for controllers

Appendix B

Flutter web modules

B.1 Calendar module screenshots

Figure B.1: Calendar module agenda, overview (event list display)

Figure B.2: Calendar module agenda, overview (event month display)

Figure B.3: Calendar module agenda, multi-select filter

Figure B.4: Calendar module event details, data view

Figure B.5: Calendar module event details, status view

Figure B.6: Calendar module event details, modify event view

Figure B.7: Calendar module add a new event 1

Figure B.8: Calendar module add a new event 2

B.2 Medical module screenshots

Figure B.9: Medical module injury details, diagnostics

Figure B.10: Medical module injury details, add a new diagnostic (wound)

Figure B.11: Medical module injury details, add a new diagnostic (wound) 2

Figure B.12: Medical module injury details, add a new diagnostic (disease)

Figure B.13: Medical module modify a diagnostic

Figure B.14: Medical module injury details, circumstances

Figure B.15: Medical module injury details, consultations

Figure B.16: Medical module injury details, add a new consultation

Figure B.17: Medical module injury details, modify a consultation

Figure B.18: Medical module injury details, authorizations

Figure B.19: Medical module injury details, add new authorizations

Figure B.20: Medical module injury details, modify injury dates

Bibliography

[1] *The Go programming language*. URL: https://go.dev/.

[2] *Quotes about Go*. URL: http://go-lang.cat-v.org/quotes.

[3] Google. *Documentation - the go programming language*. URL: https://go.dev/doc/.

[4] *Go wikipedia*. URL: https://en.wikipedia.org/wiki/Go_(programming_language).

[5] Rob Pike. *Go at Google: Language Design in the Service of Software Engineering*. URL: https://talks.golang.org/2012/splash.article.

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

[7] RedHat. *What is a REST API?* URL: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[8] *Postman*. URL: https://www.postman.com/.

[9] Jacob Sharir. *How to use Postman for API Testing Automation*. URL: https://www.blazemeter.com/blog/how-use-postman-manage-and-execute-your-apis.

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

[11] Gilad Bracha. *The Dart Programming Language*. Addison Wesley, 2014. ISBN: 9780321927705.

[12] Google. *Dart overview*. URL: https://dart.dev/overview#web-platform.

[13] Chris Buckett. *Dart in Action*. Manning Publications, 2013.

[14] Google. *dartdevc: The Dart dev compiler*. URL: https://dart.dev/tools/dartdevc.

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

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

[17] *Flutter framework*. URL: https://flutter.dev/.

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

[19] Eric Windmill. *Flutter in Action*. Manning Publications, 2019.

[20] Chris Sells. *What's New in Flutter 2*. URL: https://medium.com/flutter/whats-new-in-flutter-2-0-fe8e95ecc65.

[21] Temitope Oyedele. *A complete guide to Flutter architecture*. URL: https://blog.logrocket.com/complete-guide-flutter-architecture/.

[22] Google. *Widget catalog*. URL: https://docs.flutter.dev/development/ui/widgets.

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

[24] Google. *Flutter State<T extends StatefulWidget> class*. URL: https://api.flutter.dev/flutter/widgets/State-class.html#:~:text=State%5C%20is%5C%20information%5C%20that%5C%20(1,such%5C%20state%5C%20changes%5C%2C%5C%20using%5C%20State..

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

[26] Google. *Pubdev*. URL: https://pub.dev/packages/provider.

[27] Google. *Simple app state management*. URL: https://docs.flutter.dev/development/data-and-backend/state-mgmt/simple.

[28] Thomas Burkhart. *GetIt package*. URL: https://pub.dev/packages/get_it.

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

[30] Google. *Using packages*. URL: https://docs.flutter.dev/development/packages-and-plugins/using-packages.

[31] Microsoft. *Database design basics*. URL: https://support.microsoft.com/en-us/office/database-design-basics-eb2159cf-1e30-401a-8084-bd4f9c9ca1f5.

[32] Marianne Winslett. *Bruce Lindsay Speaks Out*. URL: https://sigmod.org/publications/interviews/pdf/p71-column-winslet.pdf.

[33] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems 2nd Edition*. McGraw Hill, 2000.

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

[35] *Gorilla mux package*. URL: https://github.com/gorilla/mux.

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

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

[38] davidbritch et al. *The Model-View-ViewModel Pattern*. URL: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm.

[39] dart.dev. *Http package*. URL: https://pub.dev/packages/http.

[40] *Jank*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Jank.

[41] Dart. *Concurrency in Dart*. URL: https://dart.dev/guides/language/concurrency#how-isolates-work.

[42] Filip Hracek. *Flutter performances*. URL: https://www.youtube.com/watch?v=vVg9It7cOfY.

[43] Flutter. *Flutter performances profiling*. URL: https://docs.flutter.dev/perf/ui-performance#launch-devtools.

[44] Google. *Flutter performance best practices*. URL: https://docs.flutter.dev/perf/best-practices.

[45] *Testify package - Thou Shalt Write Tests*. URL: https://github.com/stretchr/testify.

[46] JetBrains. *GoLand*. URL: https://www.jetbrains.com/go/.

[47] Thomas Hamilton. *Test Coverage in Software Testing*. URL: https://www.guru99.com/test-coverage-in-software-testing.html.

[48] Flutter. *Testing Flutter apps*. URL: https://docs.flutter.dev/testing.

[49] *Goland container*. URL: https://hub.docker.com/_/golang/.

[50] *Alpine container*. URL: https://hub.docker.com/_/alpine.

[51] Google. *Install Flutter on Linux*. URL: https://docs.flutter.dev/get-started/install/linux.

[52] Google. *Flutter SDK*. URL: https://docs.flutter.dev/development/tools/sdk/releases.

[53] Tom Rankin. *Nginx vs Apache: Which Is the Best Web Server?* URL: https://themeisle.com/blog/nginx-vs-apache/.