Master thesis : Mobile application and software development at Foot 24-7

Auteur : Meyers, Cédric
Promoteur(s) : Donnet, Benoît; 15861
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en informatique, à finalité spécialisée en "management"
Année académique : 2021-2022
URI/URL : https://github.com/ErwanThebaultDeuse/foot247; http://hdl.handle.net/2268.2/14507

University of Liège - Faculty of Applied Sciences

Civil engineering - Master in computer science

Mobile application and software development at Foot 24-7

Master thesis carried out in fulfilment of the requirements for obtaining the Master's degree in Computer Science and Engineering by Cédric Meyers

Author: Cédric Meyers

Supervisor: Pr. Benoit Donnet

Academic year 2021-2022

Acknowledgements

Even though a Master thesis is mainly about personal effort, exchanging with others is essential and can prove to be very valuable in order to achieve a qualitative work.

Therefore, I would first like to thank my industrial supervisor, Rayan Kassir, for giving me the opportunity to work on this project and for our many instructive exchanges.

I would also like to extend my thanks to Erwan Thebault and the Deuse company in general for the training they offered me but also for their support during the course of this project.

Of course, I would also like to express my sincere gratitude to my academic supervisor, Mr Benoit Donnet, for his insightful feedback and valuable guidance.

Finally, I also wish to thank my family and friends for their support, advice and for proof-reading this thesis.

Abstract

This Master thesis addresses the development and improvement of the Foot 24-7 mobile application, an application which connects amateur footballers together to facilitate the practice and organisation of football games.

Nowadays, mobile applications have become an integral part of our daily lives. Statistically speaking, 4 out of 5 people in the world own a smartphone. On top of that, the average daily time people spend on their smartphone is over 3 hours and most of this time is spent using an application. To appraise the importance of mobile applications in today's industry, we can mention that the number of mobile application downloads worldwide reached 230 billion in 2021 and this number is only increasing over time. It is therefore quite clear that the mobile application industry is thriving and will continue to do so in the coming years.

On the other hand, there is no denying the importance of football in our modern societies. As a matter of fact, football is the most played and followed sport in the world. As a result, almost everyone wants to play football. However, for many football enthusiasts, the practice of this sport is sometimes very difficult: lack of partners, difficulties in finding an available pitch or venue, last-minute withdrawals, etc.

In this context, the Foot 24-7 mobile application provides a solution to all of these shortcomings that any football fan knows only too well. In a continuous effort to enhance the user experience on their mobile application, Foot 24-7 is constantly seeking to improve their application in order to meet their users' expectations. Consequently, the purpose of this thesis was to research, design and implement the most relevant improvements to the Foot 24-7 mobile application.

This thesis addressed the following development tasks using well-known application development technologies such as the `Flutter` framework, the `Django` framework and the `Django REST` framework: debugging of the original application, development of player and team rating systems, addition of the possibility to create team games, access to a user's personal teams, redesign of the whole invitation system, addition of the possibility to edit a created game, improvement of the bottom navigation bar with notifications as well as the development of a brand new access dedicated to referees.

The work carried out during this Master thesis allowed to improve the user experience on the Foot 24-7 mobile application and will hopefully help it to reach an increasingly conquered public.

Contents

1 Introduction

This Master thesis addresses the development and improvement of the Foot 24-7 mobile application, an application which connects amateur footballers together to facilitate the practice and organisation of football games.

In this report, we will first present the context in which this Master thesis took place. This will allow us to understand the Foot 24-7 project as well as to explain and clearly identify the pre-existing basis on which this thesis rests. Next, we will describe the problem as presented by Foot 24-7 and the way I decided to approach it. The main part of this report will focus on describing the solution to the stated problem. Finally, we will conclude by reviewing the status of the project following the work carried out during this thesis, which will allow us to discuss perspectives for future work on the Foot 24-7 mobile application.

The entire project implementation related to this thesis is available on the following GitHub repository: `https://github.com/ErwanThebaultDeuse/foot247`.

2 Context

2.1 Description of the company

As this project was conducted in collaboration with the company Foot 24-7 [28], we will first describe this company and its project.

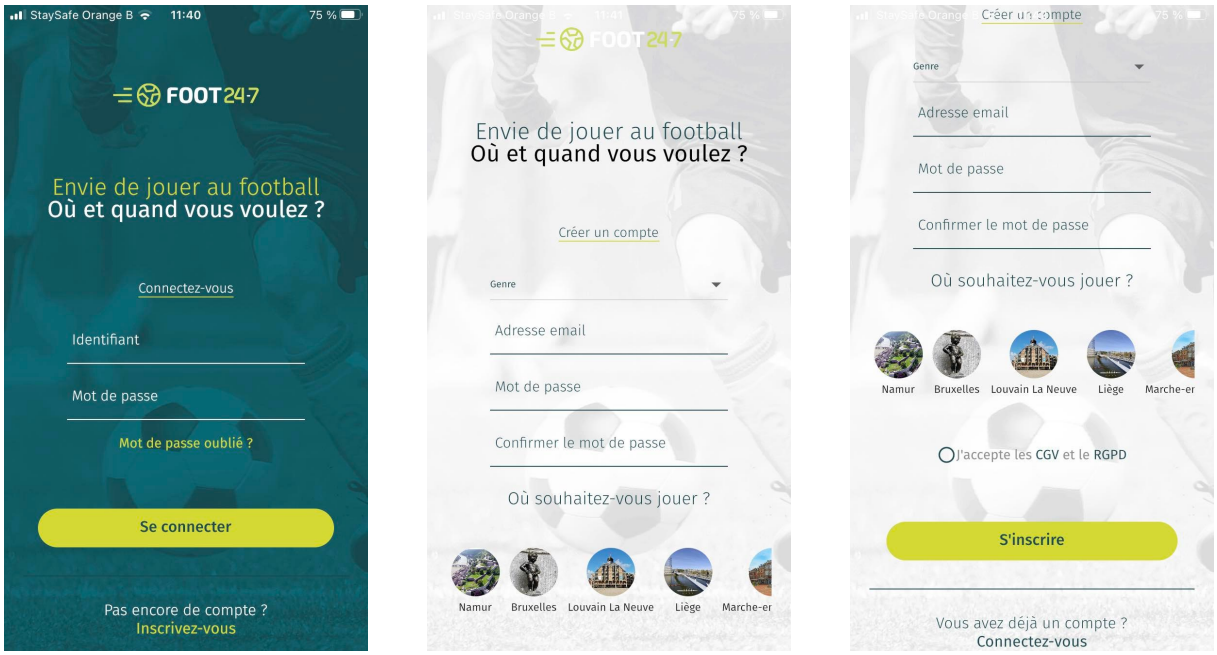
Foot 24-7 is a company founded by Mr Rayan Kassir in 2019. Mr Kassir is an avid football fan and, as such, loves playing football regularly, but when he arrived in Liège in 2014 to pursue his studies, he quickly discovered that **finding players and the required facilities to play football was particularly complex**, especially in the case of university tournaments. As he realised that many other people were experiencing the same difficulties (finding a venue or an available pitch, lack of players, last minute withdrawals, etc.), Mr Kassir decided to propose a solution allowing to remove these obstacles to the practice of the sport by **connecting amateur footballers**, even if they are isolated: Foot 24-7 was born.

On the Foot 24-7 mobile application, it is therefore possible to search for games, players (according to their level and position on the pitch), teams and even tournaments. It is also possible to create your own team and organise games by choosing the date, time and pitch. Since the mobile application is free, the user pays nothing. It is the venues, sports complexes and tournament organisers that pay a flat fee in exchange for several services: a digitalised offer, increased visibility as well as an online booking system. Indeed, in addition to its mobile application, Foot 24-7 has also developed an online platform that tournament organisers and pitch owners use for their organisation. Moreover, Foot 24-7 also has sponsors and partnerships with well-known companies such as Decathlon [15] and Adeps [9].

Foot 24-7 is now available in Belgium in Brussels, Liège, Louvain-la-Neuve, Namur and Marche-en-Famenne, but its scope should be extended in the future, notably to France.

2.2 Application presentation

In this section, we will describe and illustrate the initial mobile application in order to present its general functioning. This will allow us to **review** the **fundamental concepts** and notions used in the application as well as its **basic functionalities**.
The Foot 24-7 mobile application is available on both iOS and Google Play. Once the application is downloaded, the user is first invited to log in with an existing account if he already has one or to create a new account, as illustrated in the following Figure 1.

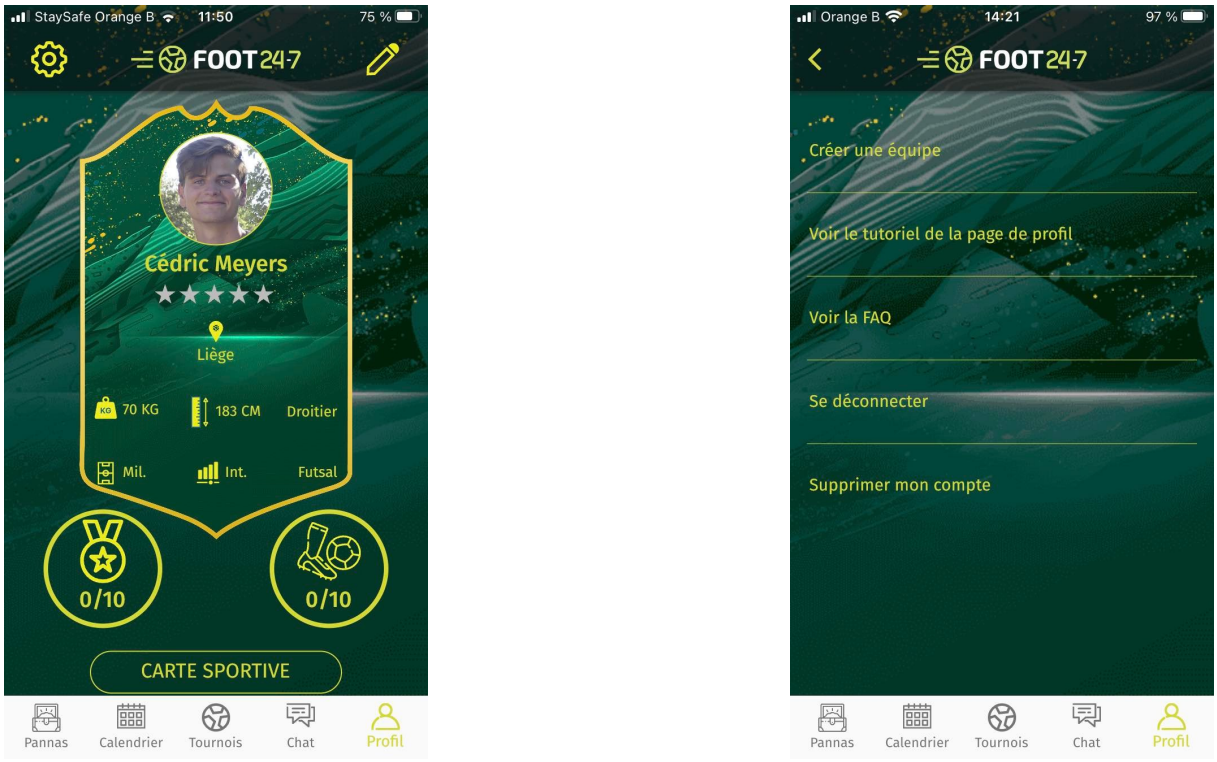


(a) User login page (b) Account creation page (c) Account creation page

Figure 1: User login and account creation pages

Once logged in, the user can then access the application and is redirected to his profile page. The mobile application is mainly divided into 5 tabs:

- The "Profil" tab allows the user to access and edit his personal data (name, first name, city, profile picture, level, position, etc.). In this tab, the user can also access the settings screen. Both the profile and settings screens are illustrated in the following Figure 2.



(a) Profile page

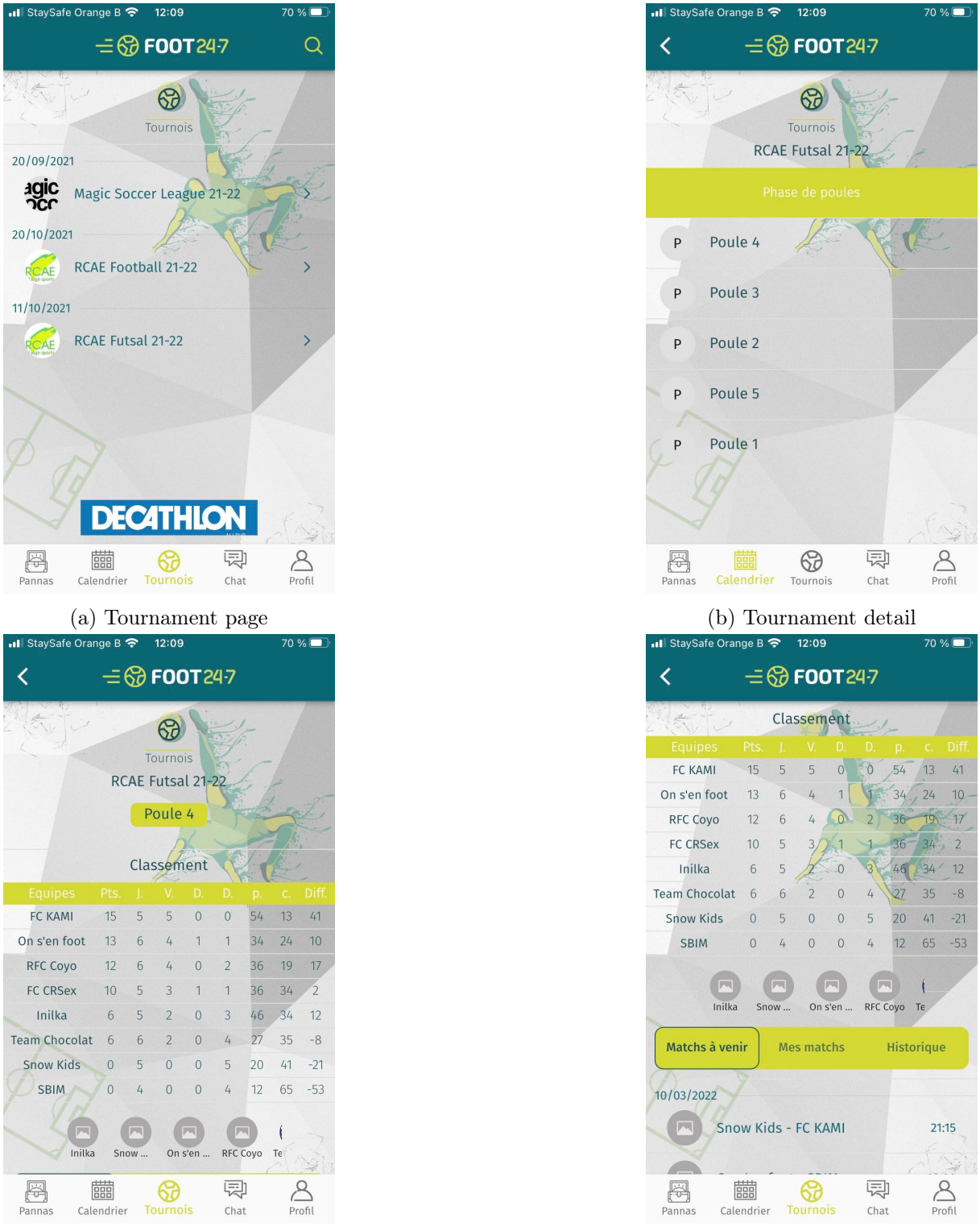
(b) Settings page

Figure 2: Profile and settings pages

- The "Chat" tab allows the user to access all the chats in which he participates. This includes of course chats between 2 users but also chats associated with a team or a particular game (see Figure 3).

Figure 3: Chat page

- The "Tournois" tab allows the user to view all the tournaments currently in progress. By clicking on a tournament, the user has access to the details of this tournament, *i.e.* the phase of the tournament (group phase or elimination phase), the participating teams, the ranking, the upcoming games as well as the history of the games. All these elements are depicted in the following Figure 4.



(a) Tournament page

(b) Tournament detail

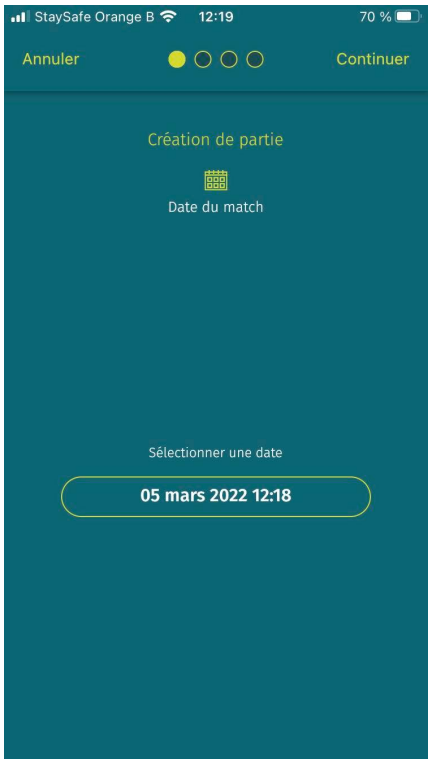
(c) Tournament group

(d) Tournament group

Figure 4: Tournament pages

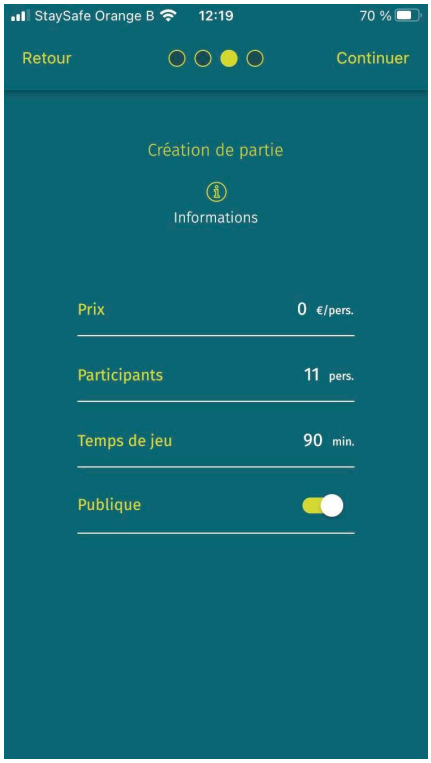
- The "Calendrier" tab allows the user to keep track of all games in which he participates, whether they are tournament games or unofficial games. In addition, this tab also offers the user the possibility to create his own game by specifying the date, place, number of participants, duration, etc. All these pages are presented in the following Figures 5 and 6.

Figure 5: Calendar page

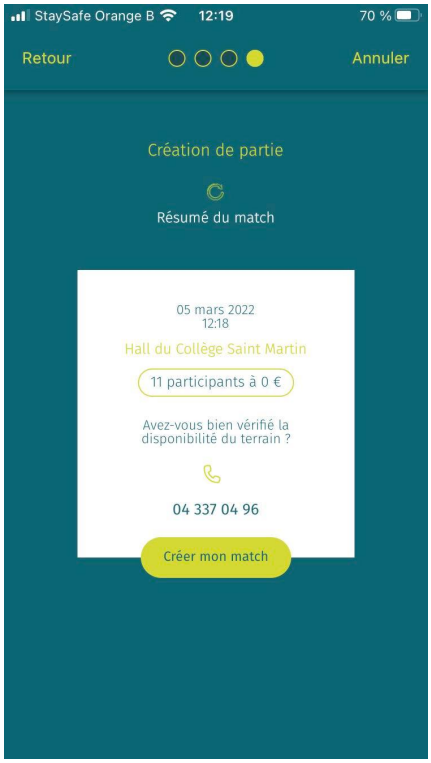


(a) First page

(b) Second page



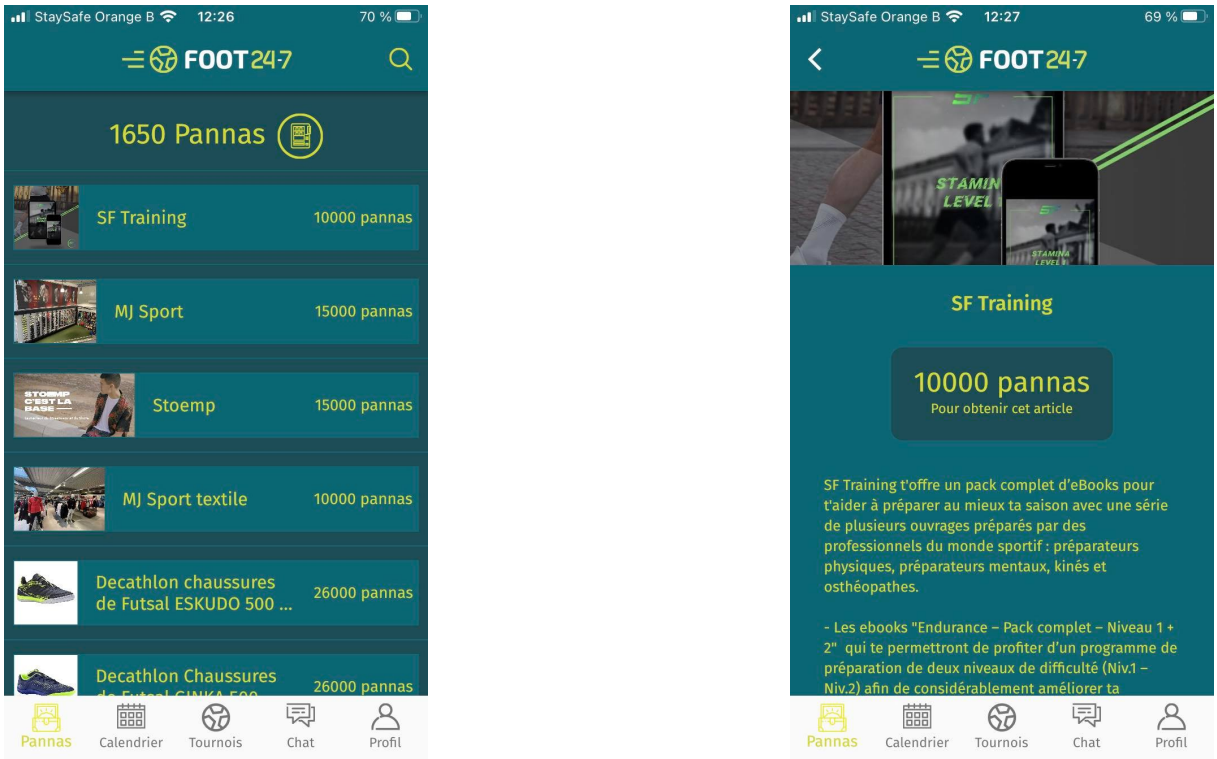
(c) Third page



(d) Fourth page

Figure 6: Game creation pages

- The "Pannas" tab allows the user to access a list of products that he can purchase using pannas. **Pannas are a kind of currency specific to the Foot 24-7 mobile application** that users can spend on various products offered by Foot 24-7 which are displayed in the "Pannas" tab. Users earn pannas during their experience on the application. Indeed, users can collect pannas for example when they complete their profile information, when they win a game, when they are elected most valuable player (MVP) of a game or when they share a game on their social networks. This system of earning pannas according to the actions performed on the application aims at stimulating users so as to maximize their use of the application and this strategy is actually called **gamification**. As a matter of fact, a gamification strategy is a process of taking an existing software and using gaming techniques to motivate user participation. The list and detail pages of the products that a user can purchase using the pannas he collects via the application are illustrated in the following Figure 7.



(a) Product list

(b) Product detail

Figure 7: Product list and detail pages

Besides these 5 main tabs, the user can also search for games, players or teams by pressing the search icon in the application bar. When pressing this button, the user is redirected to the search page illustrated in the following Figure 8.

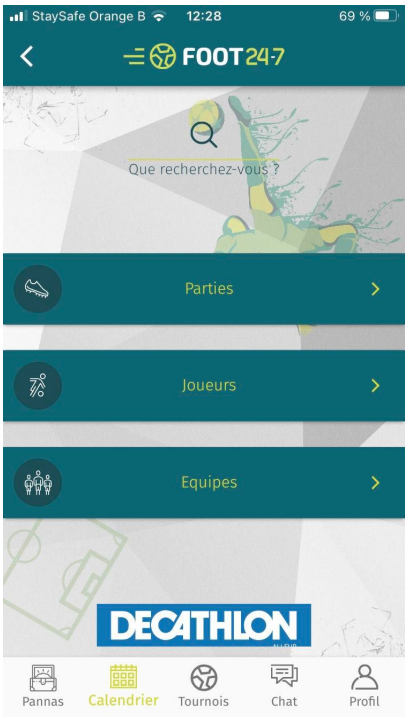


Figure 8: Search page

To conclude this section presenting the original application, we can summarise the set of features initially offered to a user of the Foot 24-7 mobile application in the following diagram 9. This diagram provides a high-level view of the original functionalities and serves as a benchmark for comparison with the high-level diagram presenting the set of functionalities available following the work carried out during this thesis, which can be seen in Figure 66.

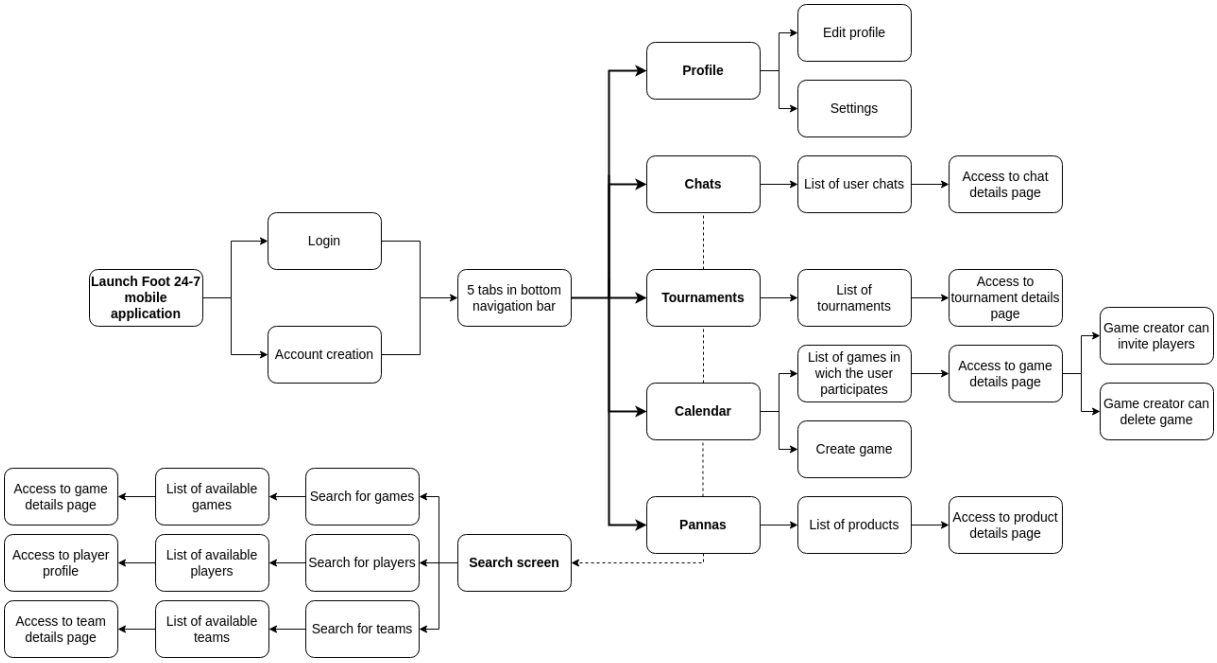


Figure 9: High-level diagram of original features

2.3 Application architecture and used technologies

The entire Foot 24-7 project relies on a `Docker` [22] [20] architecture allowing to easily replicate the production environment on any machine. As it happens, `Docker` is an open source platform which enables developers to package applications into isolated environments called containers. Containers are standard software units encapsulating code and all its dependencies to provide a portable runtime environment for an application. In other words, `Docker` containers are lightweight standalone executable software packages containing everything needed to run an application [21]. As a result, `Docker` allows to isolate an application from the local infrastructure. This enables to quickly and reliably develop, distribute and run an application on any computing environment.

As with most applications, the Foot 24-7 mobile application is built around a **frontend** and a **backend**.

In the case of Foot 24-7, the frontend is implemented using the `Flutter` framework [27]. `Flutter` is a popular frontend development framework developed by Google that enables developers to build beautiful user interfaces. `Flutter` is designed to ease the development of cross-platform applications. As a matter of fact, it allows to develop applications for Android, iOS, Linux, Mac, Windows, etc. from a single code base. To build these applications, the `Flutter` framework relies on the `Dart` programming language [14].

As far as the backend is concerned, it is implemented through the `Django` framework [19]. `Django` is a fully-featured Python [32] web framework that enables rapid development of secure and maintainable web applications. `Django` is based on the very popular *Model-View-Controller (MVC) pattern*, which is an architectural pattern that separates an application into three main logical components: the model, the view and the controller. These components are designed to handle the three main pieces of logic separately:

- The model is the functional core of the application. It represents the data, provides access to the data. It defines the processing and operations that can be applied to the data and thus exposes the application's functionalities.
- The view is the representation of the data in the model. It ensures consistency between the representation it gives and the state of the model (*i.e.* the context of the application). Hence, the view represents the outputs of the application.
- The controller represents the behaviour of the application in response to the user's actions. It provides the translation of the user's actions into actions on the model and provides the appropriate view following the user's actions and the model's reactions. The controller is therefore responsible for the behaviour and input management of the application.

Although based upon the MVC pattern, the architecture used by `Django` is actually slightly different as `Django` handles the controller part itself. Therefore, the pattern `Django` utilizes is called the *Model-View-Template (MVT) pattern*, where the view and the template in the MVT pattern make up the view in the MVC pattern.
In the context of the Foot 24-7 mobile application, since the frontend is implemented using `Flutter`, `Django` only serves as a backend through the definition of the models.

To transfer information between the `Flutter` frontend and the `Django` backend, the application uses the `Django REST` framework [18]. The `Django REST` framework (DRF) is a toolkit built on top of the `Django` web framework that allows to create `REST` interfaces. As such, the `Django REST` framework allows a **frontend** and a **backend** to **communicate with each other through a `REST` API**. `REST` APIs operate via requests and responses which means that such APIs provide a series of endpoints in the form of urls that a client can call in order to perform a request. As `REST` is actually built on top of existing HTTP methods, a `REST` API uses HTTP requests to perform standard database functions such as creating, reading, updating and deleting records. The format of the data sent back and forth in the body of a `REST` command is up to the implementer but, in the case of Foot 24-7, the serialization of data is implemented via `JSON` objects. On the one hand, this means that `Django` models get paired with DRF serializers that specify how to turn the content of a `Django` model into a `JSON` object that will be sent to the frontend (and inversely). On the other hand, it means that there also exist serializers in the frontend in order to transform `JSON` objects into `Dart` objects (and inversely). This **double serialization of data** is a central piece in the structure of the application and is fundamental to ensure communication between the `Flutter` frontend and the `Django` backend.

2.4 Code structure

As can be seen in the following Figure 10, the initial code structure consists of 5 main folders:

1. The `cron` folder
2. The `mobile_app` folder
3. The `nginx` folder
4. The `redis` folder
5. The `web` folder

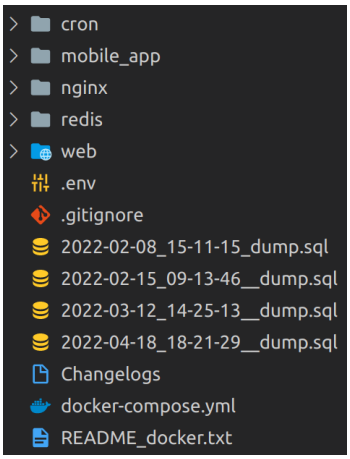


Figure 10: 5 main folders of the code structure

The 2 most important folders of this structure are the `mobile_app` folder and the `web` folder.

`mobile_app` folder
`mobile_app` is the folder containing the **frontend** of the application. Inside this folder, one can find the structure of a classic `Flutter` application, as shown in the Figure 11 below.

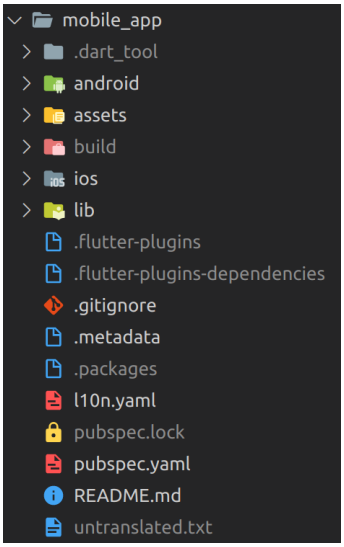


Figure 11: `mobile_app` folder

Hence, the source code of the Foot 24-7 mobile application can be found in the `mobile_app/lib` folder. In this folder, the source code is logically divided into several sub-folders, each of which concerns a particular component of the mobile application. The following Figure 12 illustrates the contents of the `lib` folder.

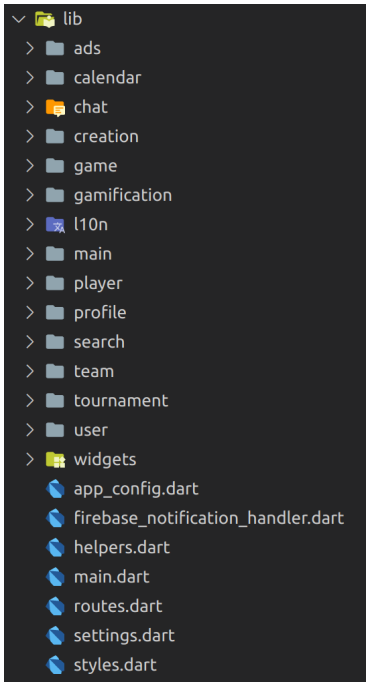


Figure 12: `lib` folder

Most of the directories in the `lib` folder contain the same 2 sub-directories: `api` and `screens`. The `screens` folders contain `Dart` files that correspond to the actual screens that can be found on the mobile application while the `api` folders contain the API between the frontend and the backend as well as data serializers (from `Dart` objects to `JSON` objects and inversely). An example of such a structure can be found in the following Figure 13.

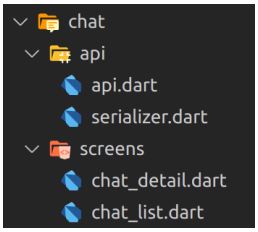


Figure 13: `api` and `screens` sub-folders

The entry point of the mobile application can be found in the `main.dart` file which is located in the `lib` folder. However, this file is mainly used to define the desired screen orientation and only creates the application by instantiating a `Routes` object which is defined in the `routes.dart` file (from the same folder). The `routes.dart` file is responsible for **managing the routing of the application**. Routing is a core concept of any mobile application allowing a user to navigate between different pages (or screens) of an application. It is thus one of the most basic things to implement in an application. In `Flutter`, navigating between different routes (*i.e.* pages) of an application is done with the use of the `Navigator` widget. The `Navigator` widget manages the different pages of an application in a navigation stack in such a way that the page displayed on the screen corresponds to the one on top of the stack. This means that when one wishes to display a new route, the `Navigator` will push this route on the top of the stack. Conversely,

when one wants to navigate back to the previous route, the `Navigator` will pop the route on top of the stack. There are several ways to indicate to `Flutter` which route to push on top of the stack. In the case of Foot 24-7, the application uses what is called **named routing**. As explained in the `Flutter` documentation [25] [26], in named routing, each screen is associated with an identifier. When we want to display a certain screen, we use the `Navigator.pushNamed` function, passing the name of the screen as an argument. To associate the different screens of the application with the appropriate names, we use the `routes` property of the `MaterialApp` widget. In some cases, it is also necessary to pass arguments to a named route. To do so, we use the `onGenerateRoute` function of the `MaterialApp` widget. All of this is implemented in the `routes.dart` file which thus allows to create the application and to call the proper screen (whose definition can be found in one of the `screens` sub-folders) of the mobile application based on the named route requested by the user.
In this file, one can also notice that the application uses the `shared_preferences` plugin [5] to store locally on the device small and simple pieces of data such as the language used in the application (*i.e.* `locale`). In this regard, we can mention that the application was structured in a way to support **internationalization** (also called localization), *i.e.* to support multiple languages in the application. The system used for internationalization is the native `Flutter` process [24] which is based on a key-value mechanism. This means that when we want to display text in the UI, instead of providing text statically in a widget, we have to create a relatively explicit key and associate it with the corresponding translations in the different languages supported by the application. The set of these key-value pairs can be found in the `l10n`[1] folder which can be seen in Figure 12. When examining this folder in more detail, we can observe that a structure allowing the support of 3 different languages (French, English and Dutch) was established. However, it is important to note that, for the moment, the application only supports one language, namely French, since many of the texts defined in the `l10n` folder are yet to be translated into English and Dutch.

`web` **folder**

`web` is the folder containing the **backend** of the application. As a matter of fact, this folder contains the `Django` project of the Foot 24-7 application. Hence, inside this folder, we can find a collection of settings for the `Django` project, including database configuration, `Django`-specific options and application-specific settings.
`Django` projects are typically divided into several `Django` apps. An app is a web application that provides specific functionality for a project while a project is a collection of configuration and apps for a particular website. This means that a project can actually contain multiple apps and that an app can also be used in multiple different projects.
In the case of Foot 24-7, the `web` folder is thus logically divided into several sub-folders, most of which representing a particular app of the project, as depicted in the following Figure 14.

[1] Localization is sometimes written in English as l10n, where 10 is the number of letters between "l" and "n" in "localization".

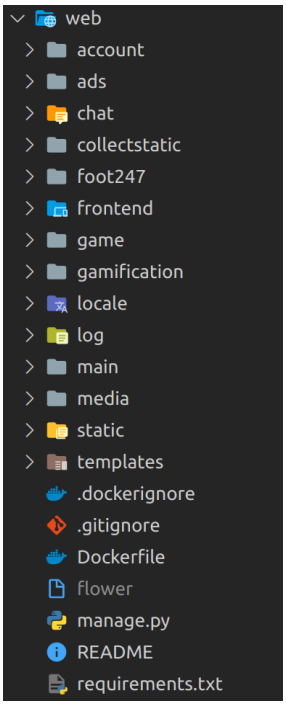


Figure 14: `web` folder

The inner `foot247` directory is the actual Python package for the project and, as such, contains the settings and configuration (such as the basic url declarations) for this `Django` project. Besides this folder, one can find several `Django` applications, namely the `account`, `game`, `chat`, `ads`, `gamification` and `main` applications.
As Foot 24-7 features both a mobile application and a web platform, these `Django` applications actually split their view and url files into two different files when needed: one for the mobile application and one for the web platform. This is why these applications generally include the same 4 files: `urls.py`, `api_urls.py`, `views.py` and `api_views.py`, those with an `api` prefix referring to the mobile application and the others to the web platform. An example of such a structure is provided in the following Figure 15.

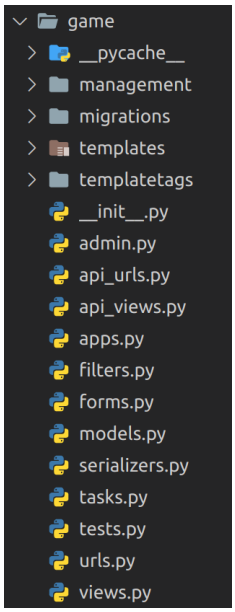


Figure 15: `Django` application structure

Besides these 2 main folders, the code structure also includes 3 other folders: the `cron`, `nginx` and `redis` folders.

`cron` folder

The `cron` folder allows to **schedule and run background tasks at fixed times, dates or regular intervals**. It is based on a system generally present in all computer systems, the `crontab`. Via the `crontab`, one can define a time interval (*e.g.* every day, every 4 hours, every minute, etc.) and assign it a task to run in parallel. This can be a purely backend task (*e.g.* checking every minute to modify a field in the database) or it can be a task with an impact on the frontend (*e.g.* notifications). Other possible uses are the renewal of SSL certificates[2] (HTTPS), a back-up system, etc.
In the case of Foot 24-7, the `cron` folder notably contains a `9am` sub-folder which schedules to send a set of notifications to the users of the mobile application every day at 9 am, as illustrated in the following Figure 16.

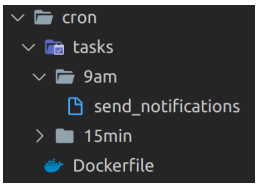


Figure 16: `cron` folder

`nginx` folder

`Nginx` [29] is a **web proxy that allows to perform load balancing**. Essentially, it is a service that allows to catch web requests and to redirect them accordingly depending on the protocol used (HTTP, websocket or other). In practice, this means that before arriving in the `Django` backend system, any request is first captured by the `Nginx` proxy (whether the request is an API request or a request to load a page, a static file such as an image, a `JavaScript` file, etc.) and the latter will then redirect it to the appropriate service (the `web` service, the `media` service, etc.). As it is a proxy, this is also where we generally define the security rules to avoid any type of network attack: addition of SSL certificates, security headers, blocking of IP addresses, limitation of upload data, 301 redirects[3], etc. There exists a multitude of possible options depending on the network needs of the project at hand.

`redis` folder

`Redis` [33] is a **database optimisation system**, based on a key-value scheme, allowing to manipulate relatively simple data (*e.g.* strings, lists, etc.). It is commonly used as a cache to store frequently accessed data in RAM so that applications can be responsive to users. To avoid losing the data stored in the `Redis` cache (in case of any problem such as a power outage), it is possible to backup the `Redis` data in a file.

[2]SSL stands for Secure Sockets Layer and, in short, is a security technology for establishing encrypted links between networked systems.

[3]A 301 redirect refers to the HTTP response status code 301 used for permanent redirecting.

To conclude this section, we can represent the application and code structure in the following high-level diagram 17.

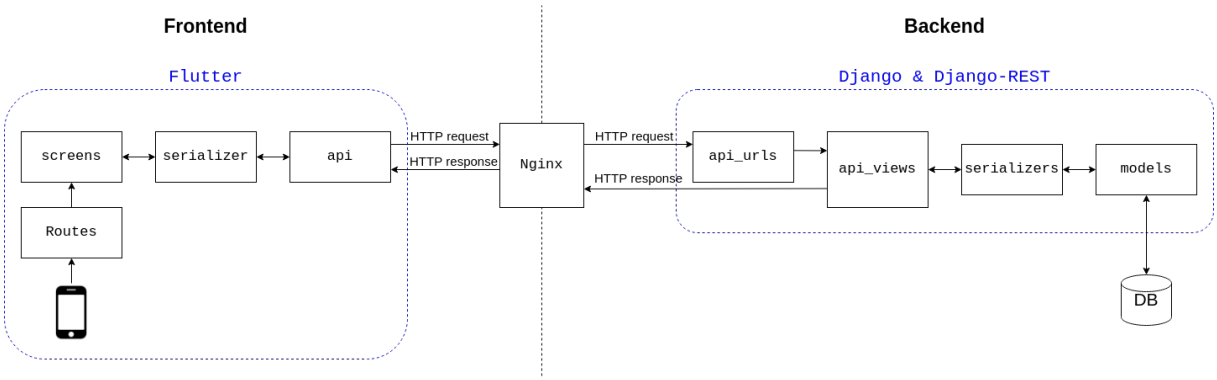


Figure 17: High-level diagram of application and code structure

3 Problem statement

In an effort to enhance the experience of their customers on their mobile application, Foot 24-7 is constantly seeking to make changes and improvements to their application in order to meet the users' expectations. In this context, Foot 24-7 was looking for an intern whose responsibility would be to **research, design and implement the best improvements to the Foot 24-7 mobile application**.

The job description for this project was quite general and suggested a great deal of **autonomy** and freedom to carry it out. As it happens, Foot 24-7's priorities were mainly focused on customer satisfaction and improving the user experience on their application but the best improvements to bring to the Foot 24-7 mobile application were yet to be determined. To answer this question, I first had to explore the application in detail in order to identify the issues that I felt were most important to address and then present and discuss them with Foot 24-7. Besides, Foot 24-7 also provided me with a list of suggestions of possible development tasks that might be interesting to develop. However, the tasks from this list that were actually considered in this thesis had to be re-evaluated according to the outcome of my research and my own suggestions, as we shall see in section 4.
In any case, it is important to realise that **this project was highly user-oriented**.

For the sake of completeness, I present here the list of task suggestions proposed by Foot 24-7:

- Establish a rating system for players based on 5 defined criteria: technical level, fair-play, physical condition, game vision and punctuality.
- Add the possibility to use Facebook credentials to sign in and log in to the application, also known as Signle Sign-On (SSO).
- Create a player invitation and referral system that allows users of the application to earn pannas by inviting their friends to join the application.
- Further development of the gamification system, which is currently only linked to the organisation of tournaments (*e.g.* rating of players encountered, creation of games, team meetings outside of championships, booking of pitches, etc.).
- Develop a pitch reservation system including integrated payment and allowing to distribute the amount paid among the different players on the application. This system should be linked to an online platform to enable pitch managers to better monitor their bookings.
- Develop a referee section where users can offer their services as referees in exchange for payment or free of charge. This implies the creation of a new status on the application and the addition of a section in the description of the games displaying the referee of the game.
- In the tournament organisation section, make improvements to the game sheet part.
- In the tournament organisation section, make improvements to the game rescheduling part in order to facilitate the modification of games. This should be achieved by providing a view directly on the application allowing players to consult the possible dates and to select them more quickly.
- In the player's profile section, the number of games played as well as the number of times the player was elected player of the game are displayed. Develop a system allowing the player's status to evolve according to different levels. This system should be based on the mechanism used in WoW-type games where different badges are unlocked according to the levels reached.

- Develop this same aspect of status evolution according to different levels for teams playing and competing against other teams.
- For games that are played between teams, set up an invitation system to optimise the organisation of such games.
- Include tournament and championship statistics in a dedicated section within each tournament.

4 Approach

In this section, we will focus on describing how I chose to approach the problem described in section 3.

As with any project in which a substantial base already exists, the first phase of this project was one of discovery and **familiarisation with the application**, the code and the functionalities already present. In fact, it was essential to understand the general philosophy of the initial project in order to integrate the new features in the most natural and coherent way but also to discover all the existing functionalities in order to steer the development of the application in the best direction. Given the significant amount of code already present, this phase was bound to require a considerable amount of time, especially since the original application actually contained many bugs. When I discovered this, I set myself the first objective of **identifying all the bugs that already existed and solving them**. Indeed, it seemed obvious to me that I first had to solve all the existing bugs before adding any new feature. This is why the familiarisation phase was particularly long since it required an in-depth walk through the entire code and the testing of all the original functionalities in order not to miss any existing bug.

Moreover, as I progressed in my understanding of the initial application, code and functionalities, I was able to **forge my own critical opinion on the application**.
First of all, from my personal experience with the application, I thought that the user experience could be further improved. Beyond the fact that the original application suffered from several bugs, there are in my opinion several other elements that put a ceiling to the quality of the user experience on the application: the user interface is not always the cleanest and most attractive (*e.g.* the "Pannas" tab, the profile page tutorial, the presentation of the ads, etc.). Besides, the access to some features or information is not always very intuitive (*e.g.* the "Créer une équipe" button should not be found in the settings of the profile page). All these elements reduce the quality of the user experience on the application. However, as UX design was not part of Foot 24-7's requests nor of the expectations for a Master thesis in computer science, it was not my place to deal with such issues in the context of this thesis. Nevertheless, some inconsistencies in the original application were too important to be ignored. In such cases, Foot 24-7 and I agreed on developing a more user-friendly solution (*e.g.* see the redesign of the invitation system in section 5.6). For the rest, I tried to integrate my newly developed features in the most intuitive way possible into the existing application. In any case, I do believe that it would be of great value for the future of Foot 24-7 to work on UX design in order to increase the application's consistency and to further improve the user experience.
From a more technical and less user-oriented point of view, I also found that some elements of the initial code structure could be improved. For instance, while discovering the initial source code, the `game` `Django` application felt a bit heavy (*i.e.* definition of many different models, views and serializers). To lighten this application, we could split it into two different applications: `game` and `tournament`, which would allow to reorganize things more clearly. In addition, some of the models defined in the `game` application should probably not be defined in this application and instead be relocated in the `main` application (*e.g.* `City`, `Stadium`). Moreover, while browsing the source code, I also sometimes noticed duplication and redundancy of certain pieces of code, which is perhaps due to the fact that several different people were involved in the implementation of this project. However, it is important to mention that, although the structure of the code is not always ideal, the code remains perfectly understandable and transferable as it is relatively natural and easy to grasp the functioning and philosophy of the code the first time one goes through it. Therefore, in order to avoid spending too much time on these structural details and in the interest of the people who developed the initial application and who will eventually take over the project after my thesis, I decided to prioritise the addition of new features in a way that would give the impression that a single person had implemented the whole project. In any

case, restructuring some elements of the code structure would also be a potential development prospect for the future of Foot 24-7.
Furthermore, as I went through the code, I also noticed that the documentation of the original code, although reasonable, was relatively poor. However, since the code structure is fairly natural and consistent, this level of documentation is sufficient to approach and understand the project.
Besides all these concerns, it is important to mention that I also decided to **draw up my own list of development points** that I felt were relevant or even **essential to address**, whether they were improvements or corrections to existing features or additions of features not initially suggested by Foot 24-7. Among these development points, we can find:

- Complete redesign of the invitation system allowing to invite players to a game or a team. Actually, the initial invitation system was quite unintuitive and presented inconsistencies (see section 5.6).
- Addition of the possibility to create a game by specifying teams rather than players (see section 5.4).
- Addition of a team rating system to reinforce the sense of responsibility of users (especially team captains) thereby rendering the organisation of games through the application more reliable (see section 5.3).
- Addition of a "Mes équipes" tab allowing users to quickly and easily access the teams to which they belong or for which they have received an invitation (see section 5.5).
- Addition of the possibility to edit a created game (see section 5.7).
- Improvement of the player participation survey tool in a tournament game.

Once I had presented these different issues to Foot 24-7, we agreed to consider these matters (and most of them as a priority) during this thesis.

In this continued effort to find the best improvements to bring to the Foot 24-7 mobile application, I also decided to **interview a few actual users of the application** to try and understand what they liked about the application, what they did not like and what they would like to see on the application in the future. To do so, I created a little questionnaire containing 3 simple questions:

- What do you like about the Foot 24-7 mobile application ?
- What don't you like about the Foot 24-7 mobile application ?
- What are your suggestions for improving the Foot 24-7 mobile application ? What new features would you like to see in the app ?

Between February and April 2022, I submitted this questionnaire to a small group of users of the application that I knew, which eventually allowed me to collect 9 answers[4]. Among the collected answers, the most frequently mentioned remarks related to fixing the bugs of the application, developing gamification as well as improving the user interface and more generally the user experience on the mobile application. The details of the users' feedback are given in Appendix A.

Following these considerations, I provide in the following Figure 18 a roadmap of the project. The general framework of this roadmap was established at the beginning of this thesis but was of course adjusted during the course of the project. This roadmap graphically and temporally presents the different points discussed in sections 5 and 6.

[4]Conducting an extensive survey based on a larger sample of users was beyond the scope of this project, but could certainly be of great interest in order to get a clear and complete picture of what users like and dislike about the Foot 24-7 mobile application.

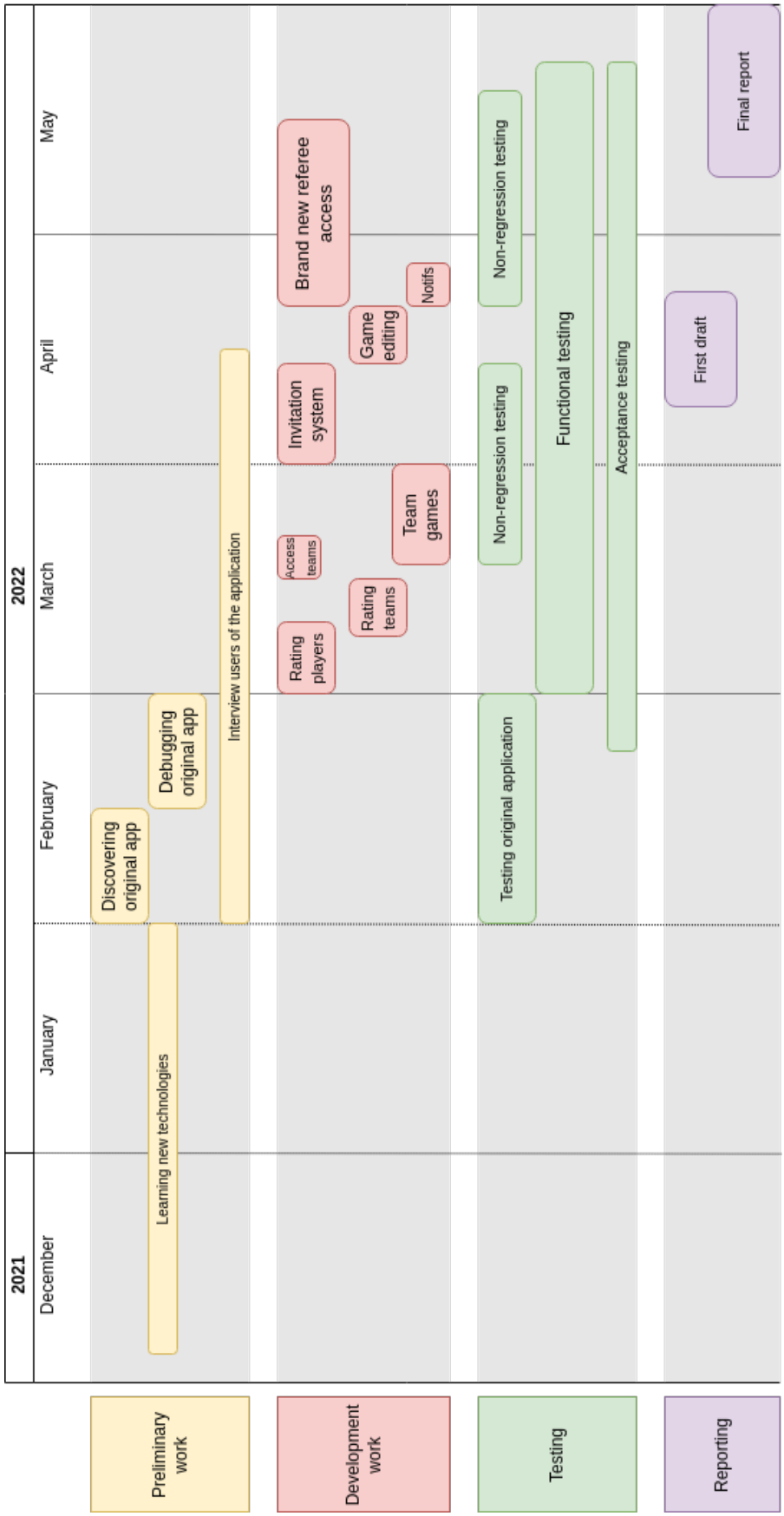


Figure 18: Roadmap of the project

5 Development of the solution

In this section, we will describe the solution developed in the course of this thesis, discuss the technical details related to the implementation of this solution as well as present the obtained results.

5.1 Debugging the original application

As discussed in the previous section 4, the discovery phase of the initial mobile application revealed the **presence of many bugs, errors or inconsistencies in the original application**. Before considering the development of any new functionality, it was therefore essential to resolve all these errors first. A summary table listing the main defects of the original application is provided at the end of this section in Table 1. We will now go through these main defects to describe their resolution:

1. A first error in the initial application concerns the addition of a player to a team. When a user tries to add a player to one of his teams, he is redirected to a new page allowing him to search for a player to invite among all the players registered in the application. However, when redirected to this page, an error from the `mobile_app/lib/team/screens/team_invite.dart` file is raised when building the list of candidate players. The details of this error are as follows:

   ```
   The following NoSuchMethodError was thrown building TeamInviteScreen(dirty,
   dependencies:  [_LocalizationsScope-[GlobalKey#63b2e]], state:
   TeamInviteScreenState#cfa9c):
   The getter 'itemList' was called on null.
   Receiver:  null
   Tried calling:  itemList
   ```

   The problem can be solved by transforming the `_buildPlayerListContainer` widget contained in the `build` method of the `TeamInviteScreenState` into a function and by using null-aware operators (*i.e.* using the `?.` operator rather than the `.` operator). In fact, when we define `_buildPlayerListContainer` as a widget inside the `build` method of the `TeamInviteScreenState`, this widget will be built as soon as the `build` method is called. However, at that time, the list of candidate players has not yet been fetched from the backend, which explains why the `_players` variable is null and why the error arises. By turning this widget into a function, it allows to build the `Container` only when the function is called, *i.e.* once the list of candidate players has been retrieved.

2. Another error relates to the creation of a team: when a user creates a team, the application does not save the availabilities and schedules specified by the user and instead saves the availabilities and schedules values set by default when reaching the `TeamCreationScreen`. The problem comes from the data serializers. Indeed, the `TeamCreateSerializer` defined in the `account.serializers.py` file does not take into account the `availability` and `schedule` fields of the `Team` model, which explains why the default values are always used. By including these fields in the `TeamCreateSerializer`, the application does save the availabilities and schedules specified by the user when creating the team.

3. When a user creates a new game, a `RenderFlex overflow` can occur on the game's details page if the organiser's name is too long. `RenderFlex overflow` is one of the most frequently encountered `Flutter` framework errors. To solve it, we can simply wrap the `TextStyles` widget containing the organiser's name with an `Expanded` widget and set the `hasOverflow` property of the `TextStyles` widget to true. The exact same problem can also occur with the address of the stadium where the game is taking place if this address is too long. In a similar way, we can solve this problem by encapsulating the `Container` widget containing the game address in an `Expanded` widget and by setting the `hasOverflow` property of the `TextStyles` widget containing the address to true.

4. When a user creates a new game, the time of the created game is always shifted one or two hours earlier (depending on whether daylight saving time (DST) is used or not) than the time requested in the game creation form. This can be explained by the fact that the `toMap` method of the `GameCreate` class used to convert the `Dart` object into a `JSON` object in order to send it to the `Django` backend converts the date of the game into the UTC time zone while the support for time zones is enabled in the `Django` project [34]. One can see that time zone support is enabled by consulting the `web/foot247/settings.py` file to see that it specifies `USE_TZ = True`. When support for time zones is enabled, `Django` stores datetime information in UTC in the database, uses time-zone-aware datetime objects internally, and translates them to the end user's time zone when they are requested. This actually means that, when a user creates a game on the mobile application, `Django` expects to receive a date in local time and not in UTC. To solve this problem, we need to remove the `toUtc` method used in the `toMap` method of the `GameCreate` class.

5. When a user creates a new game with the default duration of 90 minutes, the created game appears to last 1 minute and 30 seconds in the "Parties" tab. This can be explained by the fact that the frontend sends a `String` "90" to the backend which interprets it in seconds rather than minutes, which explains why "90" is actually translated into 1 minute 30 seconds rather than 1h30. To solve this problem, one simply has to convert the `String` containing the duration in minutes into a `String` containing the duration in seconds in the `toMap` method of the `GameCreate` class.

6. When creating a game, the creator of the game does not appear in the list of players in the `GameDetailScreen`, which seems inconsistent. Furthermore, if we follow the convention adopted for the creator of a team, the creator of a game should appear in the list of players with a star on his icon to indicate that he is the creator of the game. To solve this problem, one should avoid using the `get_players` method of the `Game` model as a source for the list of players in the `GameDetailSerializer` (see the `web/game/serializers.py` file) and instead use the `players` field directly.

7. When trying to add a player to a game, the added player is not the one selected when clicking on his icon in the list of players in the `GameDetailScreen` although the player appearing in this list is the correct one. The problem only occurs when clicking on his icon. Indeed, once the user clicks on his icon, he is redirected to a profile that does not correspond to the actual added player or worse, an error arises. As for the previous error, the problem comes from the fact that one should avoid using the `get_players` method of the `Game` model as a source for the list of players in the `GameDetailSerializer` and instead use the `players` field directly. As a matter of fact, the `get_players` method of the `Game` model returns the `gameplayer_set` associated to the particular game which corresponds to a `QuerySet` of `GamePlayer`s and not to a `QuerySet` of `Player`s. Since it is necessary to provide the primary key of a `Player` and not of a `GamePlayer` to redirect to that `Player`'s `ProfileScreen`, this explains why the redirection to a player's profile fails when using a `GamePlayer` instead of a `Player`.

8. Removing a player from a game does not work and throws an HTTP 500 error:

```
I/flutter (14708):  500
[ERROR:flutter/lib/ui/ui_dart_state.cc(209)] Unhandled Exception:
Exception:  Echec de l'action
E/flutter (14708):  #0 removePlayerGame
package:mobile_app/.../api/api.dart:207
E/flutter (14708):  <asynchronous suspension>
E/flutter (14708):  #1 _GameDetailScreenState._removePlayer
package:mobile_app/.../screens/game_detail.dart:114
E/flutter (14708):  <asynchronous suspension>
```

   Once again, the problem comes from the fact that one should avoid using the `get_players` method of the `Game` model as a source for the list of players in the `GameDetailSerializer` and instead use the `players` field directly.

9. When a user completes his profile, he receives a message stating that he received 100 pannas because he filled in all the information in his profile. However, the user is actually awarded 200 pannas. To solve this inconsistency, one simply has to change the default value of the `pannasNumber` variable in the `profile_progress_dialog.dart` file from 100 to 200 so that it matches with the number of pannas associated with the `profile_completion` achievement.

10. When the creator of a team tries to modify the level, availabilities, schedules or description of his team on the `TeamDetailScreen`, the changes are not saved in the database. It is therefore impossible for a team creator to change this information. Although there exists an `_updateTeam` method in the `TeamDetailScreen`, it is actually never used. To solve this problem, the `_updateTeam` method should be called either when the user selects an option in a select dialog (such as for the level, availabilities or schedules) or when the user presses the "Done" button on the keyboard to edit a `TextField` (such as for the team description). For this last case, it is required to modify the `widgets/input_field.dart` file in order to authorize the possibility to customize the `onEditingComplete` property of a `TextField`. In fact, this property is called when the user submits an editable content (*e.g.* when the user presses the "Done" button on the keyboard). It is therefore necessary to call the `_updateTeam` method in this property.

11. A member of a team has no possibility to leave the team he belongs to. To allow a team member to leave the team, one has to test the value of the boolean `_team.canLeave` in the `_buildBottomWidget` widget of the `team_detail.dart` file. If this value is true, then the `_buildBottomWidget` widget needs to return a button allowing the user to leave the team.

12. When a user clicks on his own icon in the `TeamDetailScreen` associated with one of the teams he belongs to, he is redirected to the visitor page of his profile when he should be redirected to his own profile. To solve this problem, one needs to compare the primary key of the player whose profile one wants to reach with the primary key of the current user. If the two keys are identical, then the user should be redirected to his own profile page. If not, the user should be redirected to the visitor page of the player's profile.

13. When a user searches for a particular team, filtering teams by name does not work properly. Actually, if the team contains spaces or capital letters in its name, then the search filtered on the name will not provide the expected result. To solve this problem, we must check that the name contains rather than starts with the `String` entered in the search form. In practice, this means that one must use the `icontains` function rather than the `istartswith` function when filtering the `queryset` on the name in the `get_queryset` method of the `TeamListView` in the `web/account/api_views.py` file. Moreover, it is also necessary to use the `trim` function on the name entered in the form to remove leading and trailing whitespaces when trying to sort the teams on their name (see `_fetchTeamList` in the `team_filter.dart` file).

14. When a team captain wants to add a player to one of his teams, he is redirected to the `TeamInviteScreen`. However, on this page, the captain is unable to filter players by name, which is very inconvenient. To solve this problem, we need to add an `InputField` to allow filtering players by name in the `_buildFilterContainer` widget of the `TeamInviteScreen`.

15. When a user creates a game, some stadiums lead to an error in the last summary page of the `CreationScreen` because the `String` containing the phone number associated with the stadium is null. To solve this, one should only display the stadium phone number information in the summary page when the `String` containing the phone number is not null.

16. On a team's detail page, the number of games played by that team is displayed. However, this number includes all the games in which the team appears (including games that have not yet been played) when this number should only include games that have already been played (*i.e.* in the past). To solve this, one has to modify the `get_game_count` method of the `TeamDetailSerializer` by filtering all the games according to their date in order to keep only those with a date lower than the present time. Similarly, on a player's profile page, the number of games played by that player is displayed. However, this number also includes games that have not yet been played. To solve this, one has to modify the `get_number_games` method of the `Player` model by filtering the games according to their date and keeping only those in the past.

17. When a chat contains more than 20 messages, a bug appears when a user belonging to this chat tries to send a new message in this chat or when he tries to load old messages by pressing the "Plus de messages" button. Indeed, beyond 20 messages in a conversation, only the last 20 messages of this conversation are initially displayed when a user clicks on this chat. This is because in this project the `Django REST` framework uses a system called **pagination** [30]. This pagination system applies when the `Flutter` frontend tries to retrieve a list of data from the `Django` backend and consists in using **lazy loading**. This means that when a user of the application tries to retrieve a list of messages (or even a list of players, teams, games, stadiums, etc.), only a small portion of the data in the database is retrieved on each call. This allows to avoid long loading times during the initialization of a page and thus a greater fluidity in the application. As an analogy, the pagination system can be seen as a book (hence its name):

    - The set of data (in this case, the messages of a chat) represents the book.
    - Each piece of data represents a word or a sentence.
    - And each call returns a page of this book. We will first call page 1, then if we want to scroll further page 2 and so on until we have retrieved all the data and thus gone through the whole book.

In this project, we can see that this pagination system is initialized in the backend via the `pagination.py` and `settings.py` files in the `web/foot247` folder. For this project, we have set a `page_size` to 20, which means that 20 pieces of data are retrieved per call before having to call the next page. This explains why the limit of messages displayed in a chat by default when this chat is clicked on is equal to 20. However, when a user tries to send a new message in a chat with more than 20 messages or when he tries to load old messages by pressing the "Plus de messages" button, the chat is updated in such a way that it no longer contains the last 20 messages sent. Nevertheless, if the user leaves this chat and then returns, all the exchanged messages are there. In fact, the problem comes from the `loadMessage` method defined in the `ChatDetailScreen`. This function is called when a user tries to send a new message in a chat or when he tries to load old messages by pressing the "Plus de messages" button. The problem is that when the first page of messages has already been retrieved, this method simply fetches the next page and replaces the list of messages with those of the next page, which is why the messages of the previous page disappear. To solve this problem, it is therefore necessary, in the case of retrieving a page which is not the first one, to add the messages of the retrieved page to the previous list of messages (rather than replacing it).

18. Finally, we can also mention that there exist certain specific teams that cause an error when a user tries to access their details page. After further investigation, it becomes clear that the problem comes from the fact that these teams contain members who possess a user account but have no corresponding player entity. To understand the problem, it is important to realise that, in the original application, a user account is represented in the database using the `Account` model (see the `account/models.py` file). However, this model does not allow to represent a player. As it happens, the player entity is represented using another separate model called `Player`. To link these two models together, the `Player` model defines a `OneToOneField` called `user` referring to the `Account` model. When a user decides to create an account on the Foot 24-7 mobile application, he thus creates two related entities: an instance of the `Account` model and an instance of the `Player` model. On the mobile application, only players are allowed to create or join a team. Therefore, how is it possible that users with no `Player` instance associated with their account appear in teams? After investigation and discussion with Foot 24-7, we realized that the problem came from the fact that these user accounts were created manually via the admin site by Foot 24-7 itself. However, these users were created without instantiating a corresponding player entity and then wrongly added to teams, which thus explains the source of the problem. After analysing the data of the initial database, I realised that there were 22 of these user accounts that had no associated player entity, but yet appeared in teams. The solution to this problem is very simple: these user accounts must be deleted as they do not correspond to any actual user of the Foot 24-7 mobile application.

Besides all these errors, there also exist many minor inconsistencies in the application. To name just one, when a user selects a tournament on the mobile application, he is redirected to a page allowing him to select a tournament group. However, as we can actually see in the presentation of the application in Figure 4b, the highlighted tab in the bottom navigation bar on this page is the "Calendrier" tab when it should be the "Tournois" tab.

Finally, let us specify here that the list of errors and inconsistencies in the original application presented throughout this section and summarized in the following Table 1 is **not an exhaustive list**. However, in order to avoid making this report excessively long, we will stop here in our description of the debugging of the original application.

Defect ID	**Defect description**	**Impact on user experience**	**Type**
D_01	Null error when building the list of players that a captain can invite to his team.	Low	Minor
D_02	When creating a team, the application does not save the availabilities and schedules specified by the user and instead saves the default values for these fields.	High	Major
D_03	`RenderFlex overflow` on a game's details page if the organiser's name or the address is too long.	Mild	Minor
D_04	Time of a created game always shifted by one or two hours.	High	Major
D_05	Duration of a created game interpreted in seconds rather than in minutes.	Mild	Minor
D_06	Game creator does not appear in list of game players.	Mild	Minor
D_07	Clicking on the icon of a game player does not redirect to the right player profile.	High	Major
D_08	Removing a player from a game does not work.	High	Major
D_09	Gain of 200 pannas when completing one's profile while the alert message specifies a gain of 100 pannas.	Low	Minor
D_10	Changes that a captain makes to his team are not saved.	High	Major
D_11	Impossible for a team member to leave the team.	High	Major
D_12	When clicking on his own icon in a team's details page, the user is redirected to the visitor view of his profile instead of his actual profile page.	High	Major
D_13	Filtering teams by name does not work.	High	Major
D_14	Impossible to filter players by name when trying to add a player to a team.	High	Major
D_15	Null error in game creation form if stadium phone number is null.	Low	Minor
D_16	Inclusion of future games when counting the number of games played by a team or a player.	Low	Minor
D_17	Missing messages in a chat containing more than 20 messages.	High	Major
D_18	Users with no player accounts included in teams.	High	Major

Table 1: Main defects in the original application

5.2 Rating other players

5.2.1 Description

In order to enhance the user experience on their mobile application, Foot 24-7 wanted to offer its users the possibility to rate other players on the application.
The player rating system should allow to rate a player according to 5 different criteria: technical level, fair play, physicality, game vision and punctuality.
When consulting a player's profile (either one's own profile or another player's profile), the average grade obtained as well as the number of ratings received should be displayed below the profile name.
It is also important to mention that **a player should only be able to rate players he has already met**, either in unofficial games or in tournament games.
Furthermore, this task also includes the **integration of gamification** so that a player earns pannas when rating another player.
Finally, it is also necessary to implement a **notification system** that would invite users to rate the players they have met during a game. In this context, for each completed game, all participants of this game should receive a notification suggesting to rate the other participants.

5.2.2 Implementation & Results

On the backend, the integration of this new feature requires first of all the definition of a new `UserNote` model including as fields the player who rates, the rated player but also the 5 rating criteria. In order to develop a star-based rating system ranging from 0 to 5 stars, the fields corresponding to the 5 rating criteria should all be `IntegerField`s between 0 and 5.
In order to ensure that a player can only rate players he has already played against, one has to implement a `get_encountered_players` method in the `Player` model defined in `account/models.py` that would return the list of player ids the player has already met. However, since this list of encountered players can potentially be very long, it is not ideal to include this information directly in the `JSON` object sent from the backend to the frontend to determine whether the current user can rate a player or not. Instead, one should rather define another method called `get_can_rate` in the `VisitProfileDetailSerializer` that would use this `get_encountered_players` method to determine if the profile the current user is visiting matches a player he has met before. This would allow to summarize the information sent to the frontend in a single boolean rather than in a potentially long list of integers and thus to optimise the performance of the application.
Furthermore, it is also important to ensure that **a player can only assess another player once**. To do so, one can implement a `get_rated_players` method in the `Player` model that returns the list of player ids the player has already rated. For the same reasons as those mentioned above, this method should then be used in a `get_already_rated` method of the `VisitProfileDetailSerializer` in order to determine whether the profile consulted by the current user corresponds to a player he has already rated or not.
Lastly, in order to display the average grade and the number of ratings received by a player in the frontend, one must also define two additional methods `get_average_note` and `get_number_notes` in the `Player` model.

On the frontend, one first has to define a new `_widgetNotationInfo` in the `ProfileScreen` which would be in charge of building the rating bar related to a certain profile. To do this, one can use the `Flutter` package called `flutter_rating_bar` [4]. Indeed, this package offers a simple yet fully customizable way to build rating bars as well as rating bar indicators, supporting any fraction of rating.
Once reaching a profile page, we need to test whether the `playerPk` variable in the `ProfileScreen` is non-null in order to determine whether the user is currently visiting another player's profile

or his own. If the user accesses his own profile, one should only build a rating bar indicator instead of an editable rating bar. As mentioned earlier, this rating bar indicator should be displayed below the profile name along with the number of ratings the current user has received, as illustrated in the following Figure 19.

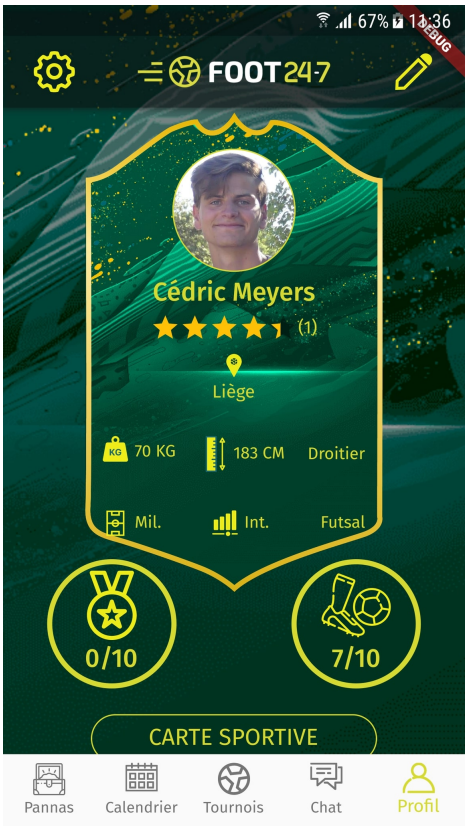
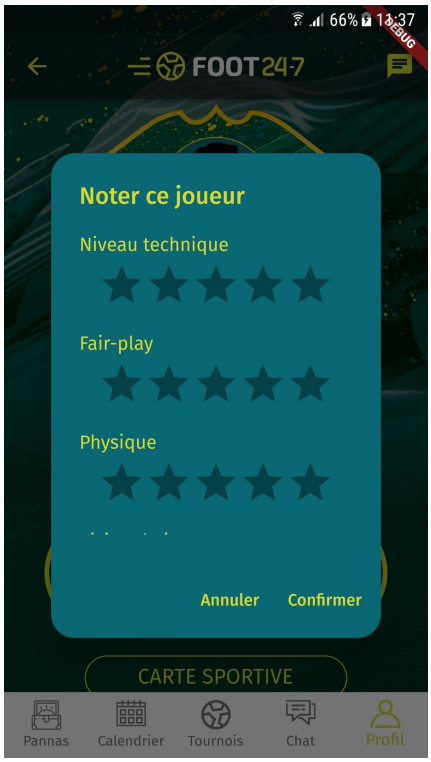
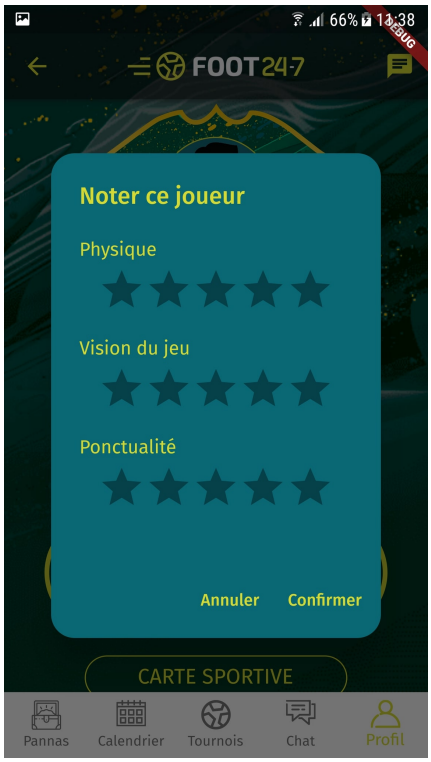


Figure 19: Profile page containing the rating of the current user

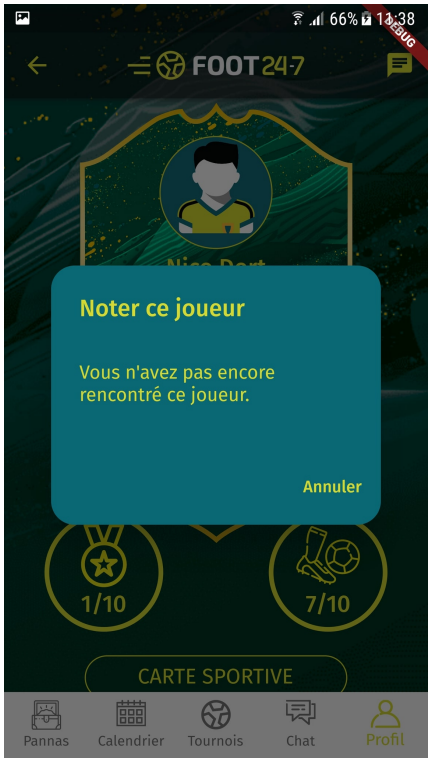
On the other hand, if the user visits another player's profile and clicks on the rating bar indicator, a rating dialog should appear on the screen. This dialog should either allow the user to fill in a form if the user has already met this player but has not yet rated him (see Figures 20a and 20b), or notify the user that he can not rate this player because he has not yet met him (see Figure 20c), or notify the user that he has already rated this player and therefore can not do so again (see Figure 20d). To distinguish between these 3 different cases, one must use the 2 booleans (`can_rate` and `already_rated`) provided by the backend via the two methods discussed above. The 3 different player rating dialogs are illustrated in the following Figure 20.



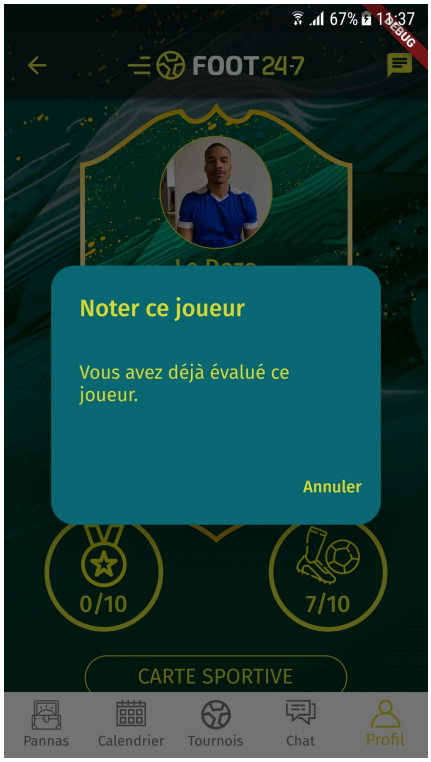
(a) Player rating form (first part)



(b) Player rating form (second part)



(c) Player not met



(d) Player already rated

Figure 20: 3 different player rating dialogs

As regards gamification, one should first start by adding a new type of achievement `player_rating` associated to 100 pannas via the admin site. In the `_createUserNote` method defined in the `profile/screens/profile.dart` file, one should then call the `achievementDone` function defined in the `gamification/api/api.dart` file in order to create a new `Achievement` in the backend. In this process, the `achievementDone` function should also be updated so as to take into account this new type of achievement. As a matter of fact, when a user completes a new achievement, it is essential to generate and display an alert dialog to inform the user that he has won pannas and why. Introducing this new type of achievement therefore involves defining a new appropriate `alertMessage` in the `achievementDone` function.
Since an achievement also includes a `meta_information` field allowing to uniquely identify this achievement, we can also define the `meta_information` for this type of achievement in the following way: `"notedPlayerPk_playerPk"`.
The alert dialog displayed to the user when assessing a player is illustrated in the following Figure 21.

Figure 21: Gamification alert dialog when rating a player

Regarding the notification system, one solution would be to implement a recurring task whose job would be to send notifications every morning inviting the various players of the application to rate the players they met in games played the day before. As mentioned in the section 2.4, implementing recurring tasks can be achieved using `cron`. In the corresponding `cron` folder, one can find in particular a `9am` sub-folder which schedules to send a set of notifications to the users of the mobile application every day at 9 am (see the `cron/tasks/9am/send_notifications.sh` file). In order to send notifications, this file relies on the implementation provided in the `web/main/management/commands/send_notifications.py` file. As it happens, this file defines a function called `send_notifications` which was initially responsible for sending reminder notifications on game days. This function can thus be used to add notifications about games played the day before. Once these notifications are added to the `send_notifications` function, users will then receive notifications inviting them to rate the participants of games they played the previous day as shown in Figure 22.

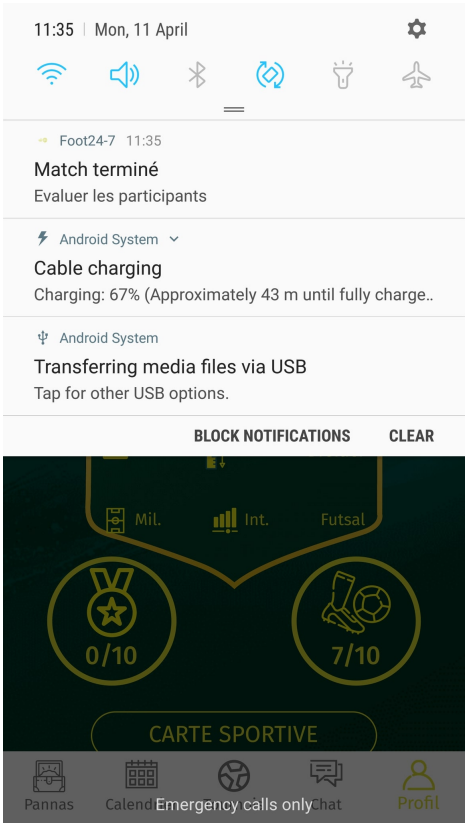


Figure 22: Notification inviting a user to rate game participants

When clicking on such a notification, the user should then be redirected to a specific page listing all the participants who took part in the particular game. At this point, it is important to clearly understand what a game participant actually is. As we shall see later in this report, adding the possibility to create a game by specifying teams rather than players (see section 5.4) as well as adding the possibility to rate teams according to different criteria (see section 5.3) were also part of this project. This thus means that a game participant can either be a team or a player. Therefore, the screen listing the participants of the completed game could potentially display a list of teams and a list of players.
Once reaching such a page, the user would then be able to click on any participant in order to access their details page and be able to rate them. As this mechanism needs to be implemented for both unofficial and tournament games, it is necessary to define two new screens `GameParticipantListScreen` and `TournamentGameParticipantListScreen` in the frontend. In order to redirect the user to the appropriate screen when he clicks on such a notification, one also has to update the `notificationRedirect` function defined in the `mobile_app/lib/firebase_notification_handler.dart` file.
For illustrative purposes, an example of a `GameParticipantListScreen` is shown in the following Figure 23.

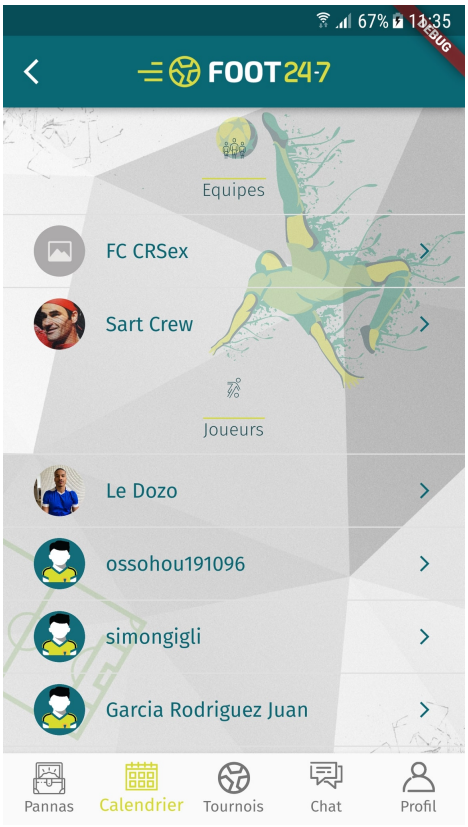


Figure 23: Page displaying the list of game participants

5.3 Rating other teams

5.3.1 Description

While I was working on this project, I once went to play soccer with some friends in a hall of the Blanc Gravier. Before our game, a game between two teams using the Foot 24-7 mobile application was to take place. However, one of the two teams never showed up for this game, leaving the other team short-handed to play football properly. The missing team had given absolutely no warning to the other team that they would not be coming after all. Therefore, all the players of the other team had made the trip to come and play but ended up wasting their time. This episode made me question the quality of the user experience when organizing football games on the Foot 24-7 mobile application. Indeed, this kind of inconvenience greatly diminishes the value and relevance of the application. To remedy this, I thought that adding a team rating system based on several criteria would make players (and particularly team captains) more accountable, encourage them to honour their commitments to other users of the application and thus **make the organisation of games via the application more reliable** (hence improving the user experience). I thus proposed to add this functionality to Foot 24-7 who, upon hearing my motivations, immediately validated this task.
This task is actually very similar to the one described in section 5.2. As a matter of fact, the team rating system should allow to rate a team according to various criteria, just as for the player rating system except that, in the case of teams, team rating should be based on 3 criteria: level of play, fair play and reliability.
When consulting a team's details page, the average grade obtained as well as the number of ratings received by this team should be displayed below the team's name. In addition, the details of the grades received by this team according to the 3 criteria must also be displayed in a dedicated section at the bottom of the team's details page.
Similarly to the player rating system, it is also important to mention that **a player should only be able to rate teams he has already played against**, either in unofficial games or in tournament games.
Furthermore, this task once again includes the **integration of gamification** so that a player earns pannas when rating a team.
Finally, as already mentioned in the section 5.2, it is also necessary to integrate the teams that took part in a game into the **notification system** that suggests to users to rate the participants of a game.

5.3.2 Implementation & Results

On the backend, the integration of this new feature requires first of all the definition of a new `TeamNote` model including as fields the player who rates, the rated team but also the 3 rating criteria. Just as for the player rating system, the team rating system must based on a star rating system ranging from 0 to 5 stars. Hence, the fields corresponding to the 3 rating criteria should all be `IntegerField`s between 0 and 5.
In order to ensure that a player can only rate teams he has already played against, one has to implement a `get_encountered_teams` method in the `Player` model that would return the list of team ids the player has already met. Similarly to the player rating system, since this list of encountered teams can potentially be very long, it is not ideal to include this information directly in the `JSON` object sent from the backend to the frontend to determine whether the current user can rate a team or not. Instead, one should rather define another method called `get_can_rate` in the `TeamDetailSerializer` that would use this `get_encountered_teams` method to determine if the team's details page the current user is visiting corresponds to a team he has met before. This would allow to summarize the information sent to the frontend in a single boolean rather than in a potentially long list of integers and thus to optimise the performance of the application.
Furthermore, it is also important to ensure that **a player can only assess a team once**. To

do so, one can implement a `get_rated_teams` method in the `Player` model that returns the list of team ids the player has already rated. For the same reasons as those mentioned above, this method should then be used in a `get_already_rated` method of the `TeamDetailSerializer` in order to determine whether the team's details page consulted by the current user corresponds to a team he has already rated or not.
Lastly, in order to display the average grades (global and according to the 3 criteria) as well as the number of ratings received by a team in the frontend, one must also define four additional methods `get_average_level_of_play`, `get_average_fair_play`, `get_average_reliability` and `get_number_notes` in the `Team` model.

On the frontend, one first has to integrate into the `_buildTitleNameTeamContainer` widget of the `TeamDetailScreen` a rating bar indicator along with the number of ratings received by the team under consideration, as illustrated in the following Figure 24. Here again, in order to build rating bars or rating bar indicators, one can use the `Flutter` package `flutter_rating_bar` [4].

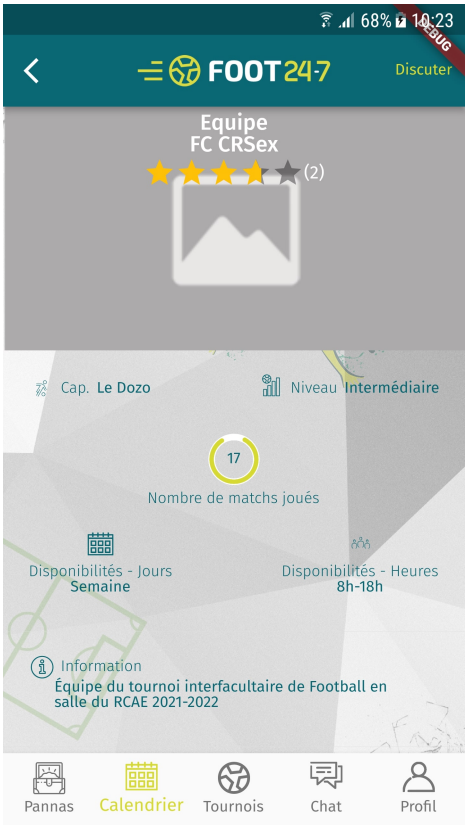


Figure 24: Global rating bar indicator on a team's details page

Besides, one has to add a section about rating at the very bottom of the `TeamDetailScreen` where the 3 rating criteria are outlined, each with a rating bar indicator. As requested by Foot 24-7, this section should be included in an `ExpansionTile`. Furthermore, it is also important to remember that a user can only rate a team he does not belong to. In this case, this section should also include a button enabling him to rate the team if he is allowed to do so, as depicted in the following Figure 25.

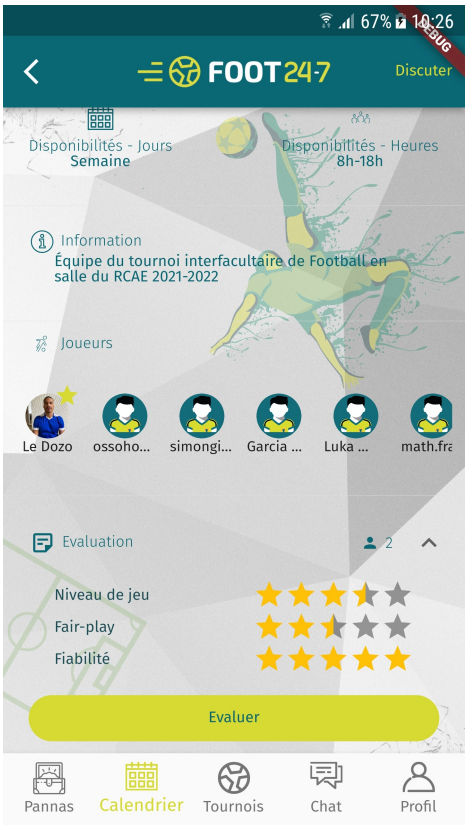
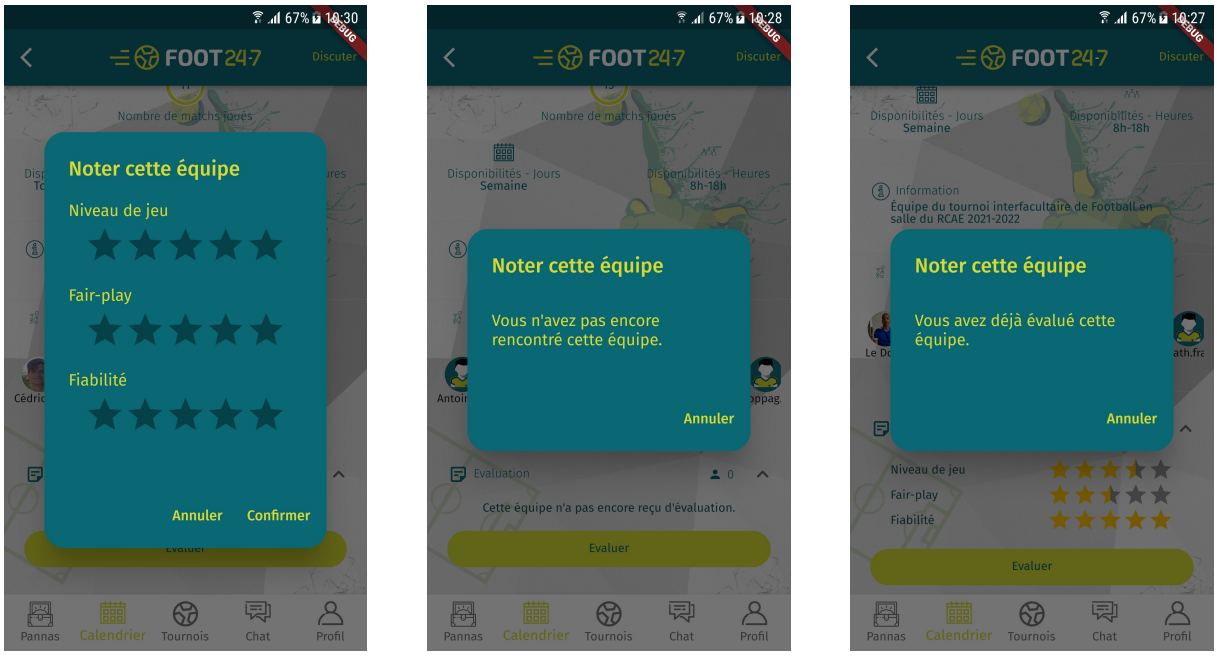


Figure 25: Rating section on a team's details page

When clicking on the "Evaluer" button on a team's details page, a rating dialog should appear on the screen. As for the player rating system, this dialog should either allow the user to fill in a form if the user has already met this team but has not yet rated it (see Figure 26a), or notify the user that he can not rate this team because he has not met it yet (see Figure 26b), or notify the user that he has already rated this team and therefore can not do so again (see Figure 26c). Here again, to distinguish between these 3 different cases, one must use the 2 booleans (`can_rate` and `already_rated`) provided by the backend via the two methods discussed above. The 3 different team rating dialogs are illustrated in the following Figure 26.



(a) Team rating form (b) Team not met (c) Team already rated

Figure 26: 3 different team rating dialogs

As regards gamification, one should first start by adding a new type of achievement `team_rating` associated to 100 pannas via the admin site. In the `_createTeamNote` method defined in the `team/screens/team_detail.dart` file, one should then call the `achievementDone` function in order to create a new `Achievement` in the backend. In this process, the `achievementDone` function should also be updated so as to take into account this new type of achievement. In fact, one has to update the `alertMessage` displayed to the user in the dialog generated by the completion of the achievement.
Since an achievement also includes a `meta_information` field allowing to uniquely identify this achievement, we can also define the `meta_information` for this type of achievement in the following way: `"notedTeamPk_playerPk"`.
The alert dialog displayed to the user when assessing a team is illustrated in the following Figure 27.

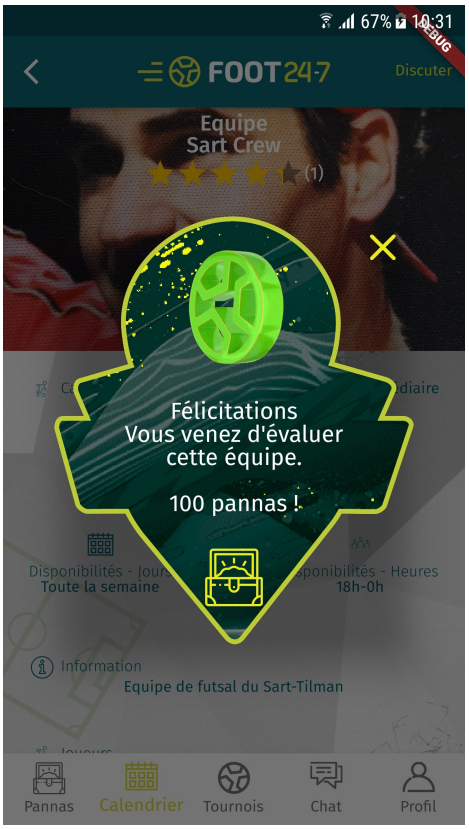


Figure 27: Gamification alert dialog when rating a team

Regarding the notification system, as mentioned in the section 5.2, adding the possibility to rate a team as well as the task allowing to create a game by specifying teams rather than players (see section 5.4) imply that a team can also potentially be part of the participants of an unofficial game (and is necessarily part of the participants of tournament games which can only be organised around teams). This must therefore be taken into account in the `GameParticipantListScreen` and the `TournamentGameParticipantListScreen`. As a reminder, one can refer to Figure 23 showing the addition of teams to the list of game participants.

5.4 Create games with teams rather than players

5.4.1 Description

While discovering the Foot 24-7 mobile application, I noticed that it was not possible to create an unofficial game by specifying teams as participants. Actually, in the original application, a user can only create a game by adding players individually. I immediately found this to be quite inefficient in the case where a user already has a predefined team and would simply like to play against another one in an unofficial game. In the continued effort to improve the user experience, I thus proposed to add this feature to Foot 24-7 who agreed with my suggestion.
In the context of this task, a user can only create a team game if he is himself the captain of at least one team. When creating the game, he must specify that it is a team game by selecting one of the teams he is captain of.
Besides, it should be possible for a user who is captain of a team to join a team game with one of his teams if that game is public and the second team does not yet exist.

5.4.2 Implementation & Results

Because this task requires a change in the `Game` model, it will actually have **many repercussions on the whole application**. As it happens, the first thing to do is to update the `Game` model by adding 2 fields: `first_team` and `second_team`. Yet, until now, a team was not supposed to appear in a classic `Game` instance. This change means that the `get_encountered_players` and `get_encountered_teams` methods of the `Player` model discussed in the sections 5.2 and 5.3 respectively must be updated to take into account the players and teams encountered in team games. In addition, this also requires updating the `TeamDetailSerializer`'s `get_game_count` method to account for unofficial games in a team's game count.

In the implementation of this task, it is crucial to **clearly define how to distinguish a simple game from a team game**. To do this, we can rely on the `first_team` field of the `Game` model. Indeed, if we modify the `CreationScreen` by including an additional tab offering the user who is creating a game to play with one of the teams of which he is captain, then we will be able to determine whether the game is a simple game or a team game before the game is even created. This is extremely important since it is essential to avoid the possibility that there may exist any ambiguity about the game type of a game already created. The creation of a game should therefore allow to directly distinguish between these two types of games. Consequently, if the user creating a game selects one of his teams, then the game will be a team game (with his team as `first_team`). On the other hand, if the user does not select a team, then the `first_team` field will be null and the game will be considered as a simple game. And, of course, whether the `first_team` field is null or not will impact the `GameDetailScreen`.

In the `CreationScreen`, we therefore need to include an additional tab that allows the user to select one of the teams he is captain of. This involves modifying the `fetchTeamList` function in `team/api/api.dart` to take into account as an argument the type of list we want to fetch from the backend. In the present case, this argument would correspond to the `String` "captain" and would indicate to the `TeamListView` in the `account/api_views.py` file that the frontend wishes to access the list of teams of which the current user is captain. This additional `CreationScreen` tab is shown in Figure 28 below.

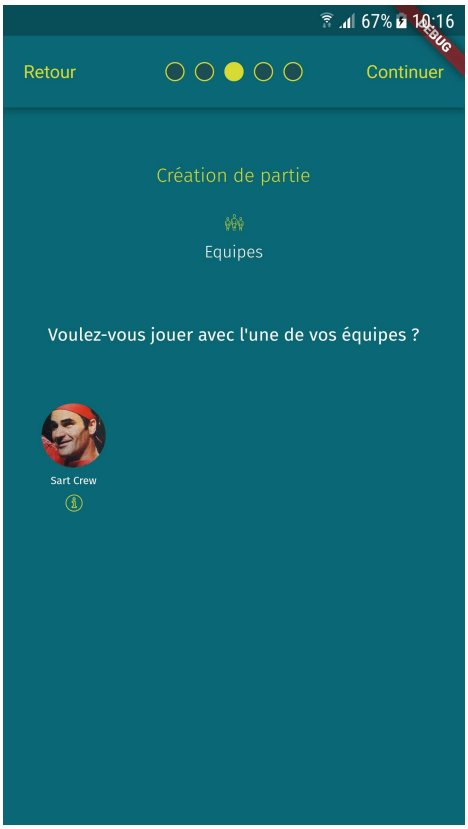
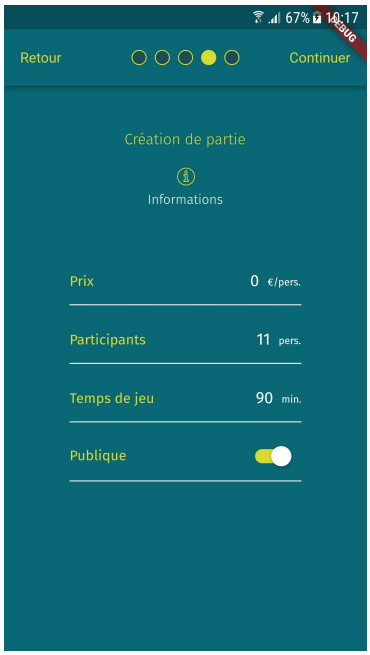
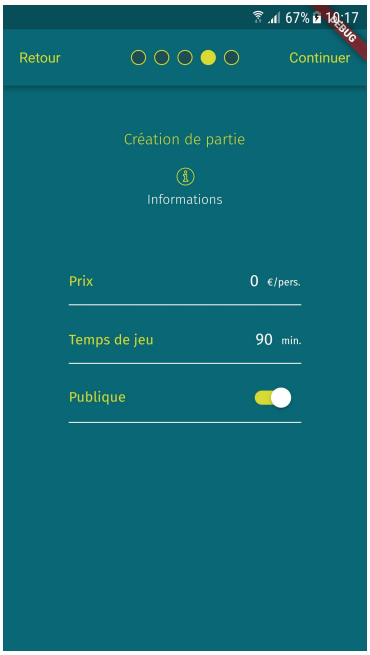


Figure 28: Additional tab in the `CreationScreen` for team games

Moreover, if the user selects one of his teams to create a game, the number of participants shown on the next tab of the `CreationScreen` becomes irrelevant. In fact, in team games, the number of participants is always equal to 2 which corresponds to two teams (no matter how many players these teams contain). We can therefore delete this `InputField` in the case of team games. Figure 29 below illustrates the difference in this tab of the `CreationScreen` between simple games and team games.



(a) Information tab for a simple game



(b) Information tab for a team game

Figure 29: Difference in the `CreationScreen`'s information tab between simple and team games

Once a team game is created, the user is redirected to the corresponding `GameDetailScreen`. On this page, some elements must be changed in the case of a team game. First, the number of available places should no longer be displayed at the top of the screen for the same reason that the number of participants should be removed from the `CreationScreen`. Second, the section at the bottom of the page should no longer display players but rather teams. Figure 30 hereafter illustrates these differences in the `GameDetailScreen` between simple games and team games.

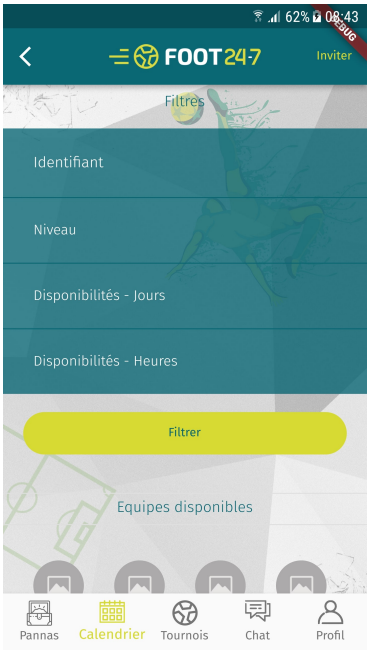
(a) `GameDetailScreen` for a simple game

(b) `GameDetailScreen` for a team game

Figure 30: Differences in the `GameDetailScreen` between simple and team games

Besides, when the creator of a team game wants to add a team to his game by clicking on the "Ajouter" button (see Figure 31a), he must be redirected to a new page allowing him to search for a team to invite instead of a player. To do this, a new file named `game_invite_team.dart` must be defined. This file should define a screen allowing the organizer of the team game to filter the list of teams available on the application according to their name, their level, their availabilities and their schedules as shown in the following Figure 31b.

(a) "Ajouter" button in the `GameDetailScreen`



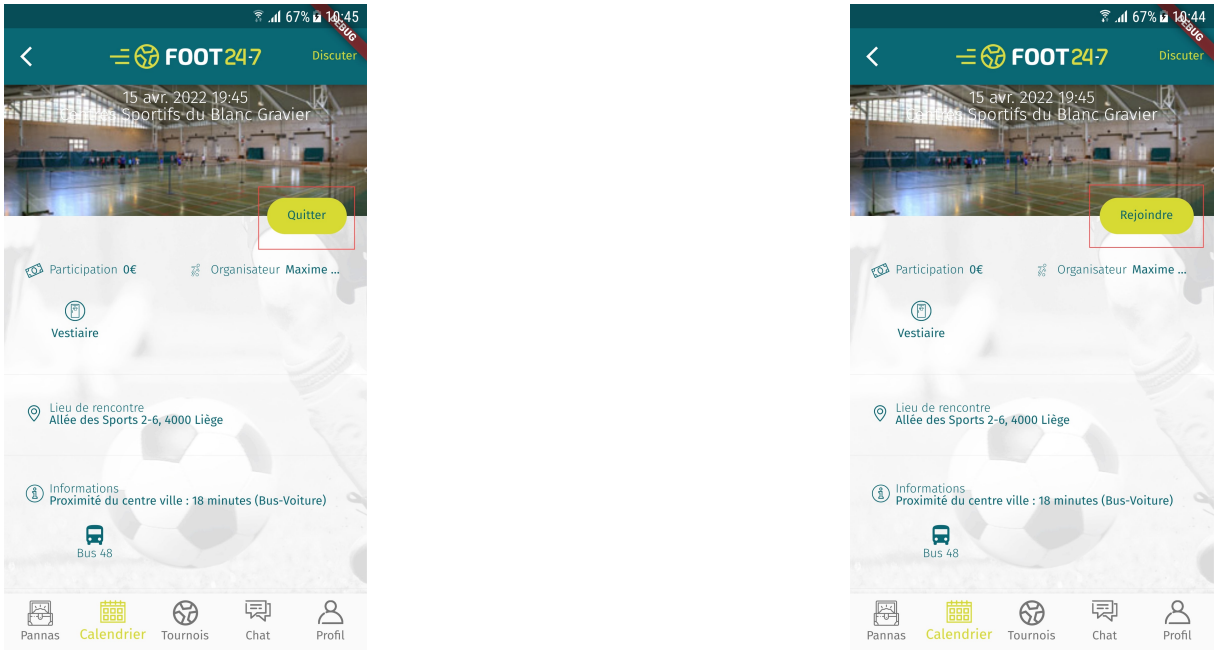
(b) `GameInviteTeamScreen`

Figure 31: Invite a team to a team game

In addition, it is also important to mention that in the case of a team game, this game must appear in the `CalendarScreen` of all players who are members of the participating teams. This thus requires to modify the `CalendarListView` defined in the file `game/api_views.py` in order to take into account team games. Besides, when a team game organizer adds a team to a team game, the captain of the added team should receive an invitation to join this game in his chats if the organizer has first created a chat for this game. However, we will not go into more details about these matters here since we will have the opportunity to discuss them further in the section 5.6.

This task also involves significant modifications to the `GameDetailSerializer` defined in the `game/serializers.py` file. As a matter of fact, most of the methods of this class must be updated to take into account team games.
To begin with, we can mention the `get_can_add` method which allows to display the "Ajouter" button to the creator of a game if there are still places left in this game (see Figure 31a). In the case of team games, it is therefore necessary to check whether there already exists a second team in the game and whether the current user is indeed the creator of the game.
Next, we can mention the `get_can_leave` method which allows to know if the current user can leave the game in question or not, which has an impact on the button displayed at the top of the screen. If he can, one should display a "Quitter" button on the screen (see Figure 32a). To check this in the case of team games, it is necessary to determine whether the user is either the creator of the game or the captain of the second team.
We can also mention the `get_can_join` method which allows to know if the current user can join

the considered game or not, which also has an impact on the button displayed at the top of the screen. If he can, a "Rejoindre" button should be displayed on the screen (see Figure 32b). To check this in the case of team games, it is necessary to verify that the user is captain of a team, that the considered game is public and does not yet include a second team, and that the user is not already present in the first team. The following Figure 32 illustrates the buttons displayed at the top of a game's details page that allow a user to join or leave that game.

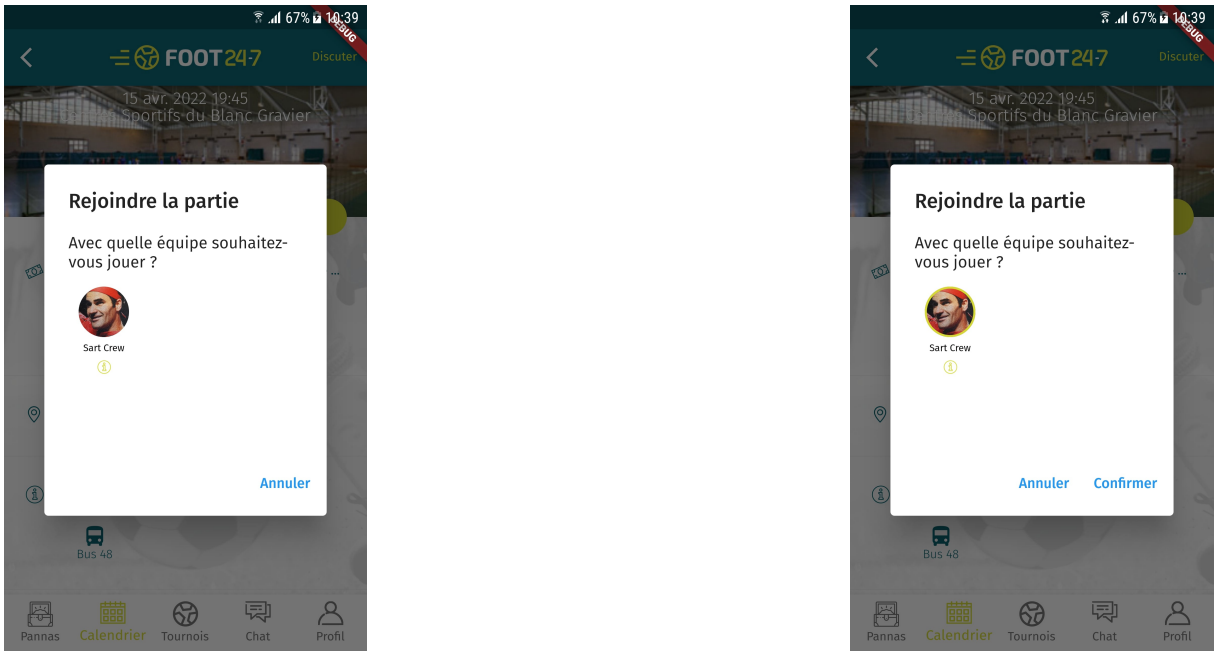


(a) "Quitter" button of the `GameDetailScreen`

(b) "Rejoindre" button of the `GameDetailScreen`

Figure 32: "Quitter" and "Rejoindre" buttons of the `GameDetailScreen`

However, as far as joining a team game is concerned, there is in fact an additional case linked to the game invitation mechanism which we will have the opportunity to discuss in more details in the section 5.6. In any case, it should be noted here that allowing a team captain to join a public team game requires more work than in the case of a simple game. Indeed, in the context of a team game, if a team captain wishes to join this game by clicking on the "Rejoindre" button, one can not simply display a dialog asking for his confirmation but must instead offer him the possibility to choose among the teams of which he is captain. As long as the team captain has not selected one of his teams, he should not be able to confirm his participation in the game. This requires to modify the `_buildTitleButtonWidget` defined in the `GameDetailScreen` in case `_game.canJoin` is true. In this case, it is important to know whether it is a team game or not (by checking if `_game.firstTeam` is null or not as explained above). If it is a team game then a dialog should be displayed asking the user to choose from one of the teams of which he is captain. This team game joining dialog is defined by the `_showChooseTeamDialog` function defined in the `GameDetailScreen` and is depicted in the Figure 33 below.



(a) Before selecting a team

(b) After selecting a team

Figure 33: Team game joining dialog

Of course, all these changes in the `GameDetailSerializer` also require defining corresponding functions in the `game/api/api.dart` file (`joinTeamGame`, `leaveTeamGame`, etc.) as well as defining the associated views and serializers in the `game/api_views.py` and `game/serializers.py` files (`GameJoinTeamView`, `GameLeaveTeamView`, `GameJoinTeamSerializer`, etc.).
Finally, it is also important to recall the notification system discussed in the sections 5.2 and 5.3 which allows users to rate other participants of completed games. In the case of team games, it is essential that the `GameParticipantListScreen` displays not only the teams that have participated in the game but also the players that make up those teams as they can also be rated. To do this, the `GameDetailSerializer` must be updated so that it can return the list of players participating in the game even in the case of a team game (see the `get_players` method). In addition, in the case of team games, the `send_notifications` function defined in the `web/main/management/commands/send_notifications.py` file must also be updated so that all players of both teams receive a notification once the team game is over.

As a final remark, we can also specify that in the case of a team game, only the team captains can communicate via the chat associated with that game. Therefore, if a random member of either team wishes to chat by pressing the "Discuter" button in the top right-hand corner of the `GameDetailScreen`, then a special message should be displayed informing him that only team captains can communicate, as shown in the following Figure 34. In order to differentiate a player who is participating in a team game but is not a captain of one of the two teams from a player who is simply not participating in the game, a `get_is_team_player` method can be defined in the `GameDetailSerializer`. This will allow to display a different message to players who are completely unrelated to the game, specifying that only participants of a game can access the associated chat.

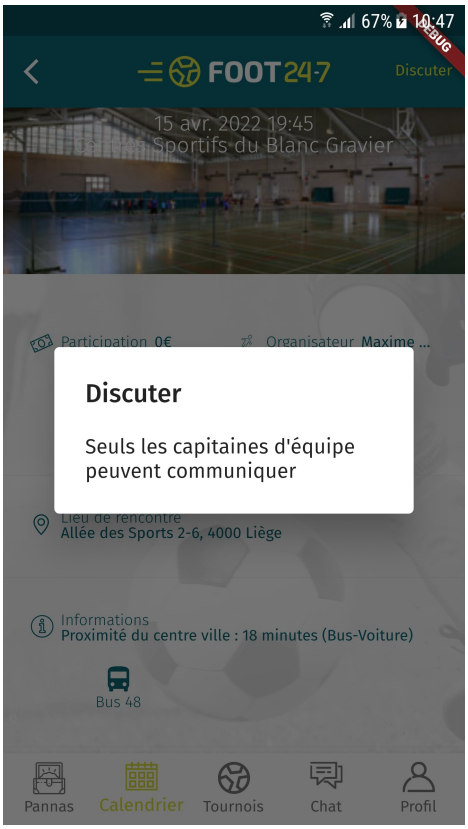


Figure 34: Chat restricted to team captains alert dialog

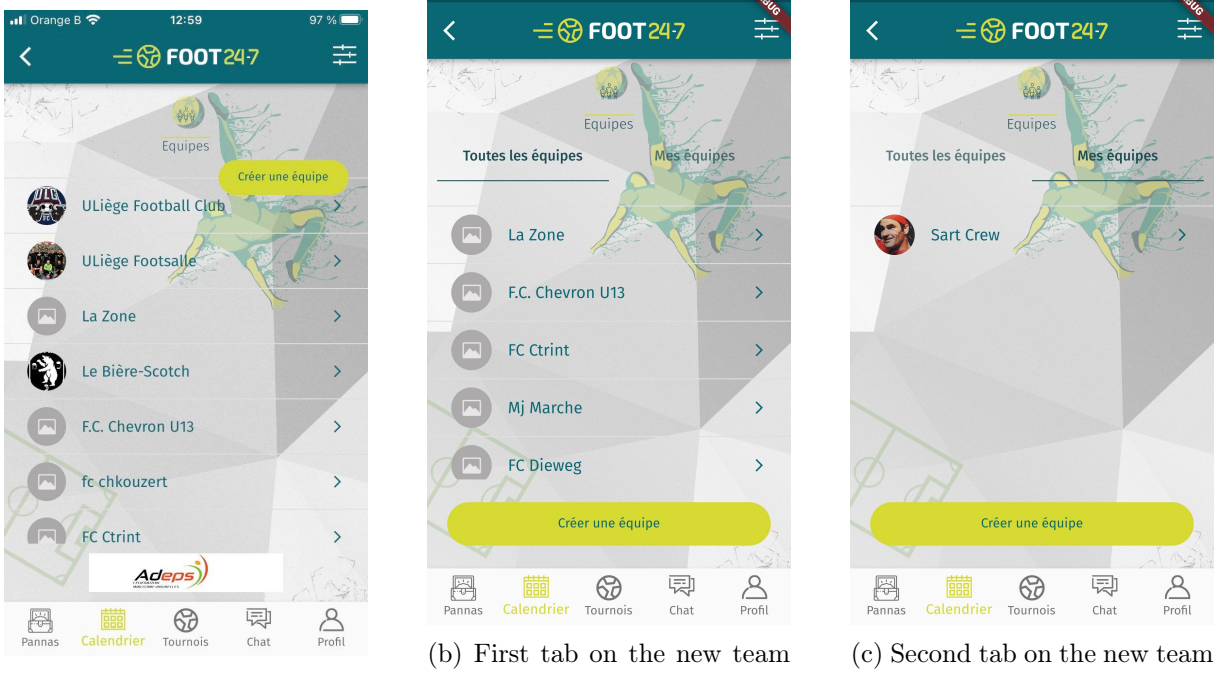
5.5 Access to personal teams

5.5.1 Description

As I was discovering the Foot 24-7 mobile application, I realised that there was no quick and easy way for a user to access his personal teams in the original application, which is not user-friendly. Indeed, it seems obvious that there should exist a **simple way for a user to access the teams to which he belongs**. I therefore proposed to add this feature to Foot 24-7 who then approved the idea.
In order to integrate this new feature in the best possible way into the existing application, the screen listing the teams of the application that is accessible from the search screen (see Figure 8) should be split into two tabs: one tab containing all the teams available on the application and one tab containing all the teams to which the current user belongs.

5.5.2 Implementation & Results

This functionality is relatively straightforward to integrate into the existing application. To do this, one needs to modify the `team_list.dart` file so as to add a tab bar containing two tabs named "Toutes les équipes" and "Mes équipes" respectively. The views associated with these tabs must be populated using the `fetchTeamList` function defined in the `team/api/api.dart` file. To achieve this, it is thus important to provide the type of list that we wish to retrieve from the backend as an argument to this function. If we specify as type a `String` "all" then we will retrieve all the teams registered on the application whereas if we use as type a `String` "me" then we will retrieve all the teams to which the current user belongs. Of course, this also requires to modify the `TeamListView` of the `account/api_views.py` file in order to take into account the requested list type in the construction of the team list. The new team screen and the old one are both depicted in the following Figure 37.



(a) Old team list screen

(b) First tab on the new team list screen

(c) Second tab on the new team list screen

Figure 35: Team list screen update

5.6 Redesigning the invitation mechanism

5.6.1 Description

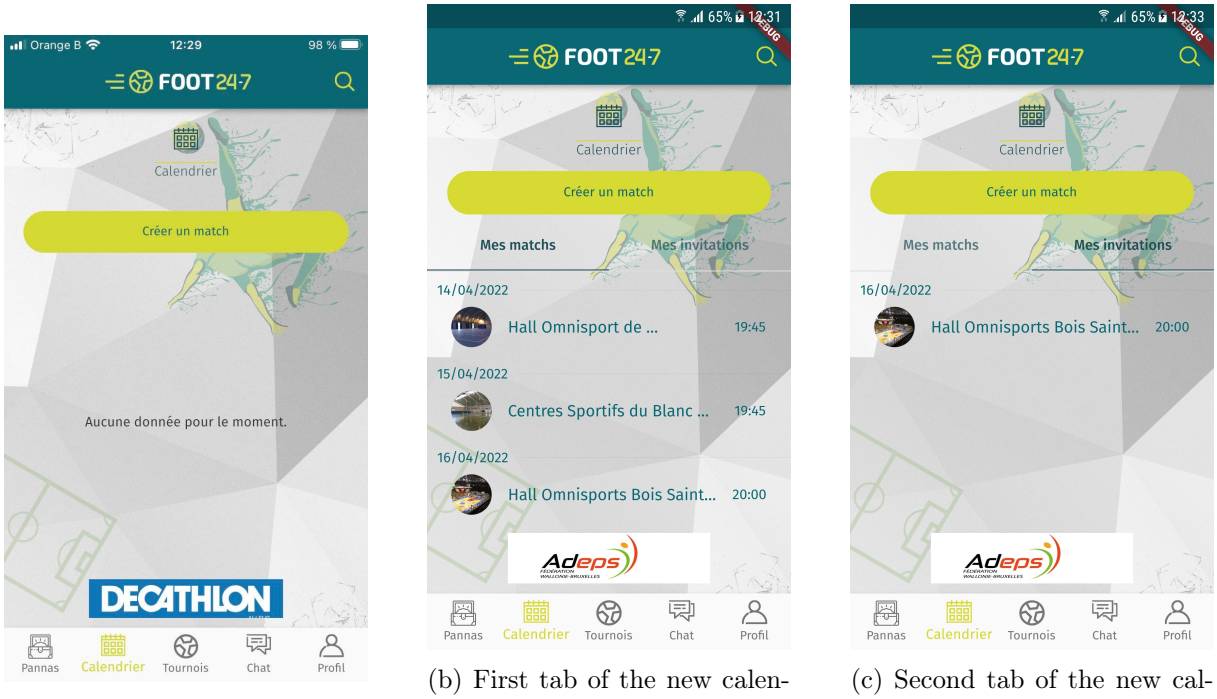
One of the biggest issues I discovered when familiarising myself with the code and the Foot 24-7 mobile application relates to the invitation system used on the application to invite players to join either games or teams. As it happens, the **original invitation system was quite unintuitive and presented some inconsistencies**.
As far as joining a game is concerned, in the original application, when a game organizer adds a player to his game (*i.e.* invites this player to join the game), this player seems to be directly included without his agreement. Indeed, if this player consults the list of games (containing all the games taking place in his city) accessible from the search screen (see Figure 8) and clicks on the game in question, he notices that he is part of the list of players of this game and that he can also leave this game by clicking on the "Quitter" button (see Figure 32a) even though he has never heard of this game and simply stumbled upon it by chance. On the other hand, although the player seems to be directly included in the game, this game does not appear in his calendar. This means that **when a game organizer invites a player to a game, that player seems to be directly included in the game but has no way of knowing it**. This kind of situation is obviously not acceptable in an organizational application. The truth is that when a player is added to a game, he is directly part of the players in that game but is **not yet active**. In fact, if we look at the `GamePlayer` model in the `game/models.py` file, we can see that this model linking the `Game` and `Player` models contains an `is_active` field which is set to false when the player has not accepted the invitation to the game and to true when he has accepted it (or when he has decided himself to join a public game without invitation). The reason why the added player can only leave the game and not join it on the `GameDetailScreen` is that the `get_can_join` method defined in the `GameDetailSerializer` only checks if the player is not part of the game's players **without taking into account if he is active or not**. Actually it is possible for the player to join the game he was invited to (and thus become active in that game) but only if he receives and accepts an invitation to that game via his chats. However, for this to happen it is required that the creator of the game has created a chat for this game beforehand. This mechanism is not at all intuitive, especially since there is no suggestion for the game creator to create a chat. If he has no reason to create a chat, then he will not do so and all the players he adds will never be able to understand that they are part of the game and thus to become active in it.
As far as joining a team is concerned, the problem is the same as for joining a game. When a team creator adds a player to his team, this player is directly registered in the team and can only leave this team from the `TeamDetailScreen` (and not join it). Once the player is added to the team, this player is by default inactive (see the `is_active` field of the `TeamPlayer` model linking the `Team` and `Player` models in the `account/models.py` file) but, to become active, this player can only wait to receive an invitation from the team creator in his chats (if the team creator creates a chat for his team) and accept this invitation. This mechanism is no more consistent in the case of team joining than it was in the case of game joining.
Therefore, I suggested to Foot 24-7 to correct and redesign this invitation mechanism in order to make it more user-friendly (and thus improve the user experience). Understanding the inconsistencies in the original invitation system, Foot 24-7 encouraged me to solve this problem.

5.6.2 Implementation & Results

To clarify the game invitation system, a first thing to do would be: when a game creator adds a player to his game, the icon of this player should appear transparently in the `GameDetailScreen` in order to indicate that this player has been invited but has not yet accepted the invitation (*i.e.* is still inactive). This would make it possible to clearly distinguish between active and inactive players in a game. The **icon of an inactive player** would therefore **appear transparently** until that player becomes active. To do this, the `GameDetailSerializer` of the backend would have to include the `is_active` information in its `get_players` method so that the `_buildPlayerWidget` in the `GameDetailScreen` of the frontend could then use an `Opacity` widget whose `opacity` property would vary according to whether the game player is active or not. The following Figure 36 illustrates the difference between active and inactive game player icons.

Figure 36: Difference between active and inactive game player icons

Next, to allow an inactive player to become active, a **new tab called "Mes invitations"** should be added to the calendar page where users would have access to all the games they have been added to but for which they are not yet active. To do this, it is necessary to modify the `calendar/screens/calendar.dart` file so that it contains a tab bar with two tabs named "Mes matchs" and "Mes invitations" respectively. The view associated with the "Mes matchs" tab would be identical to the one that existed previously, *i.e.* containing all the games in which the user is participating and active. To populate the view associated with the other tab, we would then have to define a new function `fetchGameInvitationList` in the `calendar/api/api.dart` file which would correspond to a new view in the backend called `CalendarInvitationListView` (see the `game/api_views.py` file). This view would only return games for which the current user is participating but inactive. The new "Calendrier" tab is shown in comparison to the old one in Figure 37 below.



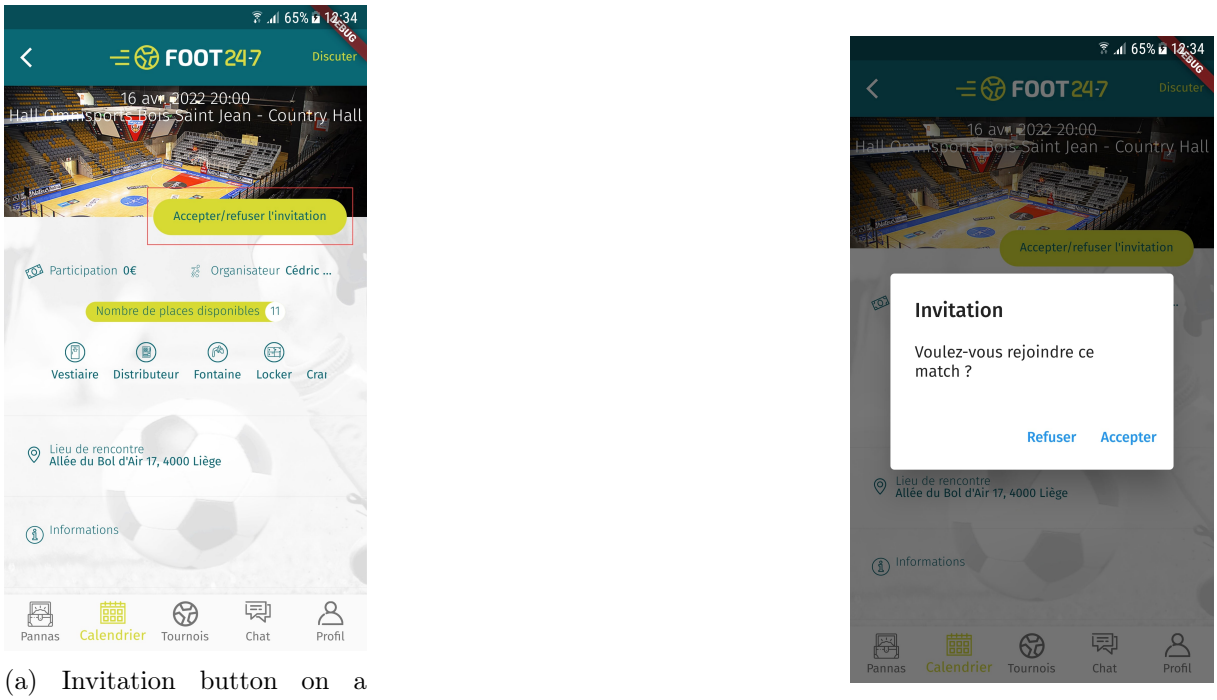
(a) Old calendar page

(b) First tab of the new calendar page

(c) Second tab of the new calendar page

Figure 37: Comparison between the old and the new "Calendrier" tab

In addition, when clicking on a game in the "Mes invitations" tab, the button displayed in the `GameDetailScreen` should specify "Accepter/refuser l'invitation" and when clicked on, would display an invitation dialog on the screen allowing the user to choose to either accept or decline the game invitation. To do this, the `GameDetailSerializer`'s `get_can_join` method would have to be updated to also check if the current user is an inactive player in the game. As a matter of fact, this would allow to check on the `GameDetailScreen` if the player can both join and leave the game. In such a case, this would mean that the button should specify "Accepter/refuser l'invitation" (see the `_buildTitleButtonWidget`). Furthermore, this improved game invitation mechanism could perfectly coexist with the existing chat invitation mechanism. The new `GameDetailScreen` invitation button and the associated invitation dialog are both illustrated in the following Figure 38.



(a) Invitation button on a `GameDetailScreen`

(b) Game invitation dialog

Figure 38: Invitation on a `GameDetailScreen`

The mechanism described above solves the problems related to the invitation system for players to a game. However, as explained in section 5.4, it is also possible to create a game by specifying teams rather than players, which was not the case in the original application. This thus raises the question of how the **game invitation system for teams** should work. As explained in the section 5.4, since a user can only create a team game by specifying one of the teams he is captain of as `first_team`, and a team game can only contain two teams in total, only the second team could be invited (and thus inactive) to a team game. This means that we could therefore add an `is_st_active` field in the `Game` model which would be set to false by default. Similarly to the invitation of a player to a game, when a creator of a team game adds (*i.e.* invites) a team to his game, the added team should be specified in the `second_team` field but should appear transparently in the `GameDetailScreen` until the `is_st_active` field is set to true. To do this, the `GameDetailSerializer` of the backend should include the `is_st_active` field so that the `_buildTeamWidget` in the `GameDetailScreen` of the frontend can then use an `Opacity` widget whose `opacity` property would vary depending on whether the team is active or not. The following Figure 39 illustrates the difference between active and inactive team icons.

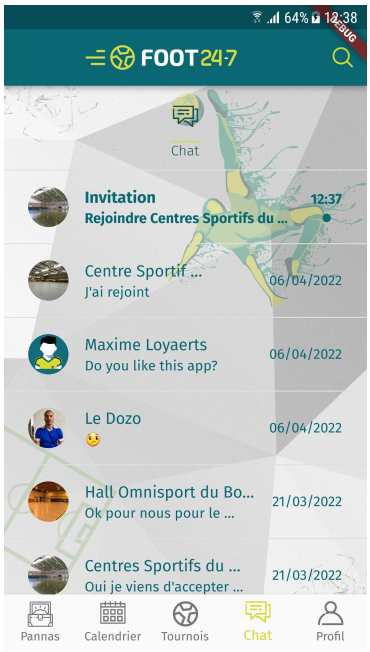
Figure 39: Difference between active and inactive team icons

Invitations to team games would also appear in the "Mes invitations" tab of the calendar page, but only in the calendar page of the invited team captain. In this tab, we should only display team games for which the second team is one of the teams of which the user is the captain and for which `is_st_active` is false. To do so, these games must be taken into account in the `CalendarInvitationListView`. In the "Mes matchs" tab of the calendar page, we should display the team games to all members of the first team (which is always active since its captain created the game) and to all members of the second team if and only if `is_st_active` is true.

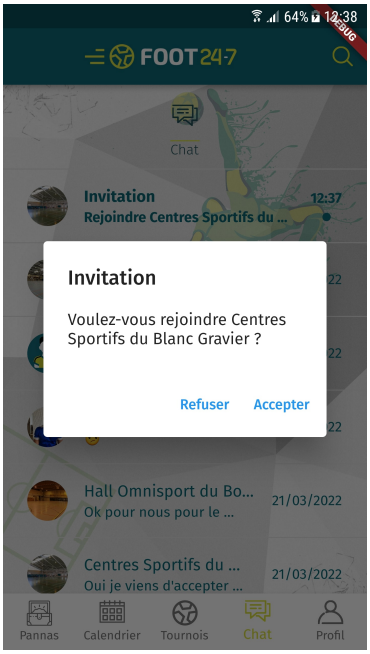
As with simple games, when clicking on a team game in the "Mes invitations" tab, the button displayed in the `GameDetailScreen` should specify "Accepter/refuser l'invitation" and when clicked on, should display the same invitation dialog as the one shown in Figure 38b. To do this, the `GameDetailSerializer`'s `get_can_join` method should be updated to check if the team game contains an inactive second team and that the user is the captain of that team. As before, this would allow to check on the `GameDetailScreen` if the team captain can both join and leave the team game. In such a case, this would mean that the button should specify "Accepter/refuser l'invitation" just as it was the case for simple games. It should be noted here that this means the `_showInviteDialog` method of the `GameDetailScreen` in charge of displaying the invitation dialog shown in Figure 38b must support invitations to both simple and team games.

For the sake of consistency, a team game invitation system should also be included via the chats, as it already exists for simple games. This system should allow the captain of an inactive team in a team game to receive an invitation in his chats if the creator of the game has first created a chat for that game. To do this, the `ChatListSerializer`'s `get_invit_id` method in the `chat/serializers.py` file would have to be updated to take team games into account and return the primary key of a team game associated with a chat if that game contains an inactive second team whose captain is the current user. This `get_invit_id` method allows to differentiate

chats associated with an invitation from regular chats. Furthermore, in order to allow joining a team game via a chat invitation, a new chat type for team games must also be defined in the `get_type` method of the `ChatListSerializer` so that chats associated with team games can be distinguished from other types of chat. We also need to define a new `get_invited_team` method in the `ChatListSerializer` in order to retrieve the primary key of the team invited via a chat. All this information will then be used in the frontend to allow a team captain to join a team game via a chat invitation (see the `_showInviteDialog` method of the `ChatListScreen` in the `chat/screens/chat_list.dart` file). Joining a team game via a chat invitation is illustrated in the following Figure 40.



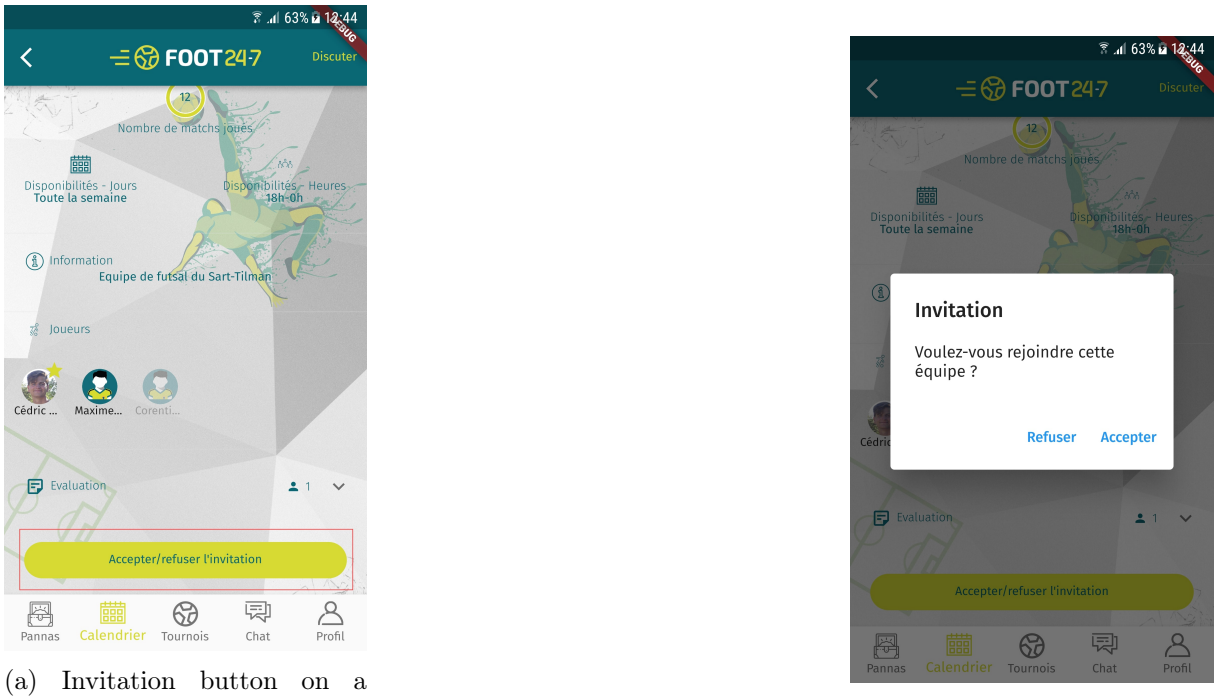
(a) Team game chat invitation



(b) Team game chat invitation dialog

Figure 40: Team game chat invitation system

Finally, the **team invitation system** still needs to be clarified. Since this system basically works in the same way as the game invitation system, it should be modified in a similar way to become more intuitive. To do this, as explained in section 5.5, the user should have access to all the teams in which he appears in the "Mes équipes" tab (see Figure 35c). When clicking on one of these teams, the user is redirected to the `TeamDetailScreen` (see the `team/screens/team_detail.dart` file) and if the user is not yet active in this team, the button displayed at the bottom of the page should specify "Accepter/refuser l'invitation" and not "Quitter" as it is the case in the original application. However, joining a team slightly differs from joining a game. Actually, since a team is not a public object, players can not try to join a team if they were not added by the team creator. This is why there is no "Rejoindre" button on the `TeamDetailScreen` (the button can only specify "Supprimer" for the team creator or "Quitter" for all the other team players) and neither is there a `get_can_join` method in the `TeamDetailSerializer` (see the `account/serializers.py` file). This means that one should add a `get_can_join` method in the `TeamDetailSerializer` that would only check if the user is an inactive team player of this team. If the user can join the team, then the button at the bottom of the `TeamDetailScreen` should display "Accepter/refuser l'invitation" as shown in Figure 41a. The `TeamDetailScreen` invitation button and the associated invitation dialog are both illustrated in the following Figure 41.



(a) Invitation button on a `TeamDetailScreen`

(b) Team invitation dialog

Figure 41: Invitation on a `TeamDetailScreen`

Moreover, as with game invitations, the icon of an inactive team player should appear transparently in the `TeamDetailScreen` until that player becomes active. To do this, the `TeamDetailSerializer` of the backend should include the `is_active` information in its `get_players` method so that the `_buildPlayerWidget` in the `TeamDetailScreen` of the frontend could then use an `Opacity` widget whose `opacity` property would vary depending on whether the team player is active or not, much like what was explained earlier. The following Figure 42 illustrates the difference between active and inactive team player icons.

Figure 42: Difference between active and inactive team player icons

As a final remark, it is important to highlight the fact that **the redesign of the invitation system affects the whole application**. Indeed, once the invitation system redesigned, we still have to change the `get_encountered_players` and `get_encountered_teams` methods of the `Player` model (see the `account/models.py` file) to consider only the encountered players and teams that were active in the games played by the user (and where this user was himself active). This also implies modifying the `GameParticipantListScreen` (see the `game/screens/game_participant_list.dart` file) by displaying only the active players and teams of the completed game. Moreover, in the case of team games, the `get_players` method of the `GameDetailSerializer` must also be modified to consider the players of the second team only if this team is active. Finally, only active team players of active teams should be considered as active players of the team game.

5.7 Game editing

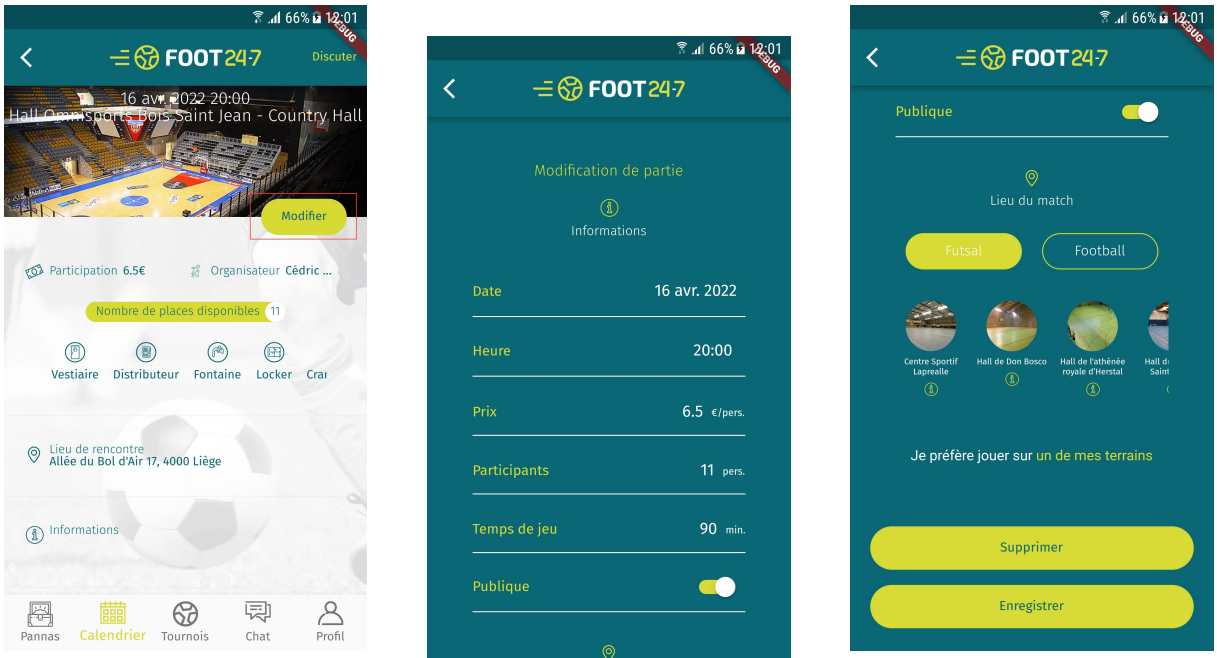
5.7.1 Description

While I was exploring the Foot 24-7 mobile application, I noticed that it was not possible for a user to modify a game that he had created. Consequently, if a game creator wished to modify his game, the only way to do so was to delete the created game in order to recreate a new one with the desired new information. This flaw greatly affects the user experience on the application and should therefore be resolved. Upon exposing the problem to Foot 24-7, they gave me the green light to address the situation.

5.7.2 Implementation & Results

To begin with, it is essential to determine the fields that could be edited in a game. In order to provide the best user experience possible, a game creator should be able to modify as many fields as possible in the game he has created. The retained editable fields are: date, time, price, number of participants, game duration, public or private character of the game and the stadium. Let us remind here that the modification of the teams (in the case of a team game: see section 5.4) or of the players participating in the game is performed directly via the `GameDetailScreen`, `GameInvitePlayerScreen` and `GameInviteTeamScreen` of the `game/screens` folder. Editing the players or teams participating in a game is therefore not part of this task.

In the original application, the button displayed at the top of the `GameDetailScreen` in case the user is the creator of the considered game indicates "Supprimer" and thus only allows to delete this game. To allow the game creator to edit the game, this button should instead specify "Modifier" and when clicked on, redirect the user to a new page (see `GameEditScreen` in the `game/screens/game_edit.dart` file) containing a form to fill in in order to edit the game. This form would thus list all the editable attributes mentioned above. At the bottom of the `GameEditScreen`, a "Supprimer" button allowing to delete the game as well as a "Enregistrer" button allowing to save the game's changes would also be displayed. The following Figure 43 shows the new button on the `GameDetailScreen` of the game creator and the new `GameEditScreen` containing the game edit form.



(a) Game edit button on the `GameDetailScreen`

(b) Game edit page (first part)

(c) Game edit page (second part)

Figure 43: Game editing page

When the game creator reaches the `GameEditScreen`, the form should be initialized and pre-filled with the current attribute values of the game. For this purpose, one needs to retrieve the game details from the backend. This is fairly straightforward except for the `duration` field. In fact, the duration of the game is stored in the `Game` model (see the `game/models.py` file) of the backend using a `DurationField`. In `Django`, a `DurationField` always corresponds to a `String` formatted as "days HH:mm:ss". To display the game duration in minutes in the frontend, one must thus rely on this format in order to correctly convert the game duration into minutes (see the `fromJson` method of the `GameDetail` class in the `game/api/serializer.dart` file).

As regards the actual game edition, one must pay attention to a certain number of issues. First of all, a critical attribute to edit is the price for participating in the game. As a matter of fact, it should not be possible for a user to specify anything other than a positive real number for the game price. This thus requires to check when the user tries to submit the form by pressing the "Enregistrer" button that the specified price is indeed a real number and that this real number is also positive (see the `_isPriceValid` method of the `GameEditScreen`). If this is not the case, the edit form should not be submitted and an appropriate error message should be displayed below the corresponding field.
Another critical attribute is the duration of the game. Indeed, as with the price, it should not be possible for a user to specify anything other than a positive real number for the duration of the game. Again, this requires checking that the duration is indeed a positive real number when the user attempts to submit the edit form (see the `_isDurationValid` method of the `GameEditScreen`). If it is not, the edit form should not be submitted and an appropriate error message should be displayed below the offending field.
Furthermore, the number of participants in the game should only be displayed in the case of a simple game and not a team game (see section 5.4) and is also a critical field to modify. As it happens, the number of participants must first of all be an integer. In addition, it must not be possible for a user to specify a number of participants lower than the actual number of players already present (even inactive) in the game. This thus requires checking these conditions before submitting the form (see the `_isPlayerCountValid` method of the `GameEditScreen`) and

displaying an adequate error message on the screen in case they are not satisfied.
Finally, the last critical field in the edit form relates to the stadium: when editing a game, one must check that the user provided a valid stadium, which is particularly complex because of the possibility of specifying either an official or a custom stadium. The Figure 44 below displays a dynamic diagram detailing the management of a game's stadium modification.

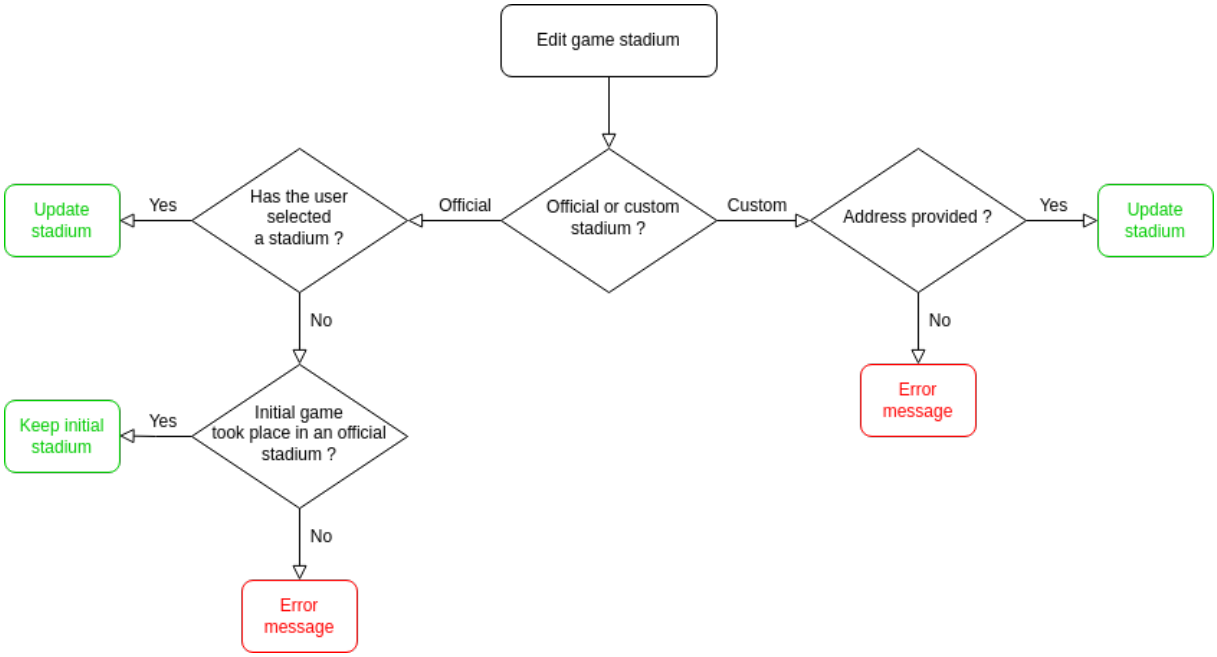


Figure 44: Edit game stadium dynamic diagram

As illustrated in the Figure 44 above, in the case where the user wishes to specify a custom stadium, one must check that the game creator has at least provided an address for this stadium (the "Nom" and "Description" fields are not mandatory). If this is not the case, one should display an error message on the screen preventing the user from submitting the form. On the other hand, in the case where the user wishes to specify an official stadium, if the user does not select any of the stadiums in the list but the initial game was to take place in an official stadium, this stadium should be retained when submitting the form. Indeed, it would be inconvenient for the user to be forced to reselect the stadium of the game if the user had no intention of changing it and simply wanted to change another field in the form. However, if the user wishes to specify an official stadium but does not select one, even though the original game was to take place in a custom stadium, then an error should be displayed on the screen and the form should not be submitable since the game is not assigned to any stadium in such a scenario. To check for all of these cases, an `_isStadiumValid` method can be defined in the `GameEditScreen`.
When it comes to the date and time fields of the game, it is important to realise that the user can only change them through dialogs generated by the `Flutter` functions `showDatePicker` and `showTimePicker` respectively. As these functions incorporate checking that the specified date and time have a valid format, there is no need to check the format of these fields. However, there is still a critical case to address. As a matter of fact, one must check that the specified date and time correspond to a future moment and not to a moment already passed. To do so, we can define an `_isDateValid` method in the `GameEditScreen`. If this is not the case, then an error message should also be displayed in the form.
As we will have the opportunity to discuss and test these critical cases at length in section 6.6, we will only illustrate here the case where a game creator attempts to modify his game by specifying a custom stadium but fails to provide an address for that stadium (see the following Figure 45).

Figure 45: Missing custom stadium address error in game editing form

Once the user has properly filled in the game edit form, he should be able to click on the "Enregistrer" button at the bottom of the `GameEditScreen` in order to save his modifications. To do this, one needs to define a new API function called `updateGame` (see the `game/api/api.dart` file) to send the new game information to a new view called `GameUpdateView` in the backend (see the `game/api_views.py` file) in order to save the changes in the database.
However, for performance reasons, one should check that the game was indeed modified before submitting the form. If this is not the case, it would be both inappropriate and inefficient to update the game in the database when this game was actually not modified at all. In other words, the `updateGame` function should not be called if the game creator did not change anything in the form. To this end, one should define an `_hasChanged` method in the `GameEditScreen` which would compare each of the fields in the edit form with their initial value. If at least one of these fields is different from its initial value, this function would return true. Otherwise, it would return false and clicking on the "Enregistrer" button would simply redirect the user to the `GameDetailScreen` without calling the `updateGame` function.

Finally, if a game creator modifies one of his games, all participants in that game should receive a notification stating that the game was modified and redirecting to the corresponding `GameDetailScreen`. To do this, one must define a `save` method in the `Game` model. In the case of a simple game, this method should send a notification to all players present (even inactive) in the game. However, this method must also take into account team games (see section 5.4) in order to send notifications to all active members of active teams in the game. This task is therefore also related to the redesign of the invitation mechanism (see section 5.6).

5.8 Improving the bottom navigation bar with notifications

5.8.1 Description

Following discussions with Foot 24-7 and the results of the interviews I conducted (see Appendix A), we realized that many users of the Foot 24-7 mobile application were criticizing it for not notifying them when they had received a new message. When a user receives a new message in his chats, the corresponding chat appears in bold in the "Chat" tab to inform the user that this chat has not yet been read. However, this requires the user to click on his "Chat" tab by himself otherwise he has no way of knowing that he received a new message. This flaw does indeed reduce the quality of the user experience on the application and many users of the application claim that they do not use the application's chats because there is **no alert when a new message is received**. To solve this problem, the bottom navigation bar should display the number of unread chats on the "Chat" tab.
Considering the developments described in section 5.6, it would also be fitting to display the number of pending game invitations on the "Calendrier" tab in order to improve the user experience.

5.8.2 Implementation & Results

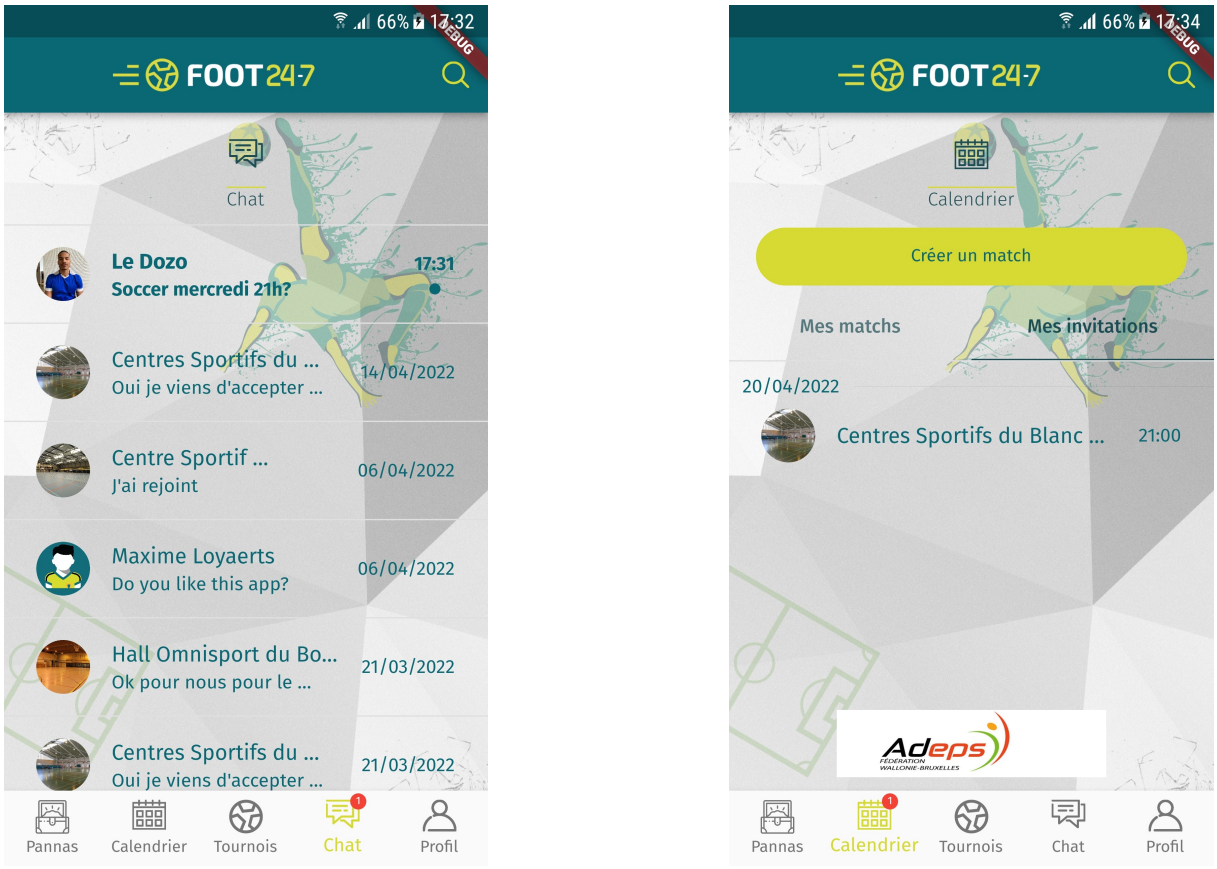
The widget responsible for building the bottom navigation bar is the `BottomNavigation` widget defined in the `widgets/bottom_navigation.dart` file. In the original application, this widget was a simple stateless widget that simply built the bottom navigation bar by highlighting the tab given as an argument to its constructor. In order to enrich the bottom navigation bar with notifications while ensuring a good separation of responsibilities between widgets, this widget should be independent and be able to fully support its own construction. To do so, this widget should be transformed into a stateful widget that would itself take care of retrieving the information required for its construction from the backend. As it happens, in order to include alerts in the bottom navigation bar, we first need to retrieve the number of unread chats of the user as well as the number of pending invitations. To this end, we have to define two new API functions:

- A `fetchNbUnreadChats` function defined in the `chat/api/api.dart` file allowing to return the number of chats containing one or more unread messages for the current user.
- A `fetchNbInvitations` function defined in the `calendar/api/api.dart` file allowing to return the number of pending invitations for the current user.

These two API functions must fetch this information from the backend, which implies defining two new views:

- The `fetchNbUnreadChats` function queries the `ChatUnreadView` defined in the `chat/api_views.py` file, which is responsible for determining the number of chats containing one or more unread messages by the current user.
- The `fetchNbInvitations` function queries the `CalendarInvitationCountView` defined in the `game/api_views.py` file which is responsible for determining the number of pending invitations received by the current user.

When building the bottom navigation bar, we then need to call these two API functions to determine if an alert should be displayed on either the "Chat" or the "Calendar" tab.
The following Figure 46 illustrates the alerts displayed in the bottom navigation bar when a user has received a new message in his chats and when a user has received a new game invitation.



(a) Unread chat notification

(b) Game invitation notification

Figure 46: Notifications in the bottom navigation bar

In the context of this task, it is crucial to **refresh the bottom navigation bar at the appropriate time**. In the case of a notification for an unread chat, this notification should disappear (or the number displayed in the alert should decrease by one if other unread chats exist) from the bottom navigation bar when the user clicks on the unread chat. In the case of a notification for a game invitation, this notification should disappear (or the number displayed in the alert should decrease by one if there are other pending invitations) from the bottom navigation bar when the user responds (either accepts or declines) to this invitation. For this second case, I chose to decrease the notification count only when the user answers the invitation (and not when he first consults it) in order to stimulate users to quickly reply and therefore not to forget their pending invitations. This allows to **maximise the users' reactivity** when organising games via the application and thus to improve the overall user experience. To allow refreshing the bottom navigation bar, one must define an `onRefresh` method in the `BottomNavigation` widget. This method would refetch the number of unread chats as well as the number of pending invitations.

As stated above, when a user clicks on an unread chat, the bottom navigation bar should be updated and decrease the number of unread chats displayed by one. To achieve this, we need to define a `GlobalKey` named `_bottomAppBarKey` in the `ChatDetailScreen` (see the `chat/screens/chat_detail.dart` file) to associate it with the `BottomNavigation` widget. Next, we need to define an `addPostFrameCallBack` method in the `ChatDetailScreen`'s `initState` method to call the `BottomNavigationBar`'s `onRefresh` method via the `GlobalKey` right after the widget is built. As a matter of fact, the chat will only be marked as read once the widget has been completely built, which thus requires to refresh the bottom navigation bar after the widget has been built. In addition, it is also necessary to define a `GlobalKey` associated with the bottom navigation bar in the `ChatListScreen` (see the `chat/screens/chat_list.dart` file) in order to refresh the

bottom navigation bar when the user pops back to the chat list just after reading an unread chat.
Similarly, in order to update the bottom navigation bar once a user responds to a game invitation, one must also define a `GlobalKey` associated with the bottom navigation bar in the `GameDetailScreen` (see the `game/screens/game_detail.dart` file). Once the user answers the invitation (either accepting or declining the invitation), one must call the `onRefresh` method of the `BottomNavigation` widget via the `GlobalKey`.
Finally, as with the `ChatListScreen`, it is also necessary to define a `GlobalKey` associated with the bottom navigation bar in the `CalendarScreen` (see the `calendar/screens/calendar.dart` file) in order to refresh the bottom navigation bar when the user pops back to the calendar after having consulted a game (and thus potentially answered a game invitation).
We will not illustrate these scenarios here as we will have the opportunity to test these use cases extensively in the section 6.7.

5.9 Development of a new referee access

5.9.1 Description

Amongst the tasks suggested by Foot 24-7 (see section 3), Foot 24-7 and I agreed to consider as part of this thesis the addition of a brand new referee access on the Foot 24-7 mobile application. This task consists in developing a referee section on the application where users can offer their services as referees in exchange for payment or free of charge. It is important to realise that this new functionality involves a **major change to the mobile application**. Indeed, since the application was originally designed to accommodate players only, allowing users to log in as referees requires the creation and management of a **new status** on the application as well as to develop a **referee access** well separated from the player access while integrating it as naturally as possible in the original application.
Foot 24-7 wanted it to be possible for a user to have either a player or a referee account on the application. However, Foot 24-7 also requested that a user could have **both types of accounts simultaneously** and easily switch from one to the other via the application. Furthermore, in order to improve the user experience on the application, Foot 24-7 also wished that a user having both types of accounts could always be able to log in using the **same credentials** for both his player and referee accounts.
Once logged in, depending on the user's status, the user would have access to **different user interfaces**. The user interface for player access would obviously not change. As for the new user interface for referee access, this interface should be very similar to the one for player access. As it happens, the same 5 tabs should be preserved in the bottom navigation bar. However, in order to clearly distinguish referee profiles from player profiles, it is necessary to design a new, more formal profile page for referees. This profile page should display the information related to a referee, namely his name, avatar, city, gender, experience, price for refereeing a game, the surface on which he referees and his availabilities. As far as the other tabs are concerned, the user interface should not change much except for a few details such as the fact that the calendar page should not display a "Créer un match" button in case of referee access since a referee should not be allowed to create a game. In any case, even if the user interface does not change drastically, it is still crucial to ensure separate accesses between a player account and a referee account. This means, for example, that if a user has both types of accounts, the chats displayed in the "Chat" tab must necessarily be different depending on whether the user is logged in as a player or as a referee. A referee should be able to create a chat with any user (player or referee) but also to create a chat related to a game he referees. In the case of a chat for a simple game, the referee and all the participating players should be able to access the chat. In the case of a chat for a team game (see section 5.4), only the referee and the team captains should be able to access this chat.
It goes without saying that a game creator should be able to invite a referee to join his game. This thus implies the **addition of a section in a game's details page** displaying the game's referee, whether it is a simple game or a team game. In addition, an invitation mechanism similar to the one described in section 5.6 should also be implemented for referees. The calendar page should therefore retain its two tabs (*i.e.* "Mes matchs" and "Mes invitations") in the case of referee access in order to display both the refereed games and the invitations to referee a game. In the context of this task, an "Arbitres" button should also be added to the search screen (see Figure 8) to allow a player or a referee to browse and filter the list of available referees on the application.
Finally, similarly to the work carried out in sections 5.2 and 5.3, it is also important to add the possibility for a player to rate the referees he has met. This **referee rating system** should include 3 criteria: punctuality, rigour and impartiality.

5.9.2 Solution design process

As mentioned in the previous section, adding a referee section required a major change to the overall functioning of the application. Therefore, before embarking on any developments, it was essential to take the time to assess all the modifications that this new functionality demanded in order to design the best solution to the problem.

To begin with, the application should now not only support the creation of a player account but also the creation of a referee account. Therefore, the account creation page (see Figures 1b and 1c) now has to offer two options to the user: creation of a player account or creation of a referee account. To understand how to create these different types of accounts, we first need to recall the original backend structure. As previously said, in the original application, a user account is represented in the database using the `Account` model (see the `account/models.py` file). However, this model does not allow to represent a player. As it happens, the player entity is represented using another separate model called `Player`. In order to link these two models together, the `Player` model defines a `OneToOneField` called `user` referring to the `Account` model. In the initial application, the creation of an account thus results in the creation of an `Account` instance but also of a `Player` instance as we can see in the `create` method of the `AccountSerializer` defined in the `account/serializers.py` file. In order to integrate a referee entity in the application, it is therefore necessary to create a new `Referee` model in the `account/models.py` file which would also be linked to the `Account` model using a `OneToOneField` called `user`. In addition, the type of account specified by the user on the account creation page must also be included in the `CredentialsCreate` class (see the `user/api/serializer.dart` file) which contains all the information to be sent to the backend in order to create a new account. Once all the authentication information is sent, the `AccountSerializer` must be updated. In fact, depending on the type of account requested, the `AccountSerializer` should associate either a `Player` instance or a `Referee` instance to the newly created `Account` instance.

Now that we have addressed the account creation, we need to consider the login. As mentioned in the previous section, Foot 24-7 wanted it to be possible for a user to hold both types of accounts simultaneously and to switch from one to the other via the application using the same credentials. To solve this problem, there are several solutions but I will only describe here the solution I finally chose. My approach consists in keeping the same login page (see Figure 1a) where the user simply has to provide his email and password to log in without including any information about his status (player or referee). In the initial application, once the email and password have been verified, the user is then redirected to the named route `/` which is also the initial route used when the application is launched (see the `initialRoute` property of the `MaterialApp` widget in the `routes.dart` file). Originally, this route automatically redirects (if the user is logged in) to the profile page of a player. However, in the context of adding a referee access, this default behaviour needs to be redesigned. As a matter of fact, at this stage, it is essential to determine the status of the user in order to know if he is connected as a player or as a referee to redirect him to the proper profile page.
The issue is to decide **where to store** the information about **the current status of the user** and **how to retrieve it when logging in** or when launching the application. Again, at this point, there are several different possible solutions but I will only describe here the chosen solution. My solution is to store the current status of the user in the `Account` model itself via a `last_login_status` variable that would be initialized at the account creation time according to the account type chosen by the user. In order to make this approach scalable, this variable should be of type `String` so that it could be extended to any type of entity. Indeed, there already exist other types of entities in the backend (*e.g.* championship manager, delegate referee). If one day one wishes to add a separate access for these entities on the mobile application, storing the status of the user in a `String` would then allow to easily generalize this approach.

When logging in or launching the application, we then need to retrieve the user's status information to determine to which profile page the user should be redirected. To achieve this, I decided to implement a new widget called `StatusChecker` which would be responsible for fetching the status of the currently logged in user from the backend and redirecting him to the right screen according to his status. To do so, the constructor of this widget would require two arguments: a widget related to the player status and another widget related to the referee status (and we could add more in the future if other types of accesses were needed). Depending on the status retrieved, the `StatusChecker` would then return one of these two widgets. This solution allows to elegantly solve the routing problem. As a matter of fact, by using the `StatusChecker` widget, we can associate as many different screens as we want to the named route `/`.

Now that we are able to determine the status of a user when logging in or launching the application, we now need to decide how to **differentiate the various screens of the application** between player and referee access. As regards the profile screen, since a player's profile and a referee's profile should vary significantly from each other, the current `profile.dart` file (see the `profile/screens` folder) should be replaced by two separate files: `profile_player.dart` and `profile_referee.dart`. In this way, we could avoid ending up with a too heavy `profile.dart` file. When it comes to the other existing screens, there would be no need to define separate files since the user interface for these screens would basically be the same (apart from a few details), only the information fetched from the backend would change depending on the current user status. Separating these screens into two distinct files (one for each type of access) would result in a massive code duplication and should therefore be ruled out.

Moreover, let us recall here that a user should be able to possess both types of accounts simultaneously and switch from one to the other via the application. To allow this, the settings page (see Figure 2b) should include additional buttons allowing a user to either create an account of the kind he does not yet have or, if he has both, to switch to the other type of account. In any case, these buttons should trigger a change in the user's status saved in the backend.

Finally, when it comes to chats between users, there is still a **critical issue to address**. As it happens, in the original application, the `Chat` model (see the `chat/models.py` file) is associated to users via the `ManyToManyField users`. However, this model does not contain any information about the status of the users involved in the chat. Therefore, even if we are able to determine the current status of the user, when we will try to retrieve the chats of a user in a given status, we will not be able to discriminate his chats according to his specific status targeted when the chat was created. Consequently, the chats of a user would be common to both player and referee access, which is not acceptable. To remedy this, it is imperative to include in a chat between users the status of the involved users. In `Django`, when we need to associate data to the relationship between two models linked via a `ManyToManyField`, we use an intermediate model. This intermediate model is associated to the `ManyToManyField` using the `through` argument to point to the model that will act as an intermediary, as explained in `Django`'s documentation [16]. In the case of chats between users, we will thus have to define an intermediate model called `ChatUser` containing two `ForeignKey` fields (one to the `Account` model and the other to the `Chat` model) and a `CharField` about the user's status. In this way, we will be able to **discriminate the chats of a user according to his status**.

Of course, the integration of this new feature also implies many other changes in the application, but since the major problems have already been identified and a solution to each of these problems has been designed here above, we will not go into more details here and leave the development of these other issues to the next section.

5.9.3 Implementation & Results

In the context of this task, the first thing to do would be to add the `CharField last_login_status` in the `Account` model (see the `account/models.py` file). To avoid any problems with already existing user accounts, this field should have as default value `'player'` as all already existing users necessarily correspond to players.
Then, still in the `account/models.py` file, we have to define the `Referee` model. This model should contain, in addition to a `OneToOneField` with the `Account` model, the name, avatar, city, gender, experience, price (to referee a game), surface (on which to referee), pannas and availabilities of the referee.

Creation of a referee account and profile

As regards account creation, as explained in the previous section, the account creation page should now offer two options to the user: the creation of a player account or the creation of a referee account. To do this, we need to add a `String` containing the desired account type in the `user/screens/register/user_register.dart` file. This information should then be included in a new `accountType` field of the `CredentialsCreate` class (see the `user/api/serializer.dart` file). Next, the authentication information is sent to the backend using the `newAccountRegister` API function (see the `user/api/api.dart` file). It is at this point that the `AccountSerializer` defined in the `account/serializers.py` file must distinguish the creation of a player account from a referee account in its `create` method. Actually, as said before, depending on the type of account requested by the user, the `AccountSerializer` should associate either a `Player` instance or a `Referee` instance to the newly created `Account` instance. Finally, once the account creation is complete, the `newAccountRegister` API function should redirect the user to the appropriate profile screen depending on the type of account the user just created. The following Figure 47 illustrates the new account creation page.

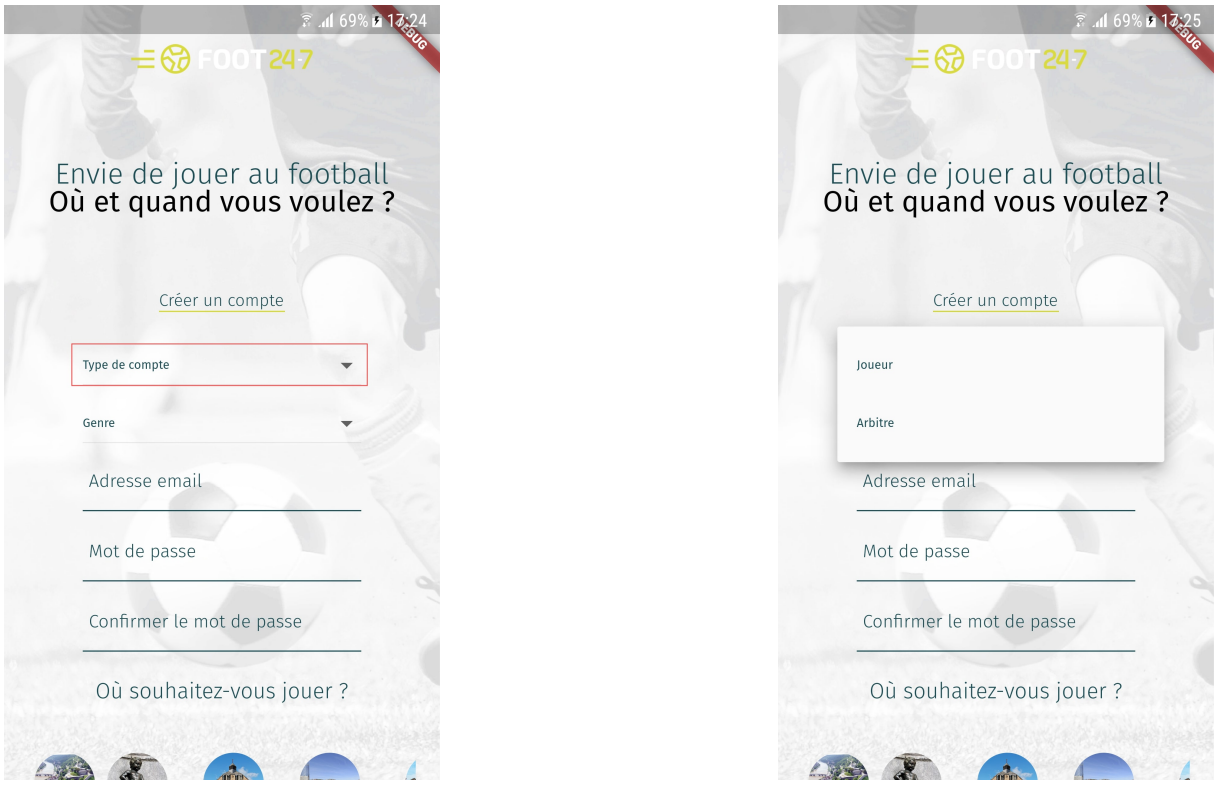


Figure 47: New account creation page

Once a user created a referee account, he should be redirected to a new profile page specifically designed for referees. As mentioned earlier, this involves replacing the `profile.dart` file by two new files: `profile_player.dart` and `profile_referee.dart` which thus also implies a significant renaming phase in both the frontend and the backend. As it happens, the `ProfileScreen` should now be called `PlayerProfileScreen`, routes to a player profile should add a `'/player'` part, API functions like `fetchProfile` should be renamed `fetchPlayerProfile`, views like `ProfileDetailView` to `PlayerProfileDetailView`, etc.
Next, we need to define a new `RefereeProfile` class in the `profile/api/serializer.dart` file and create a new API function `fetchRefereeProfile` in the `profile/api/api.dart` file. This API function should be linked to the backend using new urls (defined in the `account/api_urls.py` file) which must be associated to new views in the `account/api_views.py` file: `RefereeProfileDetailView` and `VisitRefereeProfileDetailView`. Moreover, the corresponding serializers `RefereeProfileDetailSerializer` and `VisitRefereeProfileDetailSerializer` must also be defined in the `account/serializers.py` file.
Once all of this is done, we still have to implement the new `profile_referee.dart` file which will be responsible for building the profile page of a referee. The new profile page for a referee is depicted in the following Figure 48.

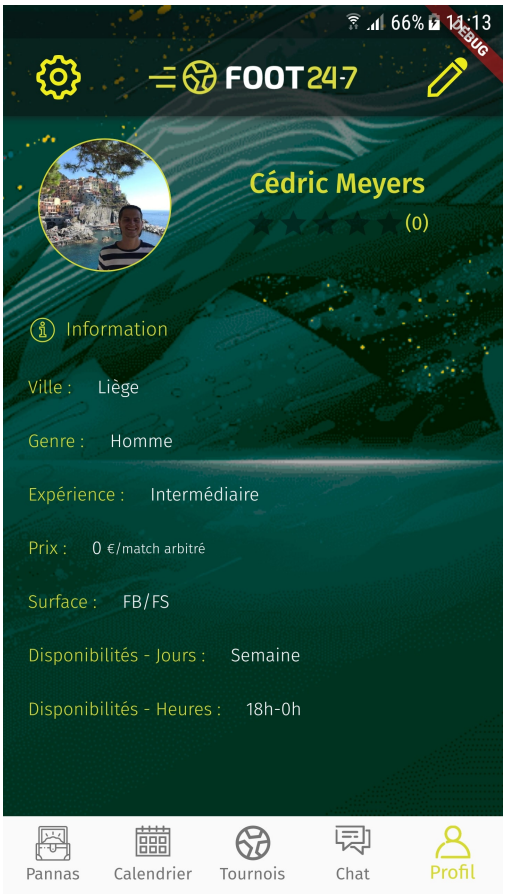


Figure 48: New profile page for a referee

Handling user login and switching between the two types of account

Managing the user login according to his status as well as the switching between the two types of account is illustrated as a dynamic diagram in the following Figure 49.

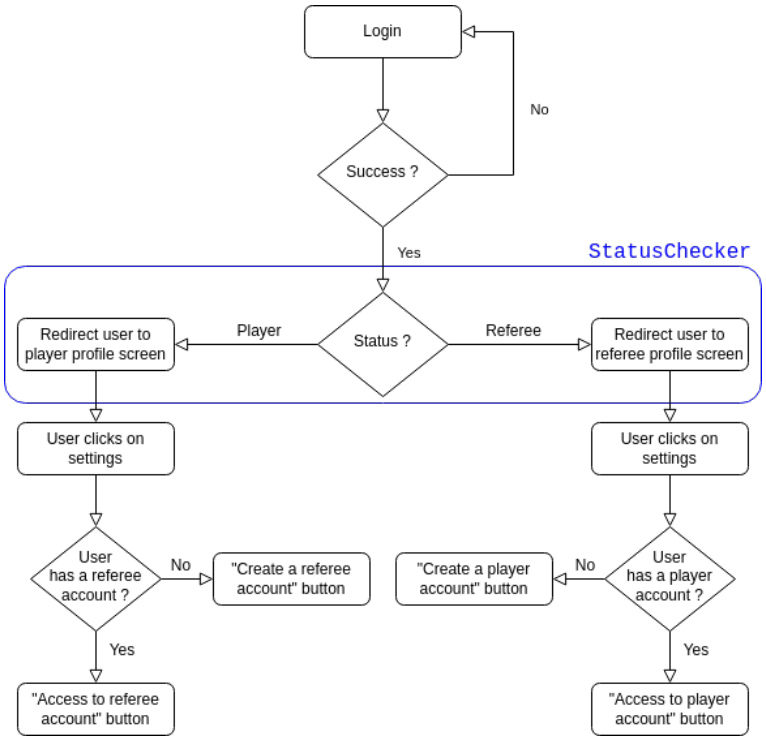
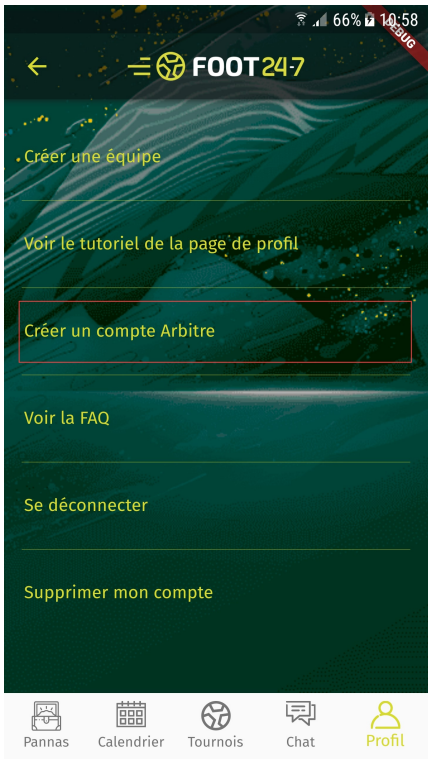


Figure 49: User login and switching between accounts dynamic diagram

As explained in the previous section, we need to implement a `StatusChecker` widget which will allow to retrieve the current status of the logged in user and thus to redirect him to the proper screen according to his status (see the `widgets/status_checker.dart` file). To this end, we have to implement a new API function `fetchCurrentUserStatus` in the `user/api/api.dart` file. This API function should query the backend at a new endpoint defined in the `account/api_urls.py` file. This thus also involves defining a new view `UserCurrentStatusView` in the `account/api_views.py` file. Next, we have to use this `StatusChecker` in the `routes.dart` file in order to match the appropriate screen to the initial route `/` according to the current status of the logged in user.

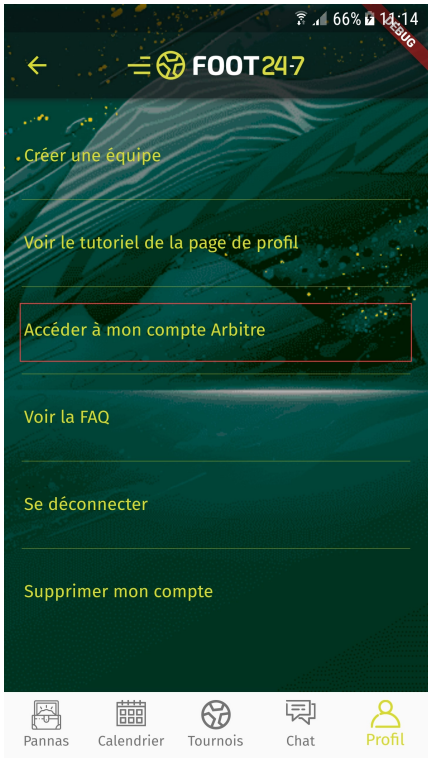
Now that a user is able to create a referee account and is redirected to the proper profile page depending on his status, we need to offer the user the possibility to either create an account of the kind he does not yet have or, if he already has both types of accounts, to switch to the other type of account. As explained in the previous section, to do this, we need to modify the settings page so that it includes new buttons for either creating an account or accessing an account. In order to determine whether we should display a button to create an account or a button to access another account, we need to know what types of accounts the current user already has. To this end, we have to define a new API function `fetchUserStatusDetail` in the `user/api/api.dart` file and a new `UserStatusDetail` class in the `user/api/serializer.dart` file. This class should include three fields: a `String` containing the current status of the user, a boolean `isPlayer` (true if the user possesses a player account, otherwise false) and a boolean `isReferee` (true if the user possesses a referee account, otherwise false). As usual, this new API function should query the backend at a new address defined in the `account/api_urls.py` file which also implies defining a new view `UserStatusDetailView` in the `account/api_views.py` file and a new serializer `UserStatusDetailSerializer` in the `account/serializers.py` file.
Besides, one should also pay attention to the fact that, when a user is in referee status, he should not have access to the buttons "Créer une équipe" and "Voir le tutoriel de la page de profil" in the settings page. Indeed, a referee should not be allowed to create a team and a referee's profile page does not require a tutorial.

The following Figure 50 illustrates the new settings page along with the various buttons offered to the user in the different possible scenarios.



(a) Create a referee account

(b) Create a player account

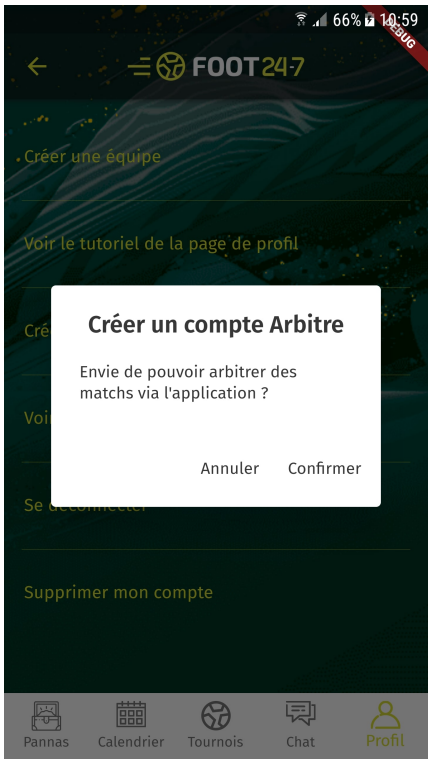


(c) Access to the referee account

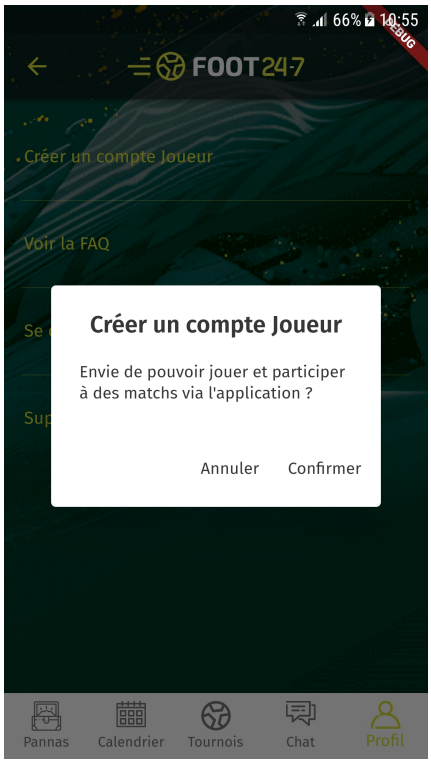
(d) Access to the player account

Figure 50: New settings page

To allow a user to create an account of the kind he does not yet have, a new API function `createAccount` must be defined in the `profile/api/api.dart` file. This function should take as argument a `String accountType` specifying the type of account the user wishes to create. Before calling this function in the `profile/screens/profile_settings.dart` file, the user should first be presented with a confirmation dialog (see the `_showCreateAccountDialog` function), as shown in the following Figure 51.



(a) Referee account



(b) Player account

Figure 51: Create account confirmation dialog

To create a new type of account for the user, the `createAccount` API function must be linked to two new views `CreatePlayerProfileView` and `CreateRefereeProfileView` in the `account/api_views.py` file depending on the type of account requested by the user. This also means defining two new urls in the `account/api_urls.py` file. These views should then create a new instance of the appropriate model (`Player` or `Referee`) linked to the current user account as well as update the `last_login_status` variable in the `Account` model. Once the new account has been created, the user should be redirected to the profile page of his new account. To allow this, the `accountType` specified by the user must be taken into account and we must use the `Navigator.pushNamedAndRemoveUntil` function to ensure that the user can no longer access the pages of the previously used account.

As regards switching between two types of accounts, one must define a `_changeAccount` method in the `profile/screens/profile_settings.dart` file. This method should call a new API function `updateCurrentUserStatus` defined in the `user/api/api.dart` file, whose purpose simply consists in updating the `last_login_status` variable of the `Account` model. To achieve this, this API function must be linked to the `UserCurrentStatusView` in the `account/api_views.py` file. Therefore, we need to define a new `put` method in this class that will take care of updating the last login status of the user. Once the login status has been updated, the user must be redirected to the profile page of the requested account using once again the `Navigator.pushNamedAndRemoveUntil` function.

Editing a referee profile and impact of referee access on gamification

We also must allow a referee to edit his profile. The page allowing a referee to edit his profile should enable him to modify all of his personal information, as depicted in the Figure 52.

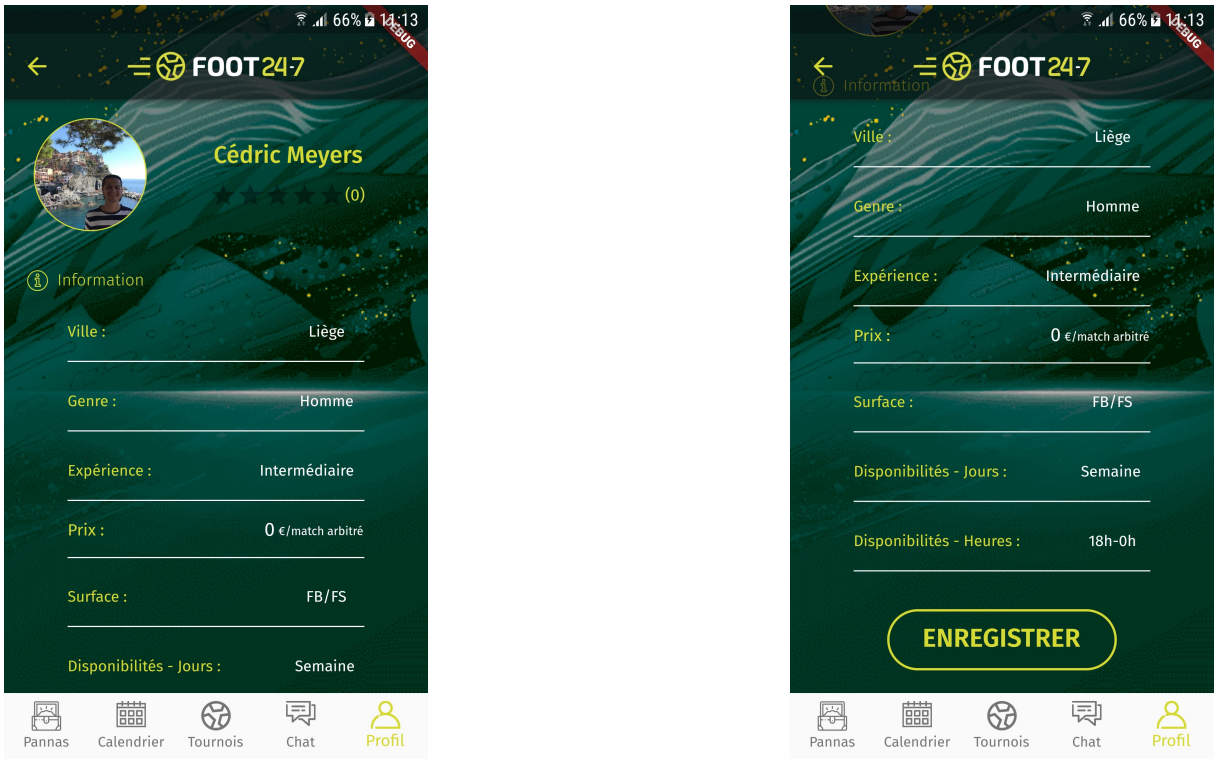


Figure 52: Referee edit profile page

To update a referee's profile, a new API function `updateRefereeProfile` must be defined in the `profile/api/api.dart` file. In the edit referee profile page, the only critical field is the price. Therefore, we must also define an `_isPriceValid` method in the `profile/screens/profile_referee.dart` file and display an error message if the specified price is not valid. The other fields are not critical since the user can only change them through a `_showSelectDialog` method defined in the same file. We will not illustrate here the management of critical cases in the referee profile editing form as we will have the opportunity to discuss it in section 6.8 (see Figure 72).
In the context of editing a referee profile, it is also necessary to define a custom `save` method for the `Referee` model (see the `account/models.py` file). This custom method will allow to check if the user filled in all the information related to his referee account, in which case he should receive pannas (as it is the case when a user completes his player profile). To do this, we need to use the `create_user_achievement` method defined in the `Achievement` model of the `gamification/models.py` file. However, the addition of referee access means that this method now needs to take the user's status as an argument in order to determine which entity (`Player` or `Referee`) should gain additional pannas. Moreover, since a same user can possess both types of accounts simultaneously, we must make sure that there are no interferences between a user's player and a user's referee. This means that we also have to take into account the user's status in the `AchievementHistory` model (see the `gamification/models.py` file). Indeed, an instance of the `AchievementHistory` model corresponds to a specific gain of pannas realized by a certain user for a certain type of achievement. Therefore, this model is notably used to determine if we should grant pannas (or not) to a certain user for a certain type of achievement. If an identical `AchievementHistory` instance (for the same user, the same type of achievement and the same meta information) already exists in the database then the user should not receive additional

pannas. However, the addition of a referee access increases the complexity of this process. To maintain the proper functioning of this feature, we thus have to integrate the user's status in the `AchievementHistory` model. Otherwise, a referee would for instance not earn pannas if he shares a game he has already shared as a player, which does not make sense.
Finally, when a referee completes and saves his profile, we must also display a dialog notifying him of the pannas he just earned, as shown in the Figure 53. To this end, we need to define a `_showUpdateNotification` method in the `profile/screens/profile_referee.dart` file.

Figure 53: Referee profile completion alert dialog

Differentiate a user's chats according to his status

As explained in section 5.9.2, in order to differentiate the chats of a user according to his status in these chats, we need to associate data (*i.e.* the status of the user) to the relationship between the `Chat` model (see the `chat/models.py` file) and the `Account` model (see the `account/models.py` file). This requires to define an intermediate model `ChatUser` in the `chat/models.py` file and to modify the `ManyToManyField users` of the `Chat` model to associate it with this intermediate model using the `through` argument. This new `ChatUser` model must contain two `ForeignKey` fields (one to the `Account` model and the other to the `Chat` model) and most importantly a `CharField` about the user's status. This last field should have as default value `'player'` so as to avoid any problem with already existing chats in the database. As a matter of fact, before the addition of the referee access, all users present in existing chats were necessarily included as players.
However, adding a `through` argument to a `ManyToManyField` is not allowed in `Django`. To get around this problem, we have to trick our minds into elaborating a more complex migration process, such as the one described in [1]. The idea consists in following these steps:

- Create the intermediate model `ChatUser` without associating it to the `ManyToManyField users` via the `through` argument and create a first migration.
- Create an empty migration and edit the created migration file in order to fill the `ChatUser`

model "manually" with the already existing data via a `create_through_relations` function (see the `0015_auto_20220427_1556.py` migration file in the `chat/migrations/` folder).

- Remove the `ManyToManyField users` from the `Chat` model and create a new migration.
- Add a `ManyToManyField users` in the `Chat` model but now containing the `through` argument pointing to the `ChatUser` model and create a new migration.

In this way, we are able to associate the `ManyToManyField users` of the `Chat` model with the `ChatUser` model while avoiding errors during the migration and above all avoiding any data loss during the process.

Still on the subject of chats between users, the addition of a referee access also implies to modify the `ChatGetOrCreateView` (see the `chat/api_views.py` file). As it happens, to check if a chat already exists, we must now take into account the status of the users involved in this chat. If it does not yet exist, when creating the chat, we must then register the users by specifying their status in this chat via the `ChatUser` model.
Next, as previously stated, it is essential that chats linked to a user's player differ from those of the user's referee. To achieve this, we need to modify the `get_user_chats` method of the `Chat` model. In fact, as far as chats between users are concerned, we need to filter these chats by retaining only those where the current user appears with his current status. For other types of chats (game-related, team-related, etc.), the fetching of these chats should be conditioned by the current status of the user. If the user is currently logged in as a player, then we can keep things as they are. If he is currently logged in as a referee, then we need to retrieve game chats of games he referees as well as tournament game chats of tournament games he referees. Indeed, the whole point of adding a referee access to the Foot 24-7 mobile application is to allow users to offer their services to referee games (whether they are simple games or tournament games). Therefore, we must modify the `Game` and `TournamentGame` models defined in the `game/models.py` file by adding a `ForeignKey` field called `referee` pointing to the `Referee` model. In addition, since simple games can be created directly via the application (which is not the case for tournament games), we also have to develop a game invitation system for referees, such as the one described in section 5.6. This is why we also need to add a `BooleanField is_referee_active` (with a default value set to false) in the `Game` model.

In the context of chats, we must also pay attention to the `get_avatar` method of the `ChatListSerializer` (see the `chat/serializers.py` file) and to the `get_name` method of the `Chat` model. As it happens, regarding the `get_avatar` method, in the case of a chat between users, we must check the status of the other user in order to get the right avatar. As for the `get_name` method, the same applies: in the case of a chat between users, one should check the status of the other user in order to retrieve the correct name.
Moreover, adding a referee access on the application also requires integrating the user status in the `Message` model (see the `chat/models.py` file). In fact, without this information, we are unable to determine which name and avatar to display in a chat details page. Again, to avoid any problem with already existing messages in the database, we need to set a default value to `'player'` for this field since all messages sent before adding this feature were necessarily sent by players. This also means that we must now make sure to include the current status of the user when creating a `Message` instance via the `send_message` method of the `Chat` model.
Finally, we must also modify the `MessageListSerializer` defined in the `chat/serializers.py` file. As it happens, in order to take into account the correct name and avatar of the user who sent the message, we have to stop using the `MessageListUserSerializer` (defined in the same file) and instead define a new method `get_user_serialize` in the `MessageListSerializer`. This will then allow us to consider the status of the user in the message in order to retrieve his appropriate name and avatar.

Invitation mechanism for referees

As previously stated, a game invitation mechanism for referees similar to the one described in section 5.6 must be implemented. This means that the "Calendrier" tab should keep its two tabs "Mes matchs" and "Mes invitations" in case the user is logged in as a referee. Therefore, we have to update the `CalendarListView` and `CalendarInvitationListView` of the `game/api_views.py` file to take into account the referee access. If the user is logged in as a player, we can keep things as they are for these two views. As regards the `CalendarListView`, if the user is logged in as a referee, we have to fetch all the games in which he appears and in which he is active as well as all the tournament games in which he appears. For the `CalendarInvitationListView`, if the user is logged in as a referee, then we have to retrieve all the games in which he appears but in which he is not yet active.
Moreover, we must also pay attention to the fact that if the user is logged in as a referee, the calendar page should not include a "Créer un match" button as it does for a player (see Figure 5) since a referee is not allowed to create a game. To ensure this, we need to query the backend using the API function `fetchCurrentUserStatus` to retrieve the current status of the user and then check its value to determine whether to display the button or not.
Furthermore, we also need to update the `CalendarInvitationCountView` in the `game/api_views.py` file by taking into account the current user status. This will allow us to maintain the notifications displayed in the bottom navigation bar (as described in section 5.8) for referees as well. The new calendar page for referees is shown in the following Figure 54.

Figure 54: Calendar page for referees

We can also note that the `GameCalendarListView` of the `game/api_views.py` file which is responsible for generating the list of games displayed after clicking on the "Parties" button of the search screen (see Figure 8) must also be updated in case the user is logged in as a referee. Actually, in this case, the list of games taking place in the referee's city as well as the tournament games he referees must be displayed.

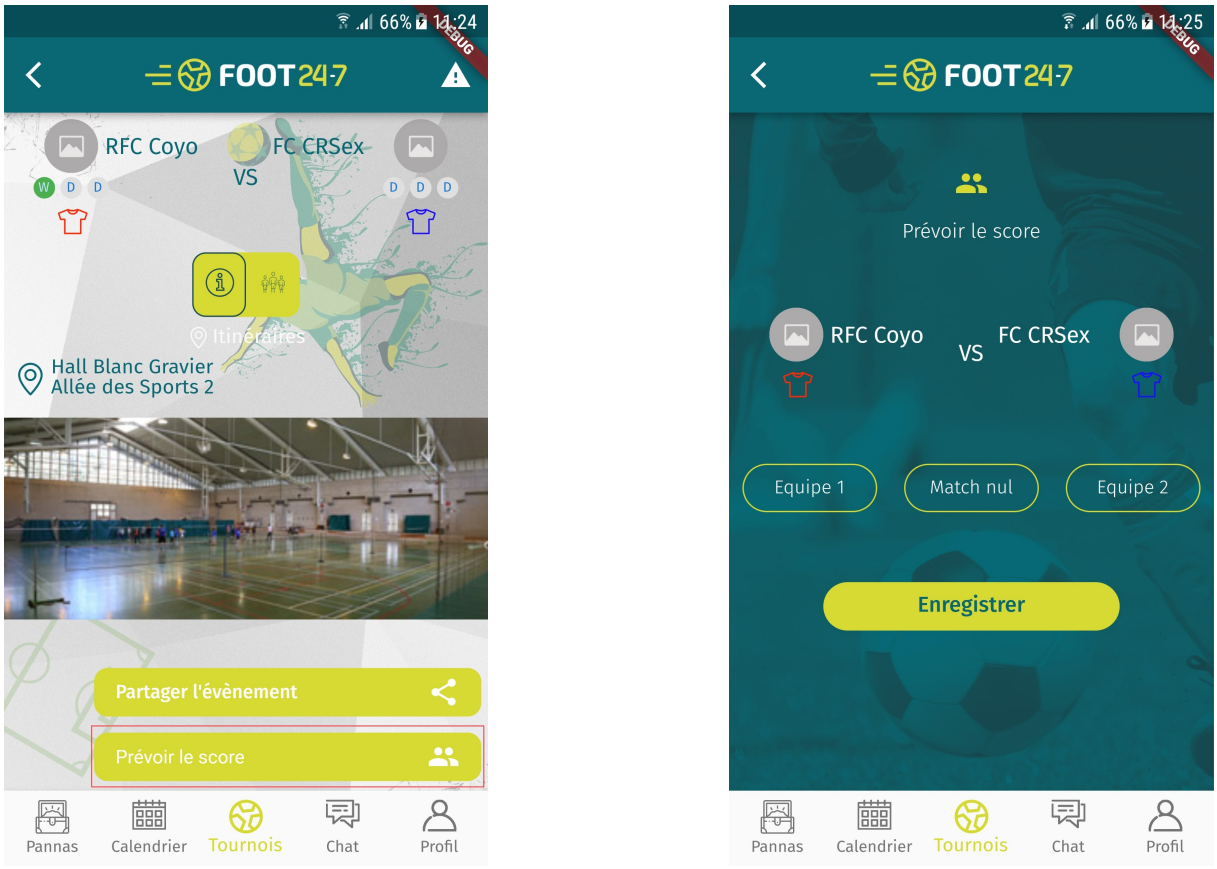
Differentiate a user's pannas according to his status

As regards the "Pannas" tab, the addition of a referee access also requires some adjustments. First of all, we must update the `ProductListScreen` defined in the `gamification/screens/product_list.dart` file and illustrated in Figure 7a. In order to display the correct number of pannas on the screen, we have to determine the current status of the user via the API function `fetchCurrentUserStatus`. Depending on the user's status, we can then retrieve the correct profile (player or referee) with the associated number of pannas.
Next, as regards the `ProductDetailScreen` defined in the `gamification/screens/product_detail.dart` file and depicted in Figure 7b, we have to update the `get_can_exchange` method of the `ProductDetailSerializer` (see the `gamification/serializers.py` file). In order to determine if a user is entitled to request a product exchange, we need to know his current status in order to find out the amount of pannas he owns and thus determine if he has enough to purchase the product requested. In addition, we also have to modify the `ProductExchangeView` (see the `gamification/api_views.py` file). As a matter of fact, we must take into account the current status of the user in order to determine which entity (player or referee) should lose pannas following the purchase of the product. As a bonus, in order to improve the clarity of product exchanges, we can also add a field about the user's status in the `ProductExchange` model (see the `gamification/models.py` file). This allows us to know which entity is actually requesting to exchange a product. Similarly to what was done earlier, we should provide a default value of `'player'` for this field to avoid any problems with existing exchange requests in the database. This change then involves specifying the current status of the user when creating a `ProductExchange` instance in the `update` method of the `ProductExchangeView`.

Updating the tournaments tab to support referees

When it comes to the "Tournois" tab, adding a referee access also requires some modifications. Firstly, we must pay attention to the "Mes matchs" tab of the `TournamentGroupDetailScreen` (see the `tournament/screens/tournament_group_detail.dart` file) which is illustrated in Figure 4d. If the user is logged in as a referee, this tab should display the list of tournament games refereed by that user. To do so, we must take into account the current status of the user in the `TournamentGameListView` (see the `game/api_views.py` file). Next, we must also check the current status of the user in the `TournamentGameDetailSerializer` (see the `game/serializers.py` file). As it happens, if the user is logged in as a referee, the `get_is_captain` method must return false since a team captain can only be a player. Moreover, the `get_players` method of the `TournamentGameDetailSerializer` must also be updated. Indeed, this method specifies an `is_editable` field for each player of a tournament game. This field allows the user corresponding to the considered player or his team captain to indicate the participation of the player in the tournament game. Therefore, if the current user is logged in as a referee, the `is_editable` field should necessarily be false as a referee can neither be a player nor a team captain.

Still in the context of tournament games, it is important to note that the original application offers the possibility for users to make predictions about the outcome of upcoming tournament games, as can be seen in the Figure 55 below. In case of a correct prognosis, the user will be awarded pannas.



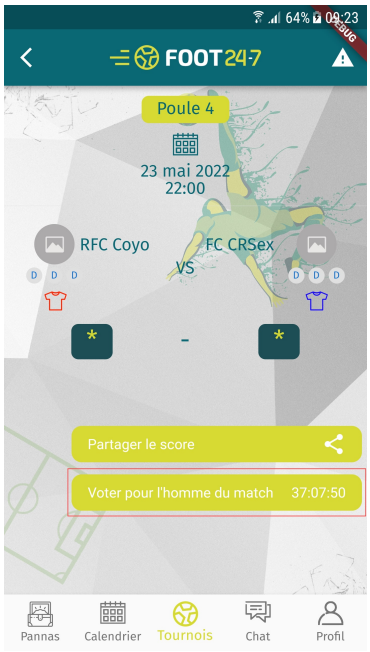
(a) Predict the outcome button

(b) Predict the outcome page

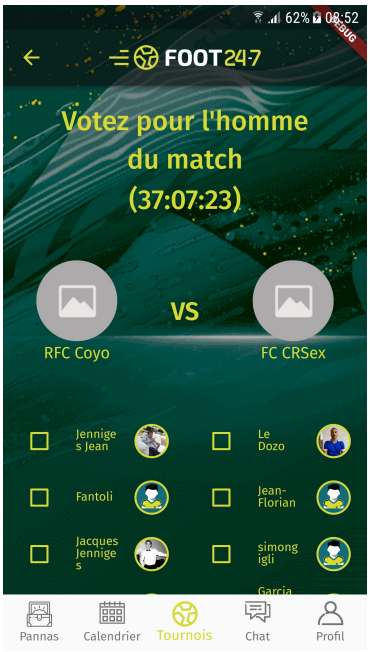
Figure 55: Predict the outcome of a tournament game

However, for obvious reasons, a referee should not be allowed to make prognoses on tournament games. Therefore, we need to remove the "Prévoir le score" button (see Figure 55a) from the `TournamentGameDetailScreen` (see the `tournament/screens/tournament_game_detail.dart` file) if the user is logged in as a referee. To achieve this, we once again need to retrieve the current status of the user using the API function `fetchCurrentUserStatus`. Furthermore, in order to make the frontend and backend as robust as possible independently of each other, we can also check the current status of the user when creating a prognosis in the backend, namely in the `update` method of the `TournamentGamePrognosticSerializer` (see the `game/serializers.py` file). In case the user is connected as a referee, then we should simply not create an instance of the `TournamentGamePrognostic` model (see the `game/models.py` file).

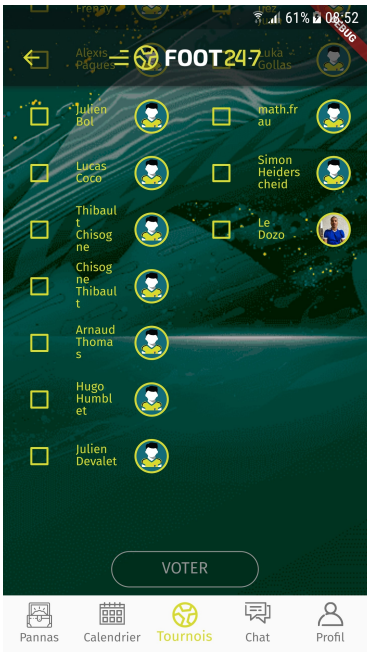
A final change to consider in the context of tournaments relates to the voting for the best player of a tournament game. In fact, in the original application, once a tournament game is over, users can vote for the best player of the tournament game, as illustrated in Figure 56. Once again, voting for the best player of a tournament game allows users to earn pannas.



(a) Vote for the best player button



(b) Vote for the best player page



(c) Vote for the best player page

Figure 56: Vote for the best player of a tournament game

However, as requested by Foot 24-7, a referee should not be allowed to vote for the best player of a tournament game. Therefore, we also need to remove the "Voter pour l'homme du match" button (see Figure 56a) from the `TournamentGameDetailScreen` if the user is logged in as a referee. Similarly to what was done for prognoses, we can also check the current status of the user when creating a vote for a best player in the backend, namely in the `update` method of the `TournamentGameBestPlayerSerializer` (see the `game/serializers.py` file). If the user is connected as a referee, then we just have to avoid creating an instance of the `TournamentGameBestPlayer` model (see the `game/models.py` file).

Adding a referee section in the game's details page

Now that all the main tabs of the application have been updated to support referees, we must allow a game creator to invite a referee to join his game and thus add a referee section in the game details page. The referee section should display the referee of a game and allow a game creator to modify the game referee, which therefore requires modifying the `game/screens/game_detail.dart` file. The new referee section is illustrated in the following Figure 57.

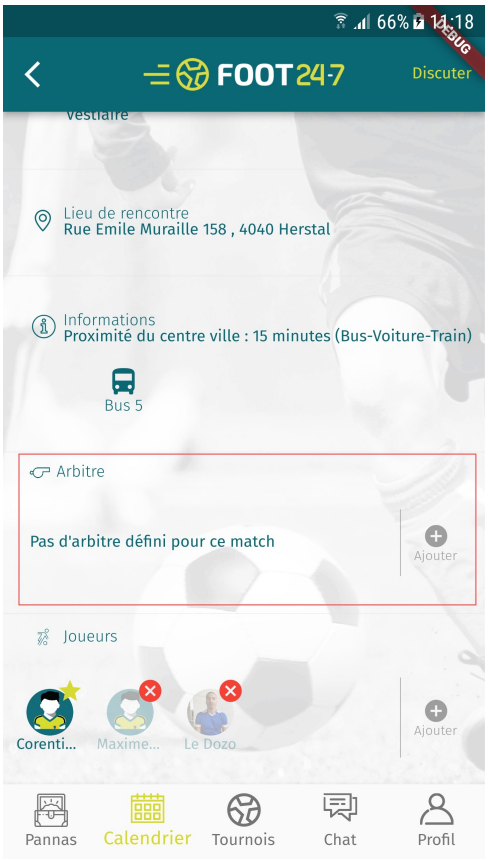


Figure 57: Referee section on the `GameDetailScreen`

To build this referee section, there are several changes and additions to bring to the existing code. First, we must modify the `GameDetail` class (defined in the `game/api/serializer.dart` file) to include information about the referee of the game, namely a `referee` field of type `GameDetailReferee` and a boolean field `isRefereeActive`. This thus also requires to define a new `GameDetailReferee` class in this file. On the backend side, the `GameDetailSerializer` defined in the `game/serializers.py` file must also be modified to provide this information. To this end, we also have to define a new `GameDetailRefereeSerializer` class in the same file.

Next, the "Ajouter" button on the game details page (see Figure 57) allowing a game creator to invite a referee to join his game should redirect the game creator to a new screen which is depicted in Figure 58 below.

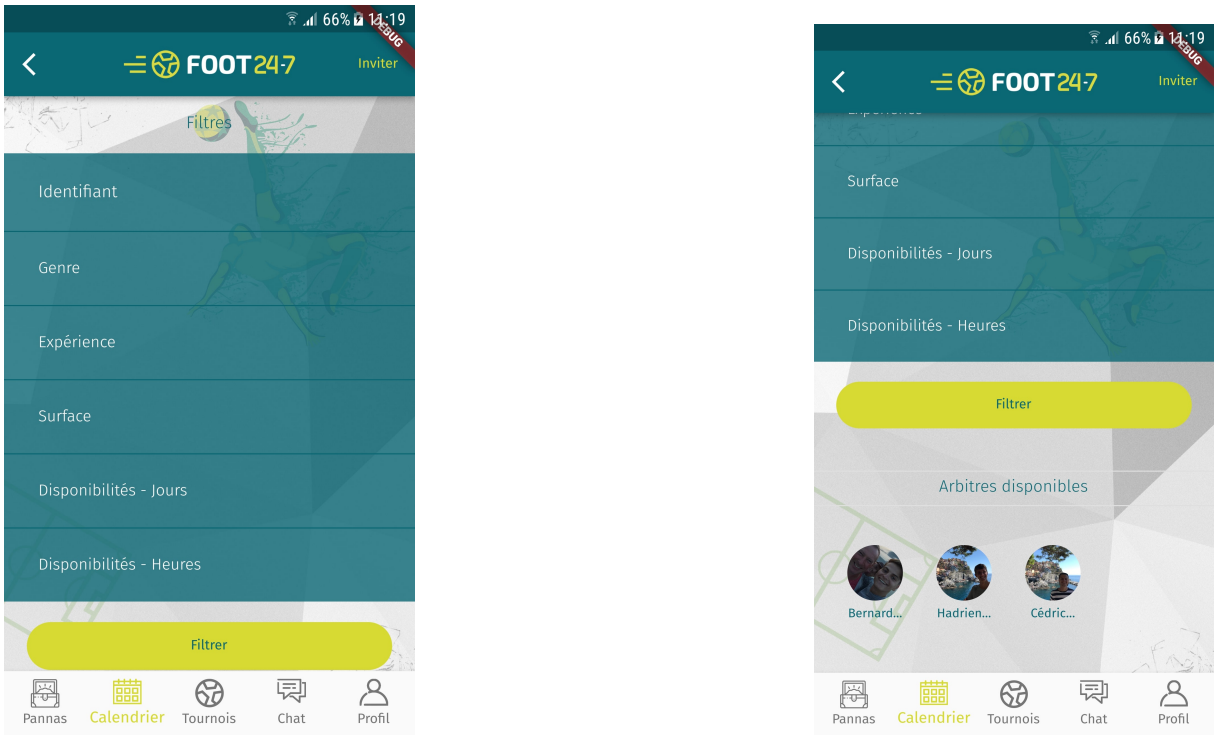


Figure 58: Game invite referee page

This new screen is defined in a new `game_invite_referee.dart` file in the `game/screens/` folder. To generate this screen, we must create a new API function allowing to retrieve the list of referees available on the application as well as new classes allowing to represent referees in the frontend. Therefore, we have to define a new `referee` folder (containing the two classical subfolders `api` and `screens`) in the `mobile_app/lib/` folder. In the `api` subfolder, we have to create a new `api.dart` file. This file must then define a new API function called `fetchRefereeList` allowing to fetch the list of available referees on the application according to the filters requested by the user. This API function shall query the backend at a new endpoint defined in the `account/api_urls.py` file which must correspond to a new `RefereeListView` (see the `account/api_views.py` file). This view should use the filters provided by the API function to filter the referee `QuerySet`. At this stage, it is important to note that the API function `fetchRefereeList` should also add a url parameter `game_invite` containing the primary key of the considered game. This allows to remove from the returned referee list the referees whose users are already involved in this very game as players. As a matter of fact, it is obvious that **a user should not be involved in the same game twice** but with different roles (player and referee). The `RefereeListView` also requires to define a new `RefereeListSerializer` in the `account/serializers.py` file. Furthermore, in order to build a referee list in the frontend, we also need to add a `serializer.dart` file in the `referee/api/` folder. This file should contain the definition of two classes: `RefereeList` and `RefereeListItem`.
Then, we must also allow a game creator to actually invite a referee to a game but also to remove a referee from a game. To do this, we have to define two additional API functions `inviteRefereeGame` and `removeRefereeGame` in the `game/api/api.dart` file. As usual, these two API functions require to define new urls in the `game/api_urls.py` file, to define two new views `GameInviteRefereeView` and `GameRemoveRefereeView` in the `game/api_views.py` file as well as a new serializer `GameInviteRefereeSerializer` in the `game/serializers.py` file.
Let us recall here that, since we want to implement an invitation system for referees similar to the one described in section 5.6, once a referee is invited to a game, he should appear transparently in the referee section so as to indicate that he has not yet accepted the invitation and is therefore not yet active in the game, as illustrated in the following Figure 59.

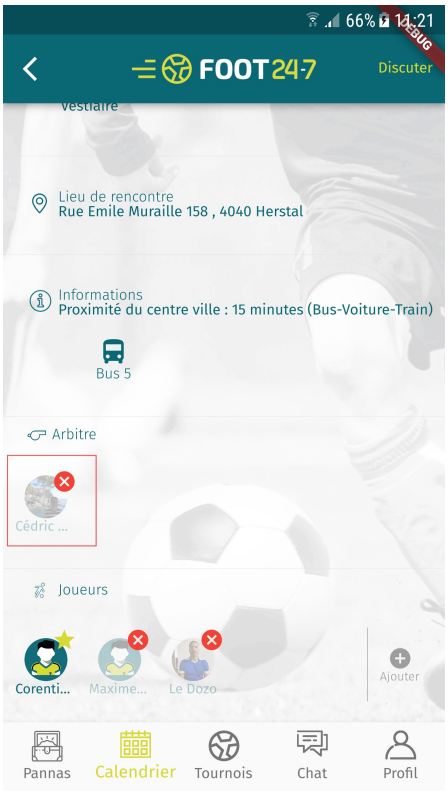


Figure 59: Referee invited to a game

Now that it is possible for a game creator to invite a referee to join his game and that this invitation can then be found in the "Mes invitations" tab of the referee's calendar page (see Figure 54), we must allow a referee to accept or decline a game invitation and, more generally, to join or leave a game. To achieve this, we have to define two new API functions `joinRefereeGame` and `leaveRefereeGame` in the `game/api/api.dart` file. Once again, these API functions involve new urls in the `game/api_urls.py` file as well as new views `GameJoinRefereeView` and `GameLeaveRefereeView` in the `game/api_views.py` file.
Next, we need to make sure that we display the appropriate button to a referee on a game's details page. Depending on the circumstances, the referee should either see a "Rejoindre" button, a "Quitter" button, an "Accepter/refuser l'invitation" button or no button at all. To this end, we have to update the `get_can_join` and `get_can_leave` methods of the `GameDetailSerializer` to support referees. These two methods should now take into account the current status of the user and distinguish between a user logged in as a player and as a referee. As regards the `get_can_join` method, in the case where the user is logged in as a referee, this method should return true if the user is the referee of the game but is not yet active in the game, or if the game is public, has no referee yet, and the user is neither a player of the game (in case of a simple game) nor a member of a team participating in the game (in case of a team game). Otherwise, this method should return false. In case the user is connected as a player, this method must now also check if this user is already involved in the game as a referee in which case it should return false. When it comes to the `get_can_leave` method, in case the user is logged in as a referee, this method should return true only if the user is the current referee of the game (whether active or not). Otherwise, it should return false. In case the user is connected as a player, nothing should change in this method.
While we are discussing the `GameDetailSerializer`, we can also mention that one also needs to modify the `get_can_edit`, `get_can_add`, `get_can_remove` and `get_is_team_player` methods of this serializer. As it happens, these methods should also take into account the current status of the user and return false in case the user is connected as a referee.

Beyond the changes to bring to the `GameDetailSerializer`, adding a referee access also requires updating the way we can add players, teams or referees to a game. Indeed, as said before, a user should not be involved in the same game twice but with different roles (player or referee). Therefore, when adding players to a game, we have to prevent a game creator from inviting a player if the corresponding user is already the referee of this game. To do this, we need to modify the `PlayerListView` (see the `account/api_views.py` file) which is responsible for generating the list of players available on the application according to the filters specified by the user. In the case where the user tries to get the list of players he can add to his game, we have to remove from the returned list the player whose user already corresponds to the referee of the considered game (if he exists). Similarly, when adding a referee to a game, we must prevent a game creator from inviting a referee if the corresponding user is already a player of the game (as said before) or a team player of the game. To this end, we need to modify the `RefereeListView` responsible for generating the list of available referees on the application according to the filters requested by the user in order to remove from the returned referee list the referees whose user matches a player or a team player of the game. Finally, we must also follow a similar reasoning regarding the addition of teams to a game. As a matter of fact, we must prevent a game creator from adding a team containing a player whose user already matches the referee of the game of interest. To do so, we have to modify the `TeamListView` (see the `account/api_views.py` file) responsible for generating the list of teams available on the application according to the filters requested by the user in order to remove from the returned list the teams containing the user corresponding to the game referee.

Allowing users to browse all the available referees

Now that we have included a referee section in a game's details page, we must allow users of the application (whether they are players or referees) to browse the list of referees available on the application. To do so, we have to add an "Arbitres" button to the search screen, as can be seen in Figure 60 below.

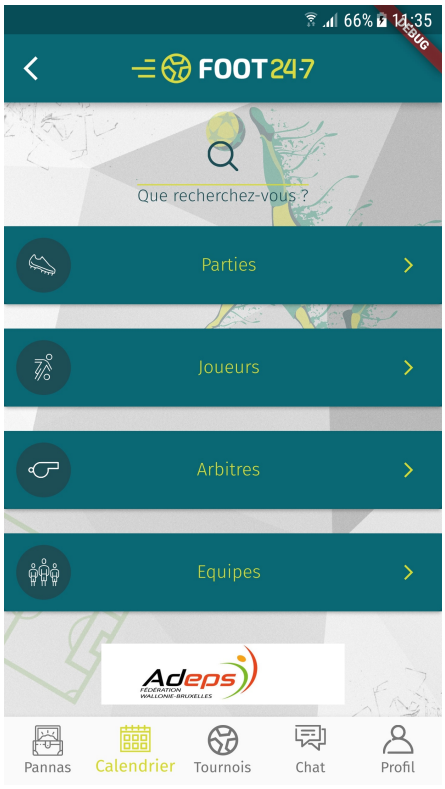
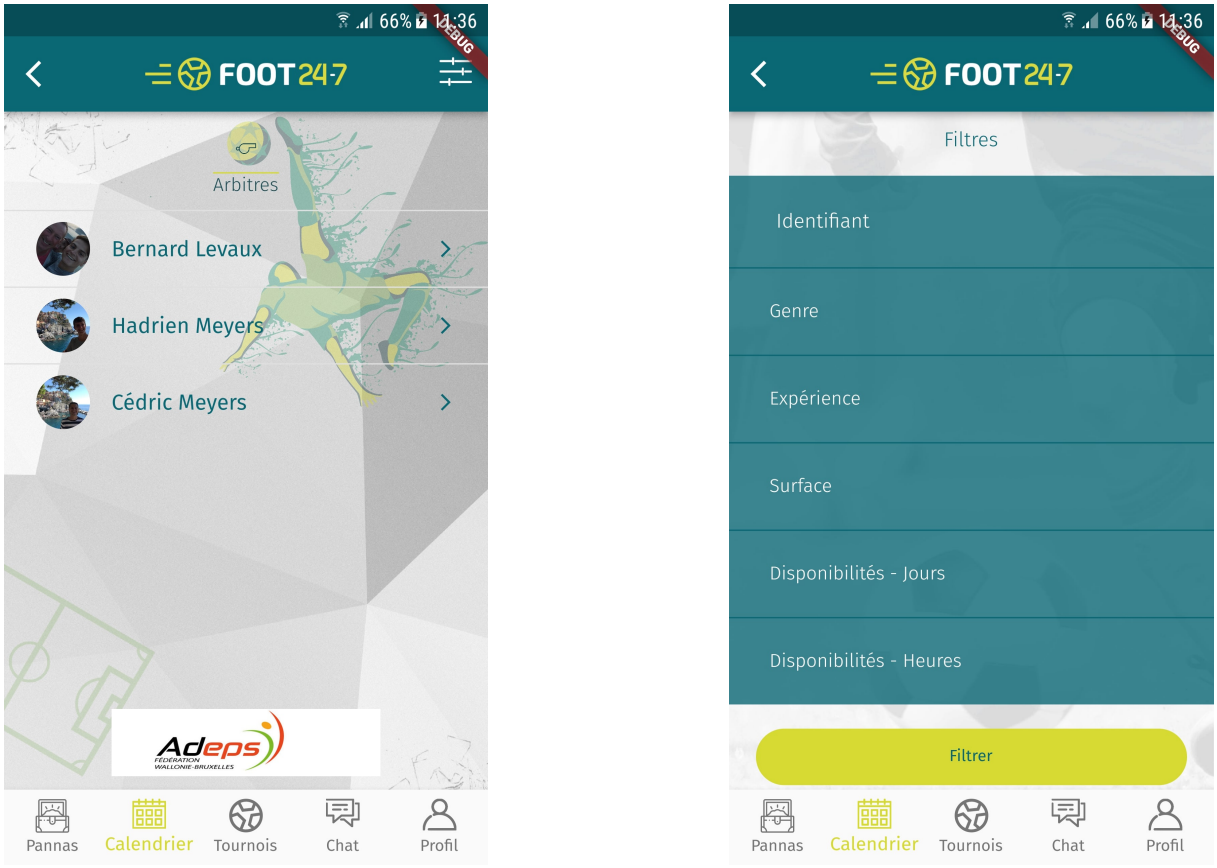


Figure 60: New search screen

This button should redirect to a new page displaying the list of referees registered on the application (as shown in Figure 61a) which thus involves defining a new `referee_list.dart` file in the `referee/screens/` folder. In addition, we should also allow users to filter the list of referees on the application. This requires defining a new `referee_filter.dart` file (in the same `referee/screens/` folder) corresponding to a new page allowing users to filter the list according to different fields (see Figure 61b). To generate the referee list, these two files would obviously rely on the `fetchRefereeList` API function, which we have already discussed at length.



(a) Referee list page

(b) Referee filter page

Figure 61: New referee list and filter pages

Updating the team list page to support referees

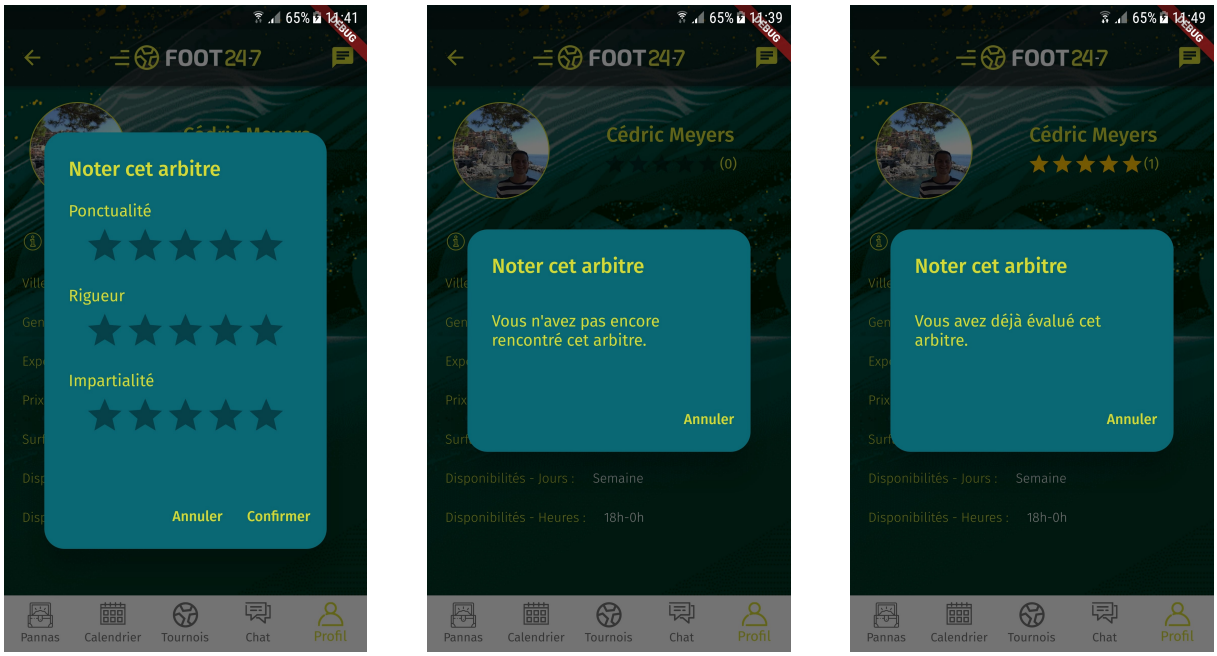
When it comes to the page listing all the teams registered on the application (as can be seen in Figure 37), we have to pay attention to the fact that a referee can not create nor join a team. Consequently, this page should not display the two tabs "Toutes les équipes" and "Mes équipes" nor the "Créer une équipe" button if the user is logged in as a referee. The new page listing the teams on the application for a referee is shown in the following Figure 62.

Figure 62: New team list screen for referees

Furthermore, since a referee can not create nor join a team, we have to make sure that the `TeamDetailSerializer` (see the `account/serializers.py` file) takes into account the user's current status. In fact, the `get_is_captain`, `get_can_join` and `get_can_leave` methods of the `TeamDetailSerializer` must all return false if the user is currently logged in as a referee.

Rating referees

Finally, we have to allow players of the application to rate the referees they have met during games. To do so, we must first create a new `RefereeNote` model in the `account/models.py` file containing the 3 evaluation criteria: punctuality, rigor and impartiality. Then, in order to display on a referee's profile page the average grade of the referee as well as the number of ratings he has received, we need to define two new methods `get_average_note` and `get_number_notes` in the `Referee` model. Besides, similarly to the work described in sections 5.2 and 5.3, we have to add methods `get_encountered_referees` and `get_rated_referees` in the `Player` model. In this way, we will be able to identify the referees that a player has already met as well as those that he has already rated. To transfer all this new information to the frontend, we need to update the `RefereeProfileDetailSerializer` and the `VisitRefereeProfileDetailSerializer` to include the `get_average_note` and `get_number_notes` methods. In addition, as in sections 5.2 and 5.3, the `VisitRefereeProfileDetailSerializer` should also define a `get_can_rate` method and a `get_already_rated` method that will determine, respectively, whether a user can rate a referee and whether a user has already rated a referee. These methods should of course take into account the current status of the user. As a matter of fact, since a referee can not rate another referee, these methods must always return false if the user is logged in as a referee.
In the frontend, we have to modify the `profile_referee.dart` file so that a referee's profile page includes a rating bar (below the referee's name) displaying the referee's average grade as well as the number of ratings the referee has received (as shown in Figure 48). Moreover, when a player clicks on such a rating bar, a rating dialog should appear on the screen allowing the player to either rate the referee or explaining him why he is not allowed to. The different possible rating dialogs are depicted in Figure 63 below.



(a) Referee rating form (b) Referee not met (c) Referee already rated

Figure 63: Referee rating dialog

In order to display the rating bar of a referee and to allow the creation of a `RefereeNote` instance, we have to modify the `profile/api/serializer.dart` file. Indeed, the `RefereeProfile` class should now include the new information provided by the backend, namely the referee's average grade, the number of grades received, as well as the `canRate` and `alreadyRated` variables. Additionally, a new class called `RefereeNote` should also be created in this file to represent a referee's evaluation by a player. In order to create a new referee assessment, we must also modify the `profile/api/api.dart` file by adding a new API function `createRefereeNote`. As usual, this new API function must be linked to the backend by defining a new url in the `account/api_urls.py` file and also requires to define a new `RefereeNoteCreateView` in the `account/api_views.py` file as well as a new `RefereeNoteCreateSerializer` in the `account/serializers.py` file.

As was done in sections 5.2 and 5.3, rating a referee should allow a player to earn pannas. When a player rates a referee, an alert dialog notifying him that he just won pannas should therefore appear on the screen, as shown in Figure 64.

Figure 64: Gamification alert dialog when rating a referee

As requested by Foot 24-7, it is essential to make sure that **a referee can not rate a player nor a team**. In the frontend, we must retrieve the current status of the user in the `profile_player.dart` and `team_detail.dart` files. This way, if the user is logged in as a referee, in the case of a player, clicking on the rating bar will not generate a rating dialog and, in the case of a team, the "Evaluer" button will not be displayed. In the backend, we need to check the current status of the user in the `get_can_rate` and `get_already_rated` methods of the `VisitPlayerProfileDetailSerializer` and `TeamDetailSerializer` (see the `account/serializers.py` file) and return false if the user is logged in as a referee.

Finally, we also have to include a game's referee in the notification inviting players to rate the participants of a game (see sections 5.2 and 5.3). As this mechanism exists for both classic and tournament games, we need to modify both the `game/screens/game_participant_list.dart` and the `tournament/screens/tournament_game_participant_list.dart` files to include the game referee if he exists. For illustrative purposes, an example of a `GameParticipantListScreen` containing a referee is shown in the following Figure 65.

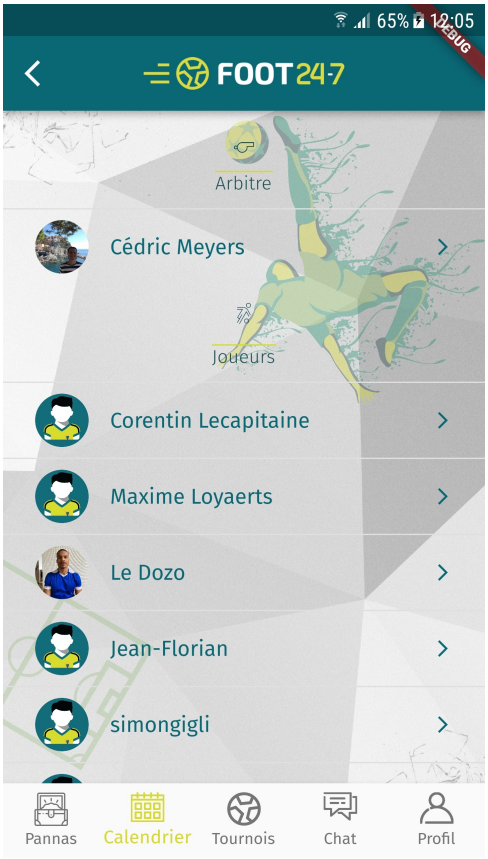


Figure 65: Referee in list of game participants

5.10 Summary

To conclude this section presenting the development of the solution, we can summarise the set of added features as well as those initially offered to the user on the Foot 24-7 mobile application in the following high-level diagram 66. This diagram should be compared to the diagram provided in the presentation of the initial application (see Figure 9) in order to better appreciate the work carried out during this thesis.

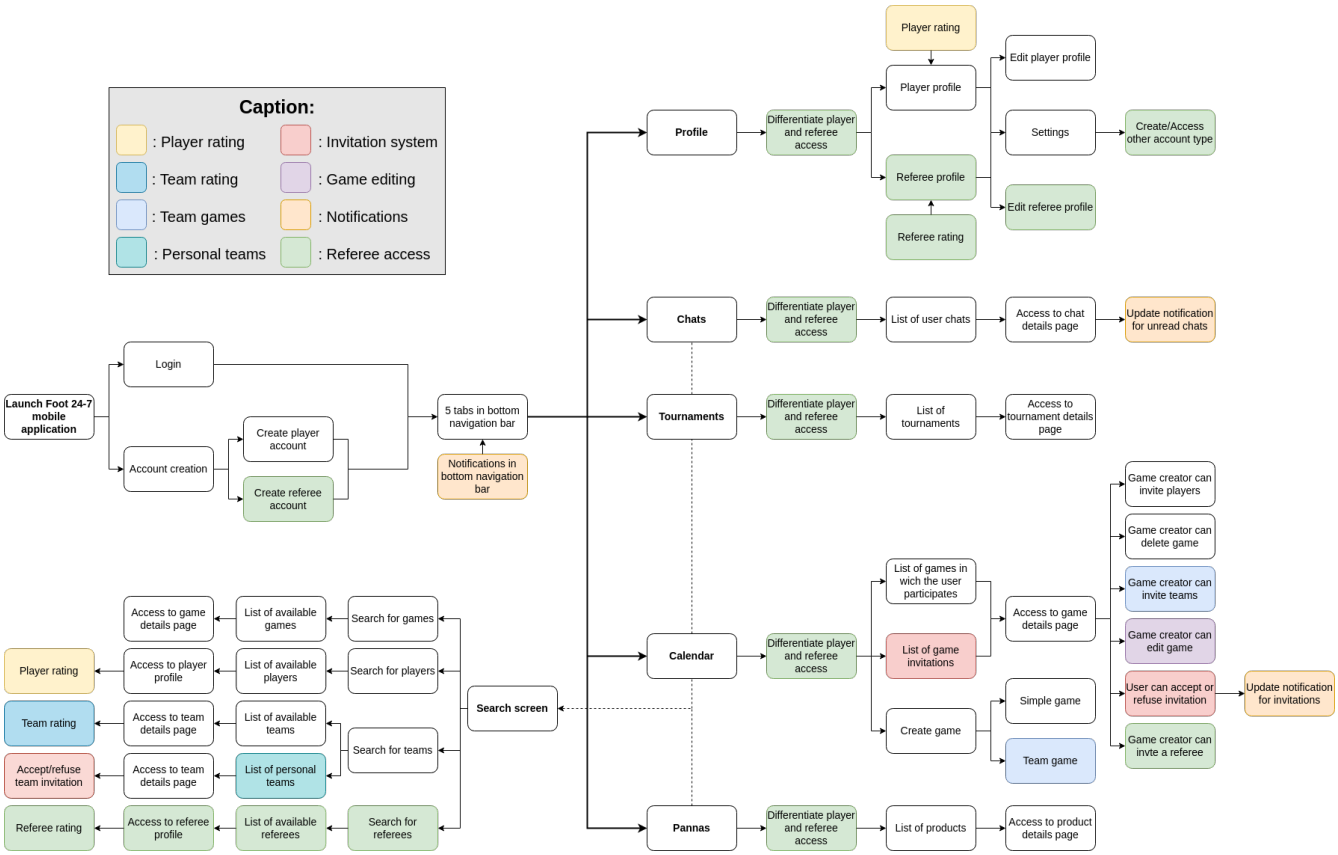


Figure 66: High-level diagram of final features

6 Testing

In this section, we will present the various kinds of tests performed during the course of this thesis. To do so, we will use the testing taxonomy described in B. Donnet's lecture [2]. In this project, **functional testing was of paramount importance**. As a matter of fact, as this project was highly user oriented and focused on the addition of new features to the Foot 24-7 mobile application, the main concern was to make sure that the added features were fully functional. Most of the testing carried out during this thesis is therefore based on testing application use cases.
Of course, in cases where it was justified, I also performed **unit tests** testing the smallest parts of the software (typically, methods/functions).
Moreover, we can also explicit the fact that the approach adopted for the development of this project fits into the philosophy of **incremental integration using a mix of feature-oriented integration and sandwich integration**. Indeed, during the course of this project, each feature was added one after the other and the proper functioning of the application was re-evaluated after each new feature was added. Besides, when adding a new feature, I naturally decided to separate the frontend from the backend and then connect the top-level user interface to the core bottom-level functionalities. This made it easier to detect exactly where the errors were as well as to keep the system in a functional state at all times. I also stuck to the **"Daily build" principle**: at the end of each day, the project compiled without errors and the application worked as intended.
In the context of this project, I also often performed **non-regression tests** to check that the addition of the new features did not interfere with existing ones.
Finally, I also performed **acceptance testing** to make sure that the added features met Foot 24-7's expectations which resulted in a formal acceptance sheet, as can be seen in the Appendix C.

6.1 Debugging the original application

In this project, the first testing phase was related to the debugging of the initial application. Indeed, as discussed in section 4, the discovery phase of the initial application revealed the existence of many bugs in the original application. This led to a **substantial testing phase** consisting in testing all the existing features of the mobile application in order to verify that they worked as intended and if not, to solve them. We will not come back on the details of this testing phase nor on the resolution of the initial errors since all that is detailed in section 5.1.

6.2 Rating other players

As regards rating other players, there were some **critical use cases** to be tested in order to verify the proper functioning of this feature. The following list of use cases presents important and representative critical use cases that had to be tested:

- A user should not be able to rate a player he has not yet met.
- A user should not be able to rate a player he has already rated.
- A user should not be able to rate himself.

When it comes to the first two use cases, the management of these critical cases has already been explained and illustrated in section 5.2. Indeed, Figures 20c and 20d of this section do illustrate the proper handling of these critical cases. However, we have not yet mentioned one particular case that had to be addressed at this stage: when a user rates a player he has already met from this player's profile and then decides to stay on this profile page, his rating should be taken into account in order to prevent him from directly rating that player again. To do this, one must query the backend again in order to update the information in the frontend. This is achieved by using the `_refreshProfile` method defined in the `ProfileScreen` and used in the `_createUserNote` method.

Regarding the fact that a user should not be able to rate himself, one must understand that the `ProfileScreen` contains a `playerPk` variable which is null if the profile matches the current user's profile and, otherwise, is equal to the primary key of the player whose profile the current user is consulting. Therefore, a very simple way to prevent the user from rating himself is to only display a rating dialog if this `playerPk` variable is not null. The value of this variable is thus tested in the `_widgetNotationInfo` widget of the `ProfileScreen` which will only use a `GestureDetector` widget allowing to display a rating dialog if the `playerPk` variable is not null. In this way, it is impossible for a user to rate himself on the mobile application.

Up to now, we only discussed the management of these critical cases in the frontend which, through the interface, limits the user's possibilities of sending bad requests to the server. However, it is important to note that I tried to **make the backend and frontend as robust as possible independently of each other**. Therefore, some use cases are actually double-checked since the backend double-checks what the frontend already checked. In other words, for these cases, anyone trying to interact directly with the server would see their malicious requests rejected by the server, without causing the server to issue an error that would force it to shut down. For instance, the verification that a user can not rate himself is in fact double-checked in the backend. As it happens, the `UserNoteCreateSerializer` defined in the `account/serializers.py` file includes a `validate` method whose goal is to check that the `user` and `noted_user` fields are indeed different. If this is not the case, the backend returns an error specifying that a user can not rate himself and refuses the creation of the `UserNote` instance. To illustrate this case, we will use the Postman tool [31] which allows to interact directly with the backend [8]. The query illustrated in Figure 67 below attempts to create a `UserNote` instance where the rating user and the rated user are the same and shows the appropriate server response to this bad request.

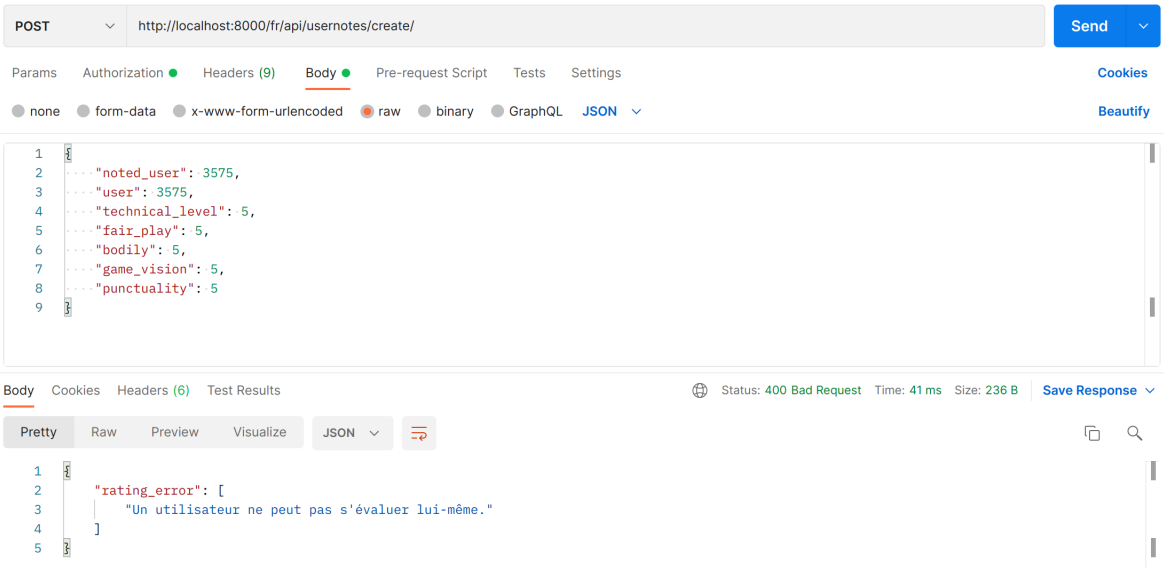


Figure 67: Backend check to a user trying to rate himself

Finally, we can also mention that additional requirements are checked in the backend itself. In fact, one can see in the `UserNote` model of the `account/models.py` file that the fields corresponding to the 5 player rating criteria each contain validators to ensure that the values associated with these fields are indeed integers between 0 and 5. Therefore, if a user tries to interact directly with the backend by specifying unaccepted grades, the backend will return an error and prevent the creation of the `UserNote` instance as illustrated in the following Figure 68.

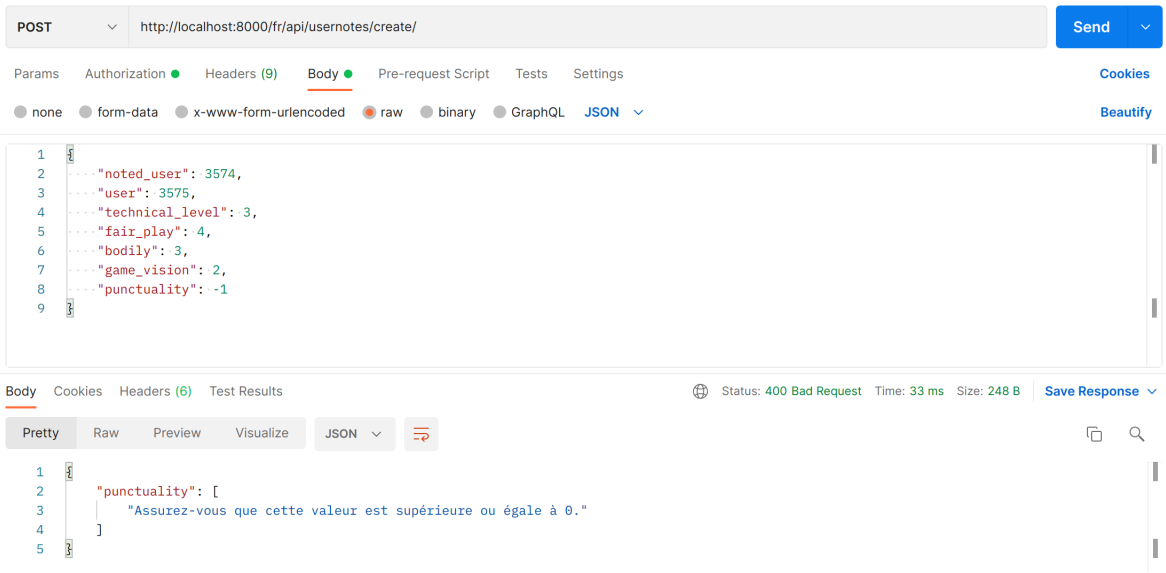


Figure 68: Backend check to player unaccepted grades

6.3 Rating other teams

As regards rating other teams, the list of **critical use cases** that had to be tested is very similar to the one presented in section 6.2:

- A user should not be able to rate a team he has not yet met.
- A user should not be able to rate a team he has already rated.
- A user should not be able to rate a team to which he belongs.

When it comes to the first two use cases, the management of these critical cases has already been explained and illustrated in section 5.3. Indeed, Figures 26b and 26c of this section do illustrate the proper handling of these critical cases. But, similarly to what was explained in section 6.2, there is also one particular use case to be tested at this stage: when a user rates a team he has already met from the team details page and then decides to stay on that page, his rating should be taken into account in order to prevent him from directly rating that team again. To do this, one must refresh the information in the frontend by querying the backend. This is achieved by using the `_refreshTeam` method defined in the `TeamDetailScreen` and used in the `_createTeamNote` method.

Regarding the fact that a user can not rate a team to which he belongs, the checking is also carried out in the frontend using the information returned by the backend. Actually, the `TeamDetailSerializer` defines 3 methods `get_is_captain`, `get_can_join` and `get_can_leave` whose return value is transmitted to the frontend and stored in 3 boolean variables `canDelete`, `canJoin` and `canLeave` of the `TeamDetail` class. These 3 variables are then used to test in the `TeamDetailScreen` whether the current user belongs to the team under consideration, in which case the "Evaluer" button (see Figure 25) will not be displayed on the screen. In this way, it is impossible for a user to rate a team to which he belongs.

Finally, just like in section 6.2, we can also mention that additional requirements are checked in the backend itself. As a matter of fact, one can see in the `TeamNote` model of the `account/models.py` file that the fields corresponding to the 3 team rating criteria each contain validators to ensure that the values associated with these fields are indeed integers between 0 and 5. Therefore, if a user tries to interact directly with the backend by specifying unaccepted grades, the backend will return an error and prevent the creation of the `TeamNote` instance as illustrated in the following Figure 69.

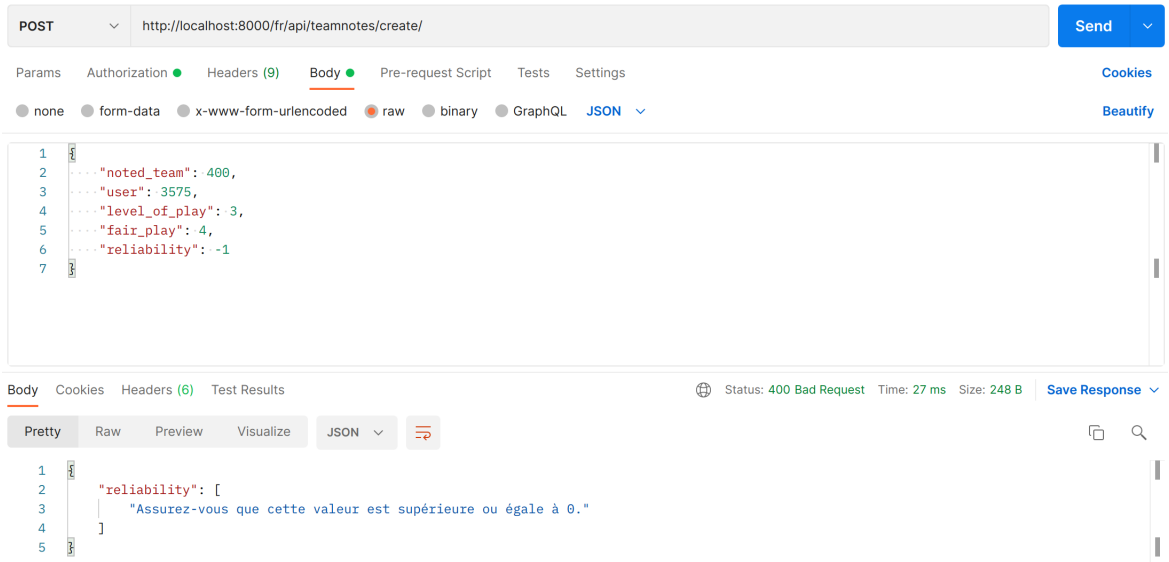


Figure 69: Backend check to team unaccepted grades

6.4 Create games with teams rather than players

As this task significantly modified the use of the `Game` model defined in the `game/models.py` file, many **critical cases** related to the entire application appeared. Among others, we can mention:

- A team captain should not be able to join a public team game if he already belongs to the first team of this game.
- A player encountered only through a team game must be assessable.
- A user should not be able to create/join a team game with a team to which he belongs but of which he is not captain.

The first use case emphasizes that it is important to avoid a situation where a player who is present in the first team of a team game but also has other teams of which he is captain would be offered the possibility to join a team game when he is already present in the first team. Indeed, it would be difficult for the user to understand why he is offered to join a team game when one of his teams is already active in that game. Furthermore, it would also be desirable to avoid as much as possible situations where a player appears in two different teams of a team game. Therefore, in the case of a team game, it is important to check in the `get_can_join` method of the `GameDetailSerializer` (see the `game/serializers.py` file) that the current user is not already present in the first team, beyond the fact that the user must also be captain of a team, that the team game must be public and not already include a second team.

As for the second point, the addition of the possibility to create a game by specifying teams rather than players implies that it is now possible to meet players through team games. It was therefore essential to modify the `get_encountered_players` method of the `Player` model (see the `account/models.py` file) in order to also take into account the players of teams participating in team games. In particular, in order to verify the proper functioning of this new feature, it was crucial to check that a player encountered only through a team game was still a player that the user could rate.

The last particular case mentioned above insists on the fact that only team captains should be able to create (or even join) team games. To ensure this, it is first important to define a `fetchTeamList` function in the `team/api/api.dart` file that takes into account the type of list to be fetched from the backend. By specifying the `String` "captain", the `TeamListView` associated with this function (see the `account/api_views.py` file) should allow to fetch only the teams for which the current user is the captain. This would allow to make sure that a user can only choose a team he is captain of when he tries to create a team game (see Figure 28) or when he tries to join a team game (see Figure 33). In the case of joining a team game, it is also important that the `get_can_join` method of the `GameDetailSerializer` checks among other things that the user is captain of at least one team as mentioned above.

Finally, we can also mention that in addition to these various tests carried out on the different critical use cases, **non-regression testing** was also very important in the context of this task. As a matter of fact, since the way the `Game` model is used and created was modified, I took the time to check that the creation and handling of simple games were still working properly.

6.5 Redesigning the invitation mechanism

When it comes to the redesign of the invitation mechanism, this task had an impact on the whole application. It was therefore essential to check that the integration of this functionality worked with all features already present. Several **critical use cases** had to be tested. Some of these are listed below:

- An invitation to a team game must only appear in the team captain's calendar.
- An inactive player in a game should not be considered as a player encountered by other active players in that game.
- An active team player whose team is inactive in a team game should not be considered as a player encountered by the active players of the first team of that game.
- An inactive team player whose team is active in a team game should not be considered as a player encountered by the active players of the teams in that game.
- An inactive team in a team game should not be considered as a team met by the active players of the other team in that game.
- Only active players of a game should appear on the page inviting to rate the participants of a game.

The first critical case is related to the feature allowing to create team games (see section 5.4). When a creator of a team game decides to invite another team to this game, it is important to make sure that this invitation only appears on the calendar page of the captain of this team since he is the only member of this team who is able to answer this invitation. In the "Mes invitations" tab of the calendar page, one should thus display the team games for which the second team is one of the teams for which the current user is captain and is inactive. To do this, we need to take into account these games in the `CalendarInvitationListView` of the `game/api_views.py` file.

All the other critical cases mentioned above are related to the functionalities allowing to rate players or teams (see sections 5.2 and 5.3). In order to ensure the proper integration of the new invitation system with the rating features, all these cases must be handled in the `get_encountered_players` and `get_encountered_teams` methods of the `Player` model (see the `account/models.py` file). These methods must take into account whether players or teams encountered in various games are active or not and at all levels (*e.g.* to be able to rate a player encountered through a team game, this player must be active in his team and his team must be active in this game). Furthermore, in these methods, one should also pay attention to only consider games (simple games or team games) where the current player is himself active (and at all levels).
For the last case, it is indeed important to only display active players (at all levels) in the `GameParticipantListScreen` (see the `game/screens/game_participant_list.dart` file). To do this, the information about whether a player is active or not in the game must be included in the `get_players` method of the `GameDetailSerializer` (see the `game/serializers.py` file) so that it can be used in the frontend.

Finally, we can also mention that besides all these use cases, **non-regression testing** was also very important in the context of this task. As it happens, since the invitation system was completely redesigned, I had to take the time to check that the original invitation system via chats was still working properly.

6.6 Game editing

As explained in section 5.7, the form allowing to edit a game was implemented in such a way that the user can not enter invalid inputs. In order to make sure that I had considered all the critical cases, I carried out functional testing, using test cases with **boundary value analysis**. To do so, I constructed **equivalent classes** (ECs) for each variable in the form, as detailed in the Table 2 below.

	Valid EC	Invalid EC
Date	1) $\geq$ current date	2) $<$ current date
Time	3) $\geq$ current time	4) $<$ current time
Price	5) Number AND ≥ 0	6) Not a number 7) < 0
Participants	8) Integer AND $\geq$ current number of players	9) Not an integer 10) $<$ current number of players
Duration	11) Number AND ≥ 0	12) Not a number 13) < 0
Official stadium	14) Not empty IF `_officialStadium` is true AND initial stadium is not official	15) Empty IF `_officialStadium` is true AND initial stadium is not official
Custom stadium	16) Address not empty IF `_officialStadium` is false	17) Empty address IF `_officialStadium` is false

Table 2: EC construction for edit game functionality

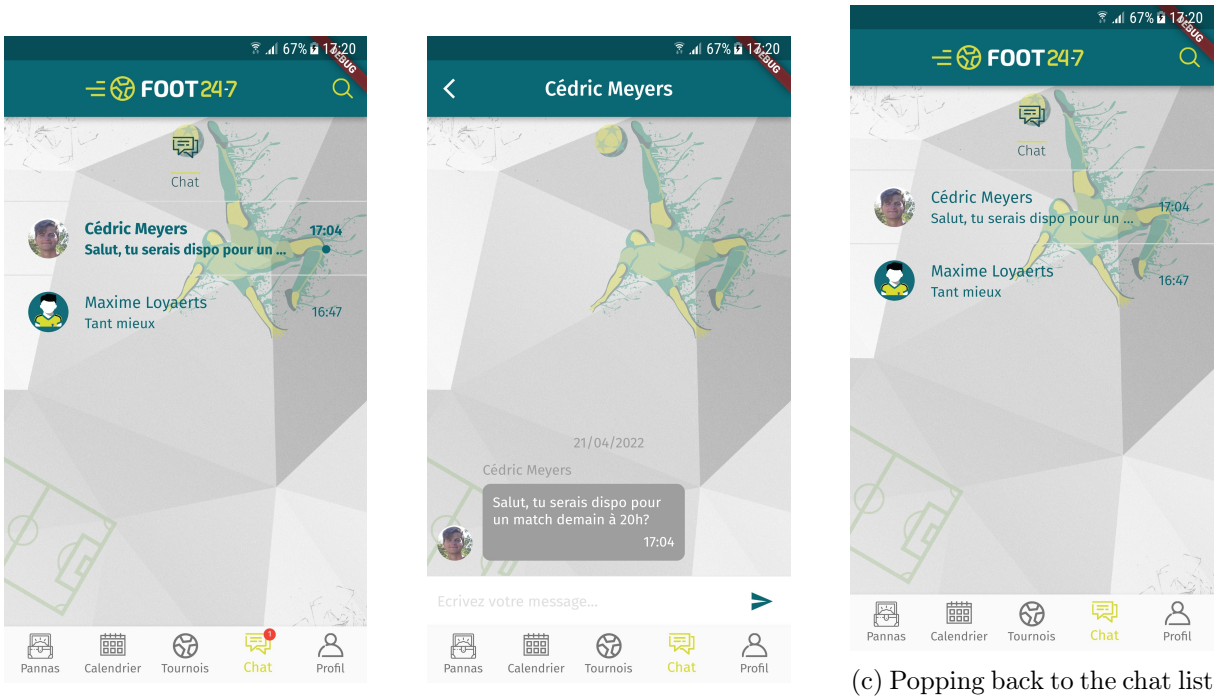
From this table, I then created different test cases. The first test case gathers all valid ECs. For the others, I tested each invalid EC separately. For the sake of clarity, the tables containing all those test cases are reported in Appendix B. Following the testing of these 11 cases, I did not detect any bug or unwanted behavior regarding the use case "edit a game".

6.7 Improving the bottom navigation bar with notifications

As explained in section 5.8, the **critical cases** in the context of this task were related to the refreshing of the bottom navigation bar at the appropriate times. It was indeed important to check that the bottom navigation bar was refreshed properly when:

- The user clicks on an unread chat.
- The user responds to a game invitation.
- The user pops back from a chat to the chat list page.
- The user pops back from a game to his calendar.

As already mentioned in section 5.8, to manage these different critical cases, it was necessary to define an `onRefresh` method in the `BottomNavigationWidget` (see the `widgets/bottom_navigation.dart` file) allowing to update the number of unread chats and the number of pending invitations, as well as to use `GlobalKeys` to call this method in the different related files.
Following the tests carried out on these 4 critical cases, I did not detect any bug related to the refreshing of the bottom navigation bar. We will only illustrate here the refreshing of the bottom navigation bar during the critical case "reading an unread chat" in the following Figure 70 but the same could be applied to the critical case "responding to a game invitation".



(a) Unread chat in the chat list

(b) Clicking on an unread chat

(c) Popping back to the chat list page

Figure 70: Testing reading an unread chat

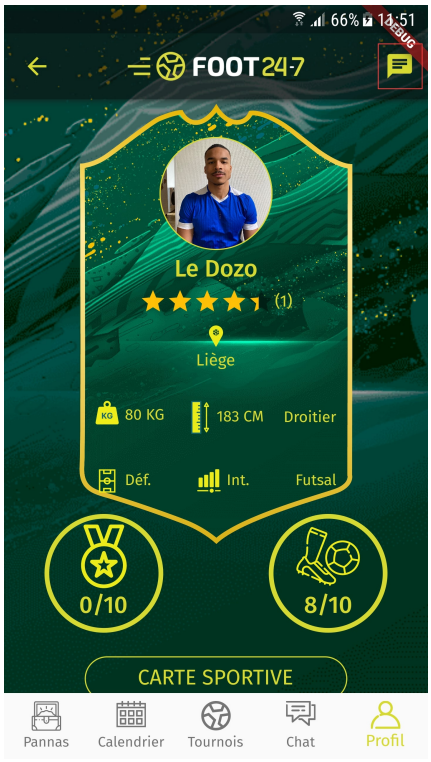
6.8 Development of a new referee access

The addition of a brand new referee access represented a major change to the functioning of the original application. Therefore, the integration of this functionality required the testing and management of a multitude of critical cases.
For instance, as regards adding players, teams or referees to a game, since a user can now hold both a player and a referee account simultaneously, we have to make sure that a user can not be involved in a same game twice but with different roles. Consequently, this required testing the following **critical use cases** to verify that the application reacted properly in these situations:

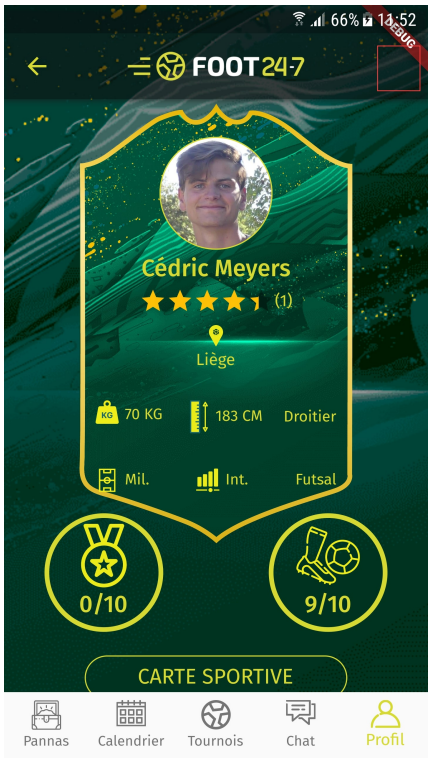
- A game creator who invites a user to referee his game should not be able to invite the same user as a player in this very same game.
- A game creator who invites a user to join his game as a player should not be able to invite the same user as a referee in the same game.
- A game creator who invites a user to referee a team game should not be able to invite a team containing the same user but as a player in the same game.
- A game creator who invites a team to join a team game should not be able to invite as a referee a user who is already part of the invited team as a player in the same game.
- A user logged in as a player should not be able to join a game (or a team game) if he is already the referee of that game.
- A user logged in as a referee should not be able to join a game (or a team game) if he is already a player (or team player) of that game.
- A user logged in as a player should not be able to join a team game with a team containing a user corresponding to the referee of that game.

As explained in section 5.9, the management of these different use cases had to be performed in the `get_can_join` method of the `GameDetailSerializer` (see the `game/serializers.py` file) but also in the `PlayerListView`, `RefereeListView` and `TeamListView` of the `account/api_views.py` file. After testing all these critical cases, I did not detect any bug or unwanted behaviour regarding the addition of players, referees or teams to a game.

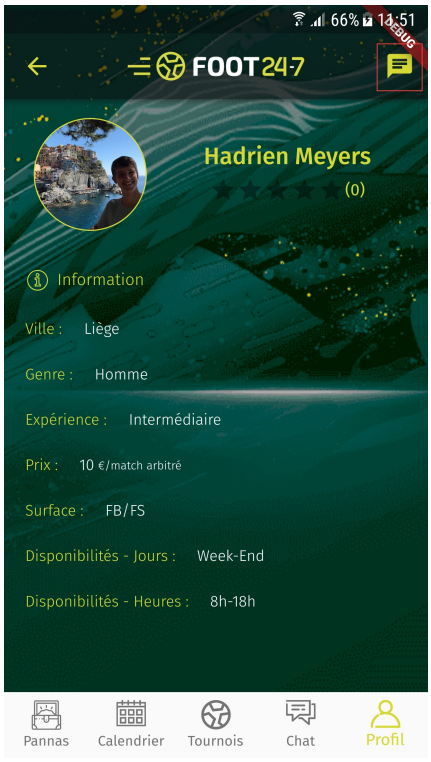
Furthermore, now that a user can hold two different types of account simultaneously, it is important to ensure that a user visiting his own profile page but the one corresponding to his other type of account (*i.e.* a user visiting his referee profile page as a player or vice versa) is unable to create a chat with himself. To achieve this, we need to modify the `profile_referee.dart` and `profile_player.dart` files of the `profile/screens/` folder so that the icon allowing to chat with the user is only displayed if that user differs from the current user. The following Figure 71 highlights the fact that a user viewing his own profile page with his other type of account does not have the opportunity to create a chat.



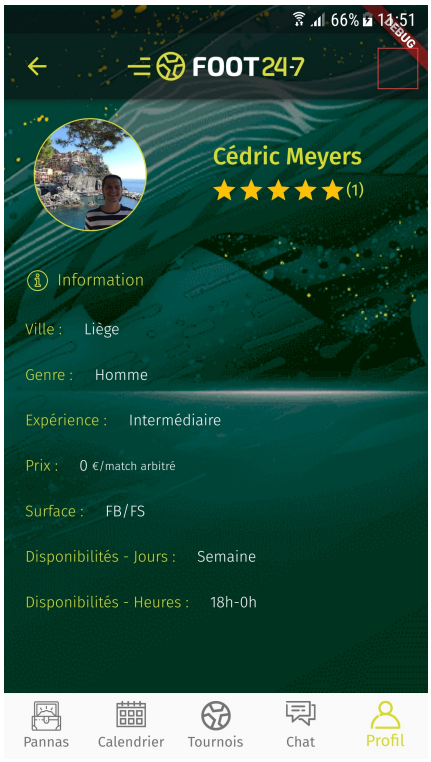
(a) Viewing another's player profile



(b) Viewing his own player profile as a referee



(c) Viewing another's referee profile



(d) Viewing his own referee profile as a player

Figure 71: Consulting one's own profile page with one's other account type

In the backend, we can also mention that the creation of a chat between a user and himself was also prevented. In fact, one can see in the `ChatGetOrCreateView` (see the `chat/api_views.py` file) that the creation of a chat between two users is only possible if these two users differ from each other.

As explained in section 5.9, a referee can also edit his profile. However, in the referee profile edit form, there is one critical field whose validity must be checked: the price specified by the referee for refereeing a game. As a matter of fact, we have to make sure that the user fills in a positive number for this field. If not, the referee should not be able to save the specified information and we should also display an error message on the screen, as can be seen in Figure 72 below.

(a) Price is not a number

(b) Price is negative

Figure 72: Checking validity of referee profile edit form

For the other fields of the form, no verification is required as the user can only modify them by choosing from a predefined set of possibilities. The validity of these fields is therefore guaranteed.

Here, I only highlighted some representative critical use cases, but obviously there are others. For example, we can mention that a testing similar to those described in sections 6.2 and 6.3 was also applied to check that the rating of referees by players worked as it should. However, to prevent this report from getting excessively long, we will not go into further details here.

Finally, it is also important to mention that besides all these use cases, **non-regression testing** was also of great importance in the context of this task. As it happens, since a brand new referee access was added to the application, I had to take the time to check that the original player access was still working properly.

7 Conclusion

This last section marks the conclusion of this enriching thesis. Using well-known application development technologies such as the `Flutter` framework, the `Django` framework and the `Django REST` framework, this thesis addressed several development tasks and allowed to integrate many new features on the Foot 24-7 mobile application: debugging of the original application, development of player and team rating systems, addition of the possibility to create team games, access to a user's personal teams, redesign of the whole invitation system, addition of the possibility to edit a created game, improvement of the bottom navigation bar with notifications as well as the development of a brand new access dedicated to referees. In this section, we will start by reviewing the status of the project following the work carried out in the context of this Master thesis. First, as every project has its own flaws, we will focus on the **current limitations** of the solution developed. Then, we will consider **future perspectives** for the further development of the project. Lastly, we will conclude this thesis with a few final words on the gratifying experience that is the completion of a Master thesis.

7.1 Limitations

As regards the current limitations of the developed solution, we can mention that the testing of the newly developed features sometimes simply consisted in preventing a user from performing a faulty action via the frontend only. I sometimes included additional checks in the backend (as illustrated in sections 6.2 and 6.3) but I did not have enough time to apply this rigorously to all the added features. For instance, regarding game editing, checking the validity of the fields of the game editing form as described in section 6.6 is only performed in the frontend. However, for the sake of completeness, it would be convenient to perform similar checks in the backend in order to double check the validity of the submitted form. In this way, we could make the frontend and the backend robust completely independently of each other.

7.2 Future works

To begin with, all the tasks initially suggested by Foot 24-7 (see section 3) that we did not have the opportunity to consider in the context of this thesis are obvious and natural perspectives for the future development of the application. Along with these tasks, we can also include the task related to the improvement of the player participation survey tool for a tournament game which I had identified when discovering the original application, code and features (see section 4) but unfortunately did not have enough time to consider either.

As mentioned in section 4, **UX design** is a key area of development for Foot 24-7. In fact, user experience is Foot 24-7's main priority but, in my opinion, the user interface could be improved: the user interface is not always the cleanest and most attractive (*e.g.* the "Pannas" tab, the profile page tutorial, the presentation of the ads, etc.), it is not always uniform and consistent (*e.g.* the dialogs allowing to select a date and time when creating a game do not respect the colour code of the application, etc.) and the access to certain features or information is not always very intuitive (*e.g.* the "Créer une équipe" button should not be located in the settings of the profile page). All these things reduce the quality of the user experience on the application. Although UX design was not part of the scope of this Master thesis, I do think it would be beneficial for the future of Foot 24-7 to reorganise, harmonise and improve the design of its application in order to provide the cleanest and smoothest user experience.

Another prospect for the further development of the application relates to the **state management** in the `Flutter` frontend. Actually, in the frontend, state management solely relies on the use of `StatefulWidget` and `setState`. The advantage of this approach is that it is very

easy and straightforward to understand. Furthermore, this method is a great way (and is even recommended by `Flutter`) to handle *ephemeral state*[5], which is the majority of state management we must perform in the case of the Foot 24-7 mobile application. However, this approach does come with some drawbacks. First of all, it is not a good way to manage *app state*[6]. In the case of Foot 24-7, app state mainly refers to the currently logged-in user. Indeed, information about the currently logged-in user is requested in several different pages of the application: the "Profil" tab, the "Pannas" tab, etc. Moreover, the addition of a brand new access dedicated to referees (see section 5.9) implies that the current status of the user must now be known in many different pages of the application which thus increases the amount of information shared by different pages in the application. On the other hand, using `setState` all over an application may not be the best approach to ensure the proper maintenance of the application as the state is scattered all over the place. Besides, `setState` is used within the same class as the UI code, mixing UI and business logic, which breaks clean code principles. In the case of Foot 24-7, as the structure of the code is relatively clear and straightforward to approach, this is not too much of a problem. However, in a long term perspective, this could become a potential issue at some point. One solution would be to use another state management option such as the `provider` package [3] (which is a recommended approach by `Flutter`). In short, `provider` can be described as a mix between state management and dependency injection. Therefore, `provider` allows the sharing of data between different widgets of the `Flutter` widget tree and would thus allow to share the app state across all the pages of the application. In this way, we would only have to fetch the user information once rather than refetch the information we need on the currently logged-in user in each page of the application, thereby optimizing the state management in the application. Another solution would be to use a design pattern such as `BLoC` [10]. `BLoC` stands for Business Logic Component and is a `Flutter` library for state management. This design pattern allows to structure a `Flutter` project by separating the business logic from the UI. This option would therefore improve the quality of the code, facilitate its maintenance and testability and, from this standpoint, would be more suitable for the long-term development of the Foot 24-7 mobile application. However, it is true that adopting the `BLoC` design pattern in the case of the Foot 24-7 mobile application would require a significant effort to restructure the application's code and would also result in a more boilerplate code. Consequently, the most appropriate option in the case of Foot 24-7 is perhaps to use the `provider` package, at least as a starting point.

Another future development area which is worth considering concerns the integration of **null safety** into the `Flutter` frontend. As explained in the `Dart` documentation [12], when opting for null safety, `Dart` assumes that types are non-nullable by default, which means that variables can not contain null unless we explicitly indicate that they can. With null safety, runtime null-dereference errors become edit-time analysis errors. Null safety thus makes code safer and provides a better development environment with fewer runtime errors. Furthermore, thanks to sound null safety[7], the `Dart` compiler can optimise our code, which leads to faster programs. Since the Foot 24-7 project was created before the introduction of null safety in 2020, this project does not support it. However, given its many advantages, it would be worth considering migrating the application to null safety. Although it is true that migrating to null safety can turn out to be a tedious process for large projects, one can always find resources [11] [6] explaining how to apply such a migration while keeping a project growable, maintainable and easy to release.

[5] As explained in the `Flutter` documentation [23], ephemeral state is the state you can neatly contain in a single widget.

[6] As explained in the `Flutter` documentation [23], app state is the state that you want to share across many parts of your app.

[7] Sound null safety means that if `Dart` determines that a variable is non-nullable, then this variable is always non-nullable.

Upgrading the dependencies used in both the frontend and the backend would also be a task to consider in the near future. Indeed, as the project starts to become quite old, many of these dependencies are relatively outdated. For example, in the backend, the project uses `Django` version 1.11.17, whereas the latest official version is 4.0.4. Using the latest stable version of `Django` would be of great interest from a security, optimization and maintenance standpoint. As a matter of fact, several versions of the dependencies used in the project may soon no longer be maintained, which is already the case for the `Django` framework [17]. On the other hand, as regards the frontend, updating the dependencies should be considered in conjunction with the integration of null safety in the project, as previously discussed.

As stated in section 4, the documentation of the code, although reasonable, remains relatively poor. Even if this level of documentation is sufficient to easily tackle the project, documenting the entire application code in a more extensive and rigorous way could nevertheless prove to be valuable in order to facilitate the sharing and maintenance of the code and thus invest in the long-term development of the application.

Another suggestion to improve the code quality in the frontend would be to use **linting**. Linting is the process of automatically checking source code for programmatic and stylistic errors. This process is performed by a lint tool also known as linter. It allows to identify certain errors made during coding such as unused variables or imports, omission of required parameters, empty if-else statements, etc. Through the definition of linting rules [13] (*i.e.* coding rules), linting allows to check the quality of the code and therefore provides a clean code base. In the case of Foot 24-7, using linting could be of great value to ensure consistency in the code. Indeed, as the project passes through the hands of several different people over time, linting would allow to impose the same coding conventions to all developers working on the project thereby easing the long term maintenance of the Foot 24-7 mobile application. There exist different packages allowing to set up linting rules in a `Flutter` project but the most popular is probably the `flutter_lints` package [7].

Finally, a last prospect for further development relates to **internationalization**. In fact, as explained in section 2.4, the code of the mobile application was structured in a way to support 3 different languages, namely French, English and Dutch. However, since many translations into English and Dutch are yet to be defined, the application currently only supports French. Therefore, translating all the texts used in the mobile application into English and Dutch would be an interesting task to consider in order to integrate the support of several languages in the mobile application and thus foster its potential international development.

7.3 Final words

This Master thesis allowed me to learn many new things on software development and more precisely on mobile or web application development, a subject that is of great interest to me.
The technologies used in this project (namely the `Flutter`, `Django` and `Django REST` frameworks) were technologies I knew either very little or not at all at the beginning of this thesis and I am happy to say that they are now technologies I master much better.
Beyond the acquired hard skills, this thesis also allowed me to acquire new soft skills. As it happens, as this thesis was carried out in collaboration with the company Foot 24-7, this project allowed me to dive into the corporate world. I was able to learn how to manage and understand a client's requests, to communicate and explain my work to people who do not necessarily have an IT background and to discover the professional relationships that exist between different companies.
Finally, I am proud of the work I have done in the context of this Master thesis and I am delighted to have pleased the Foot 24-7 company.

List of Figures

List of Tables

Bibliography

[1] edvard_munch. *Django migration error: you cannot alter to or from M2M fields, or add or remove through= on M2M fields*. Stack Overflow. Oct. 2019. URL: https://stackoverflow.com/questions/26927705/django-migration-error-you-cannot-alter-to-or-from-m2m-fields-or-add-or-remove.

[2] B. Donnet. *Lecture: PROJ0010-1: Software project engineering and management: Software engineering: Testing*. Université de Liège. 2020.

[3] dash-overflow. *provider*. Flutter. Dec. 2021. URL: https://pub.dev/packages/provider.

[4] Sarbagya Dhaubanjar. *flutter_rating_bar*. Flutter. Feb. 2021. URL: https://pub.dev/packages/flutter_rating_bar.

[5] Flutter. *shared_preferences*. Flutter. Mar. 2021. URL: https://pub.dev/packages/shared_preferences.

[6] Polina C. *Gradual null safety migration for large Dart projects*. Medium. Mar. 2022. URL: https://medium.com/dartlang/gradual-null-safety-migration-for-large-dart-projects-85acb10b64a9.

[7] Flutter. *flutter_lints*. Flutter. Apr. 2022. URL: https://pub.dev/packages/flutter_lints.

[8] Postman. *Authorizing requests*. Tech. rep. May 2022. URL: https://learning.postman.com/docs/sending-requests/authorization/#api-key.

[9] *Adeps' web page*. Adeps. URL: https://www.sport-adeps.be/.

[10] *BLoC design pattern*. Bloc Community. URL: https://bloclibrary.dev/#/.

[11] Dart. *Migrating to null safety*. Tech. rep. URL: https://dart.dev/null-safety/migration-guide.

[12] Dart. *Sound null safety*. Tech. rep. URL: https://dart.dev/null-safety.

[13] Dart. *Supported lint rules*. URL: https://dart-lang.github.io/linter/lints/.

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

[15] *Decathlon's web page*. Decathlon. URL: https://www.decathlon.be/fr/.

[16] Django. *Models*. Tech. rep. URL: https://docs.djangoproject.com/en/4.0/topics/db/models/.

[17] Django. *Unsupported previous releases*. Tech. rep. URL: https://www.djangoproject.com/download/.

[18] *Django REST framework's web page*. Django REST framework. URL: https://www.django-rest-framework.org/.

[19] *Django's web page*. Django. URL: https://www.djangoproject.com/.

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

[21] Docker. *Use containers to Build, Share and Run your applications*. URL: https://www.docker.com/resources/what-container/.

[22] *Docker' web page*. Docker. URL: https://www.docker.com/.

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

[24] Flutter. *Internationalizing Flutter apps*. Tech. rep. URL: https://docs.flutter.dev/development/accessibility-and-localization/internationalization.

[25] Flutter. *Navigate with named routes*. Tech. rep. URL: https://docs.flutter.dev/cookbook/navigation/named-routes.

[26] Flutter. *Pass arguments to a named route*. Tech. rep. URL: https://docs.flutter.dev/cookbook/navigation/navigate-with-arguments.

[27] *Flutter's web page*. Flutter. URL: https://flutter.dev/.

[28] *Foot 24-7's web page*. Foot 24-7. URL: https://foot24-7.com/.

[29] *NGINX: Advanced Load Balancer, Web Server, & Reverse Proxy*. F5 NGINX. URL: https://www.nginx.com/.

[30] *Pagination*. Django REST framework. URL: https://www.django-rest-framework.org/api-guide/pagination/.

[31] *Postman API Platform*. Postman. URL: https://www.postman.com/.

[32] *Python's web page*. Python. URL: https://www.python.org/.

[33] *Redis' web page*. Redis. URL: https://redis.io/.

[34] *Time zone support*. Django. URL: https://docs.djangoproject.com/en/4.0/topics/i18n/timezones/.

Appendix

A User feedback on the Foot 24-7 mobile application

Here are the details of the answers provided by the users of the Foot 24-7 mobile application to the questionnaire described in section 4:

- What do you like about the Foot 24-7 mobile application ?
    1. I like the gamification of the app as well as the user interface. The app is easy to use thanks to its intuitiveness.
    2. L'accessibilité aux infos de match.
    3. The concept of bringing together people that want to play football. The idea of creating a team "profile" in which you can add player. The fact that it is an easy way to check the schedule and the place of a game.
    4. There is all the news needed about the competition we play in.
    5. Pouvoir trouver et se faire de nouveaux partenaires de football grâce à l'application.
    6. Je trouve le concept de l'application assez innovant et créatif. J'aime aussi le fait que l'on puisse gagner des cadeaux via l'application.
    7. Bon concept utile et pratique.
    8. I like the fact that you can vote for the best player of the game or predict the score of a game in order to earn pannas and buy products.
    9. The centralisation of information for university tournaments.
- What don't you like about the Foot 24-7 mobile application ?
    1. According to me, there would not be any con to the application, if all the features implemented worked fine. However, this is not case... So, I don't like the navigation between pages and inner navigation in the different pages (scrolling, etc). In practice, the chat system is not useful as the notification mechanism does not alert the user when a message is received.
    2. Les bugs.
    3. The way you have to connect to it. The interface which is not always intuitive. The organisation of the the teams and the fact that it is not really easy to contact a team.
    4. We have to go in the app to send a message. It could have an interface via the "volet de navigation" of our smartphone for an easier use.
    5. Le fait qu'il n'y ait pas de notification pour un nouveau message. L'interface n'est pas toujours très intuitive.
    6. L'application contient certains bugs et l'organisation en équipes via l'application n'est pas toujours évidente.
    7. L'application n'est pas toujours simple à comprendre et présente quelques bugs.
    8. It is not really easy to reliably organize of football game via the application.
    9. Bugs in the application, lack of intuitiveness.

- What are your suggestions for improving the Foot 24-7 mobile application ? What new features would you like to see in the app ?

    1. I don't have any brand new idea to add to the concept. However, I suggest to solve all the minor and greater bugs (stated above or others that are present in the app) to drastically improve the user experience of the application.
    2. Résoudre les bugs/défauts, rendre l'interface un poil plus simple a utiliser.
    3. Improving the interface and make something more intuitive, close to some app that we use everyday. Give the ability to spot game pitch near to your location when you want to organise a game. Give the ability to organise polls in private conversation with the members of your team to organise a game (choose the date, the time...) for example.
    4. As said in the previous question. It could have also a GPS that shows the way to the field from where we are. Sometimes the game is in a place we do not know and we have to use google maps to find the place. It could be easier if it was already implemented in the app.
    5. Ajouter des notifications de nouveaux messages et rendre l'interface plus intuitive.
    6. Il faudrait résoudre les bugs de l'application. Une bonne idée serait aussi d'offrir encore plus de possibilités de cadeaux selon les demandes des utilisateurs.
    7. Il faudrait rendre l'application plus propre et résoudre les petits problèmes existant pour améliorer l'expérience a l'utilisateur.
    8. Clarify and improve the UI of the application. Develop the possibility of earning pannas to buy products.
    9. Solve the existing bugs.

B Edit game use case - Test cases

To understand the invalidity of the dates and times in the following tests, it is important to note that these tests were carried out on 20/04/2022 at 15:00. Likewise, to understand the invalidity of the number of participants, it is important to realise that the edited game considered in the following tests already contains 9 participants.

Test case	1	2	3
EC used	1), 3), 5), 8), 11), 14), 16)	2), 3), 5), 8), 11), 14), 16)	1), 4), 5), 8), 11), 14), 16)
Date	20/04/2022	19/04/2022	20/04/2022
Time	20:00	20:00	14:00
Price	0	0	0
Participants	10	10	10
Duration (min)	60	60	60
Public	True	True	True
`_officialStadium`	True	True	True
Initial stadium	Official	Official	Official
Official stadium	Empty	Empty	Empty
Custom stadium	Empty	Empty	Empty

Test case	4	5	6
EC used	1), 3), 6), 8), 11), 14), 16)	1), 3), 7), 8), 11), 14), 16)	1), 3), 5), 9), 11), 14), 16)
Date	20/04/2022	20/04/2022	20/04/2022
Time	20:00	20:00	20:00
Price	.-	-2	0
Participants	10	10	10.5
Duration (min)	60	60	60
Public	True	True	True
`_officialStadium`	True	True	True
Initial stadium	Official	Official	Official
Official stadium	Empty	Empty	Empty
Custom stadium	Empty	Empty	Empty

Test case	7	8	9
EC used	1), 3), 5), 10), 11), 14), 16)	1), 3), 5), 8), 12), 14), 16)	1), 3), 5), 8), 13), 14), 16)
Date	20/04/2022	20/04/2022	20/04/2022
Time	20:00	20:00	20:00
Price	0	0	0
Participants	8	10	10
Duration (min)	60	.-	-10
Public	True	True	True
`_officialStadium`	True	True	True
Initial stadium	Official	Official	Official
Official stadium	Empty	Empty	Empty
Custom stadium	Empty	Empty	Empty

Test case	10	11
EC used	1), 3), 5), 8), 11), 15), 16)	1), 3), 5), 8), 11), 14), 17)
Date	20/04/2022	20/04/2022
Time	20:00	20:00
Price	0	0
Participants	10	10
Duration (min)	60	60
Public	True	True
`_officialStadium`	True	False
Initial stadium	Custom	Official
Official stadium	Empty	Empty
Custom stadium	Empty	No address

C Acceptance sheet

13th May 2022

Project Client Acceptance Sheet

To the best of my knowledge, information, and belief, and on the basis of my observations and inspections, the work carried out by the intern Mr Cédric Meyers as part of his Master's thesis at the University of Liège concerning the development of the Foot 24-7 mobile application meets my expectations.

Here is the list of completed items:

- Debugging of the original application.
- Addition of a player rating system.
- Addition of a team rating system.
- Addition of the possibility to create a game by specifying teams rather than players.
- Addition of an access to a user's personal teams.
- Redesign of the whole invitation system.
- Addition of the possibility to edit a created game.
- Addition of notifications in the bottom navigation bar of the application.
- Addition of a brand new access dedicated to referees on the application.

By signing this document, I acknowledge that all project deliverables listed above comply with the project specifications and requirements.

Signature of the industrial supervisor (Mr Rayan Kassir)