



IPC – Intelligent Personalized Care

A Solution to promote Physical Agility and Recovery

Authors: Guilherme Cepeda
Tiago Martinho
Rodrigo Neves

Supervisor: Paulo Pereira, ISEL

Project report written for Project and Seminar 22/23
Computer Science and Engineering Bachelor's degree

July 2023

Instituto Superior de Engenharia de Lisboa

IPC – Intelligent Personalized Care

A Solution to promote Physical Agility and Recovery

47531 Guilherme Pedro Serra Cepeda
a47531@alunos.isel.pt

48256 Tiago Alexandre Monteiro Martinho
a48256@alunos.isel.pt

46536 Rodrigo Fernandes das Neves
a46536@alunos.isel.pt

Supervisor: Paulo Pereira
palbp@cc.isel.ipl.pt

Project report written for Project and Seminar 22/23
Computer Science and Engineering Bachelor's degree

July 2023

Abstract

According to the World Health Organization (WHO) [1], approximately one-fifth of the global population lives with musculoskeletal conditions causing chronic pain. However, there is a shortage of healthcare professionals available to provide continuous rehabilitation exercises required to prevent physical problems resulting from weakened and inflexible muscles. Additionally, the lack of physical activity and exercise contributes significantly to the prevalence of chronic diseases, which remain leading causes of death worldwide.

To address these challenges, the Intelligent Personalized Care project has been developed. Its primary objective is to combat the rising sedentary lifestyle and social isolation by leveraging advanced technologies to deliver personalized and effective remote care for patients across diverse health scenarios. The project utilizes an Android application that promotes physical agility and recovery through tele-exercise and tele-rehabilitation, incorporating immersive approaches facilitated by vision techniques and artificial intelligence. Moreover, the application offers comprehensive patient monitoring and follow-up functionalities, empowering physiotherapists to closely track and assess patient progress.

By combining human capabilities with cutting-edge technology, the Intelligent Personalized Care project aims to revolutionize prevention, treatment, and the overall quality of life. Through its innovative approach, it seeks to mitigate sedentary lifestyles, prevent physical complications, and promote active engagement in exercise and rehabilitation. This solution holds immense potential to bridge the gap between demand and available resources, enabling more individuals to access the care they need, while enhancing their overall well-being.

Contents

LIST OF FIGURES.....	7
1. INTRODUCTION.....	8
1.1. MOTIVATION	8
1.2. GOALS.....	8
1.3. REPORT STRUCTURE.....	9
2. FUNCTIONALITIES	10
3.ARCHITECTURE AND DATA MODEL	12
3.1. TECHNOLOGIES	13
3.2. DATA MODEL.....	14
4. SERVER IMPLEMENTATION	15
4.1. API	15
4.2. AUTHENTICATION AND AUTHORIZATION	15
4.3. HANDLERS	16
4.4. SERVICES.....	16
4.5. GOOGLE CLOUD STORAGE	16
4.6. RELATIONAL DATABASE	17
4.7. DEPLOYMENT OF THE API	17
4.8. ERROR HANDLING	18
4.9. SERVER-SENT EVENTS.....	18
5. CLIENT IMPLEMENTATION	19
5.1. JETPACK COMPOSE.....	20
5.1.1. VIEW.....	20
5.1.2. VIEW MODEL.....	20
5.1.3. MODEL	21
5.2. API ACCESS	21
5.3. SHARED PREFERENCES	22
5.4. WORKMANAGER	22
5.5. EXCEPTIONS HANDLING	23
5.6. INTERNATIONALIZATION	23
6.ML KIT POSE DETECTION	24
6.1. MODEL.....	24
6.2. EXERCISES.....	27
6.3. CAMERAX AND RECORD VIDEO	30

7. USER INTERFACE AND FUNCTIONALITIES	32
7.1. SPLASH SCREEN	33
7.2. AUTHENTICATION SCREENS	34
7.2.1. CHOOSE ROLE	34
7.2.2. SIGN UP	35
7.2.3. SIGN IN	36
7.3. CLIENT SCREENS	37
7.3.1. BOTTOM BAR	37
7.3.2. HOME	38
7.3.3. SEARCH MONITORS	39
7.3.4. MONITOR DETAILS	40
7.3.5. EXERCISES LIST	41
7.3.6. EXERCISE INFORMATION	42
7.3.7. CLIENT EXERCISE DONE	42
7.3.8. CAMERAX LIVE AND SETTINGS	43
7.3.9. PROFILE	45
7.4. MONITOR SCREENS	46
7.4.1. BOTTOM BAR	46
7.4.2. HOME	47
7.4.3. CLIENT DETAILS	48
7.4.4. CREATE PLAN	49
7.4.5. PLAN DETAILS	50
7.4.6. CLIENT EXERCISE	50
7.4.7. PROFILE	51
7.5. ABOUT SCREEN	52
8. CONCLUSION.....	53
8.1. FUTURE WORK	54
REFERENCES.....	55

List of Figures

Figure 1 – Project Architecture	12
Figure 2 – EA Model.....	14
Figure 3 – Diagram about pattern in MVVM.....	20
Figure 4 – Entities involved in streams of data	21
Figure 5 - 33 point skeletal match on human body.....	25
Figure 6 - ML Kit Pose Detection Pipeline	26
Figure 7 - Exercise Architecture with two Layers	27
Figure 8 - Exercise Logic Class.....	28
Figure 9 - Squat exercise logic and execution example.....	28
Figure 10 - Do Exercise Logic function	29
Figure 11 - Do Exercise Logic function continuation	29
Figure 12 - Conceptual Capturing System for the CameraX	30
Figure 13 - Use case for CameraX with VideoCapture	31
Figure 14 - Client Navigation.....	32
Figure 15 - Monitor Navigation	32
Figure 16 - Splash Screen.....	33
Figure 17 - Choose Role Screen	34
Figure 18 - Register Monitor Screen	35
Figure 19 - Register Client Screen	35
Figure 20 - Login Screen.....	36
Figure 21 - Client Bottom Bar.....	37
Figure 22 - Client's Home Screen without a monitor or plan.....	38
Figure 23 - Client's Home Screen with an associated monitor and plan	38
Figure 24 - Search Monitors Screen	39
Figure 25 - Monitor Details Screen: not the monitor of client	40
Figure 26 - Monitor Details Screen: monitor of client	40
Figure 27 - Exercises List Screen with exercise selected	41
Figure 28 - Exercises List Screen.....	41
Figure 29 - Client Exercise Done Screen	42
Figure 30 - Exercise Information Screen.....	42
Figure 31 - Use case for CameraX start.....	43
Figure 32 - Use case for Settings.....	43
Figure 33 - Use case for CameraX recording	44
Figure 34 - Use case for CameraX resting.....	44
Figure 35 - Use case for Client Profile screen	45
Figure 36 - Monitor Bottom Bar	46
Figure 37 - Monitor's Home Screen.....	47
Figure 38 - Monitor's Home Screen with requests.....	47
Figure 39 - Client Details Screen with list of plans.....	48
Figure 40 - Client Details Screen for associating a plan.....	48
Figure 41 - Create Plan Screen	49
Figure 42 - Create Plan Screen with the daily exercises.....	49
Figure 43 - Plan Details Screen	50
Figure 44 - Client Exercise Screen.....	50
Figure 45 - Use case for Monitor Profile screen.....	51
Figure 46 - Use case for Monitor Profile credential valid screen	51
Figure 47 - Use case for the About screen	52

Chapter 1

Introduction

The following chapter introduces the motivation behind the project, the goals, and the main functionalities.

1.1. Motivation

According to the WHO[1], about 20% of the world's population lives with a painful musculoskeletal condition. Continuous rehabilitation exercises are needed and there are not enough professionals to do this. It is necessary to avoid injuries and other physical problems that result from weakened and inflexible muscles. Keeping in mind that the lack of physical activity/exercise is one of the biggest causes of death, through chronic diseases.

There is a need to rethink prevention, treatment, and improvement of the quality of life. It is imperative to discover the best way to combine human capabilities with technology to ensure better services and experience for people.

1.2. Goals

Our *Intelligent Personalized Care* project aims to respond to the rapidly increasing sedentary lifestyle and isolation of the population. Using advanced vision technologies to provide personalized and effective remote care for patients in different musculoskeletal health scenarios, it will be possible to reduce and prevent sedentary lifestyle, promote physical exercise and rehabilitation in people.

This project is based on an Android application to promote physical agility and recovery, through (tele) exercise and (tele) rehabilitation using immersive approaches with vision techniques and artificial intelligence. In addition, the application will offer advanced patient monitoring and follow-up features, allowing physiotherapists to monitor the patient's progress.

1.3. Report Structure

This report is composed of 8 chapters, that will outline how the project architecture is assembled and the technologies used in it.

Chapter 2 lists the different functionalities within the project, and the different roles that exist.

Chapter 3 describes the project architecture concerning the server (backend) and the client (frontend), explaining in a grand manner how each of them work. In this chapter the different functionalities and roles will be explained with more detail as well as the technologies used, namely which were chosen and why they were chosen, it will also explain the database model built and used and how its entities relationships with one another.

Chapter 4 will describe in further detail the server-side implementation with an explanation about its structure and layers, the authentication and authorization process, the exception handling, and our working technique.

Chapter 5 will describe in further detail the client-side implementation, providing information about the Android Application and its components, as well as the requests to the API are made and how we handle with exceptions originated from the requests made. Also explains the authorization and authentication process in the client and it will introduce the artificial intelligence used in the project.

Chapter 6 describes the machine learning model used, and the techniques used to verify the specific exercises, as well as the recording of videos.

Chapter 7 shows the functional aspects of the project in a graphical way.

Chapter 8 reflects our thoughts and conclusions regarding the course of the project, as well as future work.

Chapter 2

Functionalities

The following section presents all functionalities that the application will be capable of executing. Graphical examples can be found in chapter 7.

The system supports three types of roles:

- **Monitor:** A monitor is a user that will help the client with his problem. A monitor can prescribe plans, submit his credentials, watch his clients' recordings and give feedback for the videos.
- **Client:** A client can make a connection request for a monitor, see the exercises that have been prescribed to him and film himself doing these exercises.
- **Admin:** This is a role that helps in maintaining the functionality of the Android Application. An Admin can see the credential of every unverified monitor and accept or refuse his credential. He can also submit a new exercise in our library. An Admin cannot see the videos of the users for privacy reasons. Only an Admin can create an Admin account. However, this role, currently, is not supported in the Android Application.

The functionalities of our application are:

- **Sign-in and Sign-up:** Non-Members can apply to be a client or a monitor, using the system verification procedures. Independently of the role, the name, email, and password must be provided. In the Sign-in a user must provide the email and password.
- **Creation of a Client:** In the creation of a client account, in addition to the credentials mentioned before, the weight, height, birthdate and his physical condition can also be provided. This is all to give the monitor a better understanding of the situation.
- **Creation of a Monitor:** To create an account a monitor only needs to give his email and password. However, this is not enough to start using the application. First must provide his credential of physiotherapist. It must wait that an Admin validates his credential and then has full access to all the functionalities of a monitor.

- **Client can see the available Monitors:** The client can search for a monitor by name and see his rating.
- **Upload profile picture:** Any user can upload his profile picture.
- **Connection request from the Client to the Monitor:** The client can send a connection request to the monitor with a connection text that gives more details of his problems.
- **Request Decision:** A monitor can decide whether to accept a request from a client or not. He can also see the message that comes with the request.
- **Creation of a plan:** A monitor can create a Plan. It must assign the exercises of our library for each day.
- **Assign a plan:** The monitor can assign a specific plan to a specific client. He can only assign plans created by himself and a client can only have one plan active at the same time.
- **Client uses the plan:** When a client has a plan associated, he can see the exercises of the day and record himself. Each set is a different video, and the client can give feedback about the exercise.
- **Exercises Information and preview:** A client can search for an exercise; its information and a video preview will be shown.
- **Application Feedback on video:** When a client is recording himself, if the exercises is *Squats*, *Push Ups* or *Shoulder Press* his reps and sets will be count and displayed in the screen. Messages will be also displayed according to the pose of the client, this way the user can know at real time if what he is doing is right.
- **Feedback from the Monitor:** The monitor can give feedback on a specific exercise done by one of his clients.

All these functionalities were mandatory according to the group, however there are a few additional features to be implemented in the future as described in the last chapter 8.

Chapter 3

Architecture and Data Model

The architecture of the developed system is described in this chapter, with its components illustrated in figure 1.

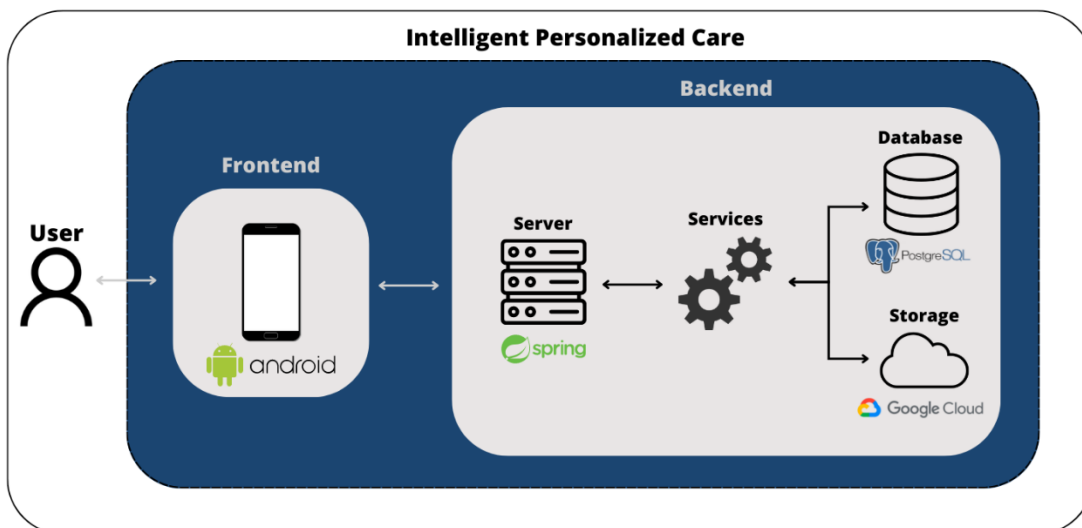


Figure 1 – Project Architecture

The system is divided into two parts, the backend, and the frontend. In the backend is where most of the system logic is, providing a HTTP API that gives a set of options/capacities to potential users and provides two different ways of storing data: a relational database to store every piece of data related to the system logic and a cloud storage to store images, videos, and documents.

In the other part of the system, the front end, is an android application that serves as a client to the backend application as well as the main factor in our usage of the artificial intelligence model.

Dividing our system in these two parts allowed us to have numerous users, in our mobile app and the users do not need to maintain knowledge about the system or its logic because this is all already available in the backend part.

The android application was chosen for the frontend part of this project because it was the most practical way of putting our project context into practice, a mobile app

which promotes remote and personalized musculoskeletal care. It also allows to have just one implementation for it to be compatible with most android devices.

3.1. Technologies

This section describes the technologies used in this application in both backend side and frontend side.

The server application, or backend is an application that exposes a HTTP API, and it was developed using the **Kotlin** programming language [2], has all the information regarding the API, relational database, storage, and services.

To store the relevant system information in the database we will use **PostgreSQL** [3], while to save the recorded videos and photos, the **Google Cloud Storage** service will be used. To deal with data from the relational database, we will use the **JDBI** [5] library, which will pass the information to the services. These will be exposed through an HTTP API, that is built using the **Spring** [6] framework.

The Android application consists of a frontend application that aims to design a representative and practical User Interface (UI) to access the resources made available in the backend.

As we are going to implement an Android application, we found the choice of the Kotlin programming language essential, as well as the **Jetpack Compose** [8] library for creating the UI. To deal with the HTTP requests and responses the **OkHttp** [9], library was used.

To process the visual data captured by the camera, the **ML Kit Pose Detection API** [9] is used. This sdk is a lightweight versatile solution for app developers to detect the pose of a subject's body in real time from a continuous video or static image. It gives access to a pre-trained model, that will be used to detect the pose of the client in real time and so we are able to give feedback while the client is doing the exercise.

The **cameraX** [9] Jetpack library is also used to help maintaining a consistent behaviour of the camera.

3.2. Data Model

The following section presents the graphical representation of the database in its Entity Association [14] format, describing the existing entities and the relations between them.

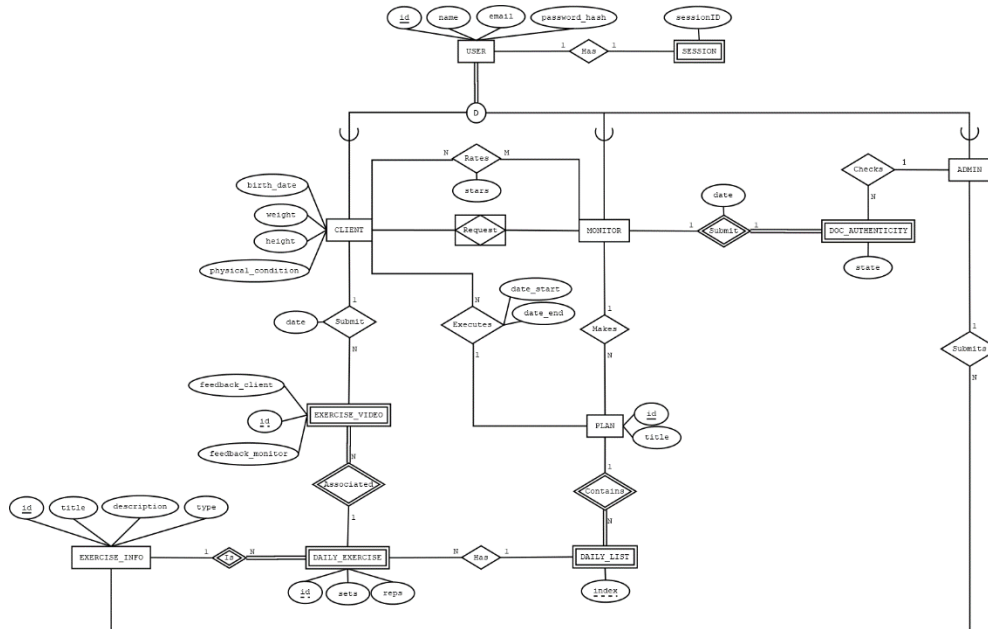


Figure 2 – EA Model

As shown in the figure 2, the **User** is the core entity, representing a user of the application, and this user can have three roles, **Client**, **Monitor** or **Admin**. Each **User** has a **Session** associated to him specifically so that the Authentication can be performed safely.

In the case of a **Monitor**, there is a need to submit his credential of physiotherapist to have access to the full functionalities of the application. This credential will be evaluated by the **Admin**.

A **Client** can make a connection request to a specific **Monitor**, as well as rate him with 1 to 5 stars.

The **Monitor** is also associated with an entity called **Plan**, where he can create plans, to then later associate with as many **Clients** as he wants. The only restriction is that the plan must be created by that **Monitor** and a **Client** can only have one plan at the same time. Each plan contains numerous **Daily_Lists**, which represents the list of exercises for that specific day, these exercises are represented by the entity **Daily_Exercise** and each of these exercises has an **Exercise_Info** with all the info associated to that exercise. The exercises differ in the **Daily_Exercise** entity where the “same” exercise could have different sets and reps.

Associated with each of these **Daily_Exercises** we’ve got an entity called **Exercise_Video** that has all the information about the video that is stored in **Google**

Cloud Storage. In this entity is stored the identifier of the video, the set that was filmed, the submit date and the feedback from both the client and the monitor.

Chapter 4

Server Implementation

In this chapter it will be described the server's implementation, each layers' mission and how they accomplish it.

4.1. API

Our Application Program Interface sets the rules and protocols that allow the communication and interaction between the Frontend and Backend. There are several options that were considered but, in the end, we decided that the combination of **Kotlin** with the **Spring** framework was the best. For the relational database it was chosen **PostgreSQL** with the Java library **JDBI**. The **Google Cloud Storage** service was used to store the videos of clients, credentials of monitors and the photos of users.

4.2. Authentication and Authorization

The user must **Sign up** in our application to use or services. Both Client and monitor must provide the name, email and password. The client can also provide his weight, height, birth date and is current physical conditional. This is all to help the monitor have a little more context about the problem of the client and make a better decision about the exercises that the client should do. The monitor is obligated to provide his credentials as a physiotherapist to provide more safety to the customers since that way no person can prescribe any exercises.

If the Signup is successful a pair of JWT's are returned. An accessToken and a refreshToken. The first has an expiration date of one hour and the latter of 1 day. The accessToken is used as authentication while the refreshToken is used to generate a new pair of JWTokens for when the accessToken expires. To impose the Authentication every request must have in the Authorization Header the accessToken.

JWT has been chosen since it is a safer way to transmit information between two parties (this case the Android Application and the Server). The usage of JWT is also useful for the Authorization since JWTs are self-contained, meaning the necessary information for any kind of verification that we chose to do is embedded in the token itself. In our case we store the id, role of the user and the session id. With this information

we must check whether there is a user with that role in that session, if not then the authentication failed. A new session is created every time that a user refreshes the tokens or Loges in. By applying this protocol, there is no need to store the tokens in the database. We decided that the JWT should be signed with the “HmacSHA512” algorithm.

4.3. Handlers

The handlers are the components responsible for handling the incoming requests and generating the responses. In our web API we use the `HandlerInterceptor` to intercept the requests before it reaches the Controllers. We have done this to impose Authentication and Authorization. We also use the injection of arguments in this class to have the user information when it reaches the Controllers.

All the Controllers are annotated with the **@RestController Annotation** since we decided to follow the REST Architecture. In the Handlers we call the **Services** where the logic of the API will be made.

We also have the need for an Exception Handler since we decided to throw our custom exceptions when there is an error. In this Handler the exception is captured and depending on the type of exception a Response is made. Because of that we have a Media Type “application/problem+json” that represents when an operation is not successful and carries the explanation for the problem in the response body.

4.4. Services

In the Services is where the Business Logic is implemented. To do this, requests to the Google Cloud Storage (GCS) and our PostgreSQL Database are made. We only request something from the GCS when we are storing or getting an object that was requested by the user. Before this request is made, we use the data stored in the Database to check the information about the user and the video/photo itself (ex: when a client is sending a video of an exercise, we first must check whether a video for that exercise and set was already uploaded or even if that exercise exists, between others). It orchestrates the flow of data and retrieves information from the Database/GCS. By encapsulating this logic within the Services layer, the application can ensure proper data handling and maintain the integrity of not only the system but the data itself.

4.5. Google Cloud Storage

Google Cloud Storage (GCS) is a scalable and durable object storage service provided by the Google Cloud Platform. It allows us to store and retrieve videos/photos in the form of `ByteArray` (in the context of our application, in another contexts other forms of objects can be stored). It’s design to be highly available, secure, and cost-

effective. The familiarity of Google Cloud Services and the deployment using the App Engine were also factors considered.

To differentiate the type of objects that are stored in the GCS, when making the request we specify the content type, if it's a video is *video/mp4*, if it is a photo, it's *image/png*, else when inserting the credential, the file it's stored as a PDF.

It's also useful to use this service because the App Engine uses Google Cloud Storage to store containers to launch instances and artifacts generated during the build phase.

4.6. Relational Database

To store information about the users, plans and exercises we decide to use a relational database. This decision was made because of the relationship that the users have between themselves and the plans. In the representation of a plan, we were faced with a problem of *impedance mismatch*, as the representation of a plan at the application level was a little hard to translate to the relational data model used in SQL databases. A solution was to store the plan in a No-SQL database, like Firestore, and the information about this plan in the PostgreSQL database. We decided against this solution because the trade-offs of maintaining three very different types of storage were bigger than the complication of passing an instance of a plan to the respective tables.

A major advantage of the relational databases is the Transactions. There was a need to define a Transaction Manager that considers that a Cloud Storage is not a database, therefore the concept of a Transaction does not exist. To do this every query made to the database is made inside a transaction, however if there is an error it's caught at application level and if a blob was inserted in the Google Cloud Storage is then deleted. This way we can ensure that the data is consistent between the SQL database and Cloud Storage.

For the deployment of the database, we continued to take advantage of the Google Cloud Services and decided to use the SQL service. The service Big Query was also used to query the database.

4.7. Deployment Of The API

For the deployment of the API the Google Cloud Service App Engine was used. App Engine is a fully managed, serverless platform for developing and hosting web applications at scale. App Engine can automatically create and shut down instances as traffic fluctuates, this way our server has the properties of scalability and elasticity.

To determine how and when new instances are created, we specify a *scaling type* for our app in the app.yaml file. Here we also specified the scaling settings that are applied to the App Engine. It was decided that if the CPU usage exceeds 65% then a new instance is created. When the request volume decreases, App Engine reduces the number of instances to make sure they all are being used for optimal efficiency and cost effective.

4.8. Error handling

Error handling is a crucial part of our application to ensure smooth operation and provide meaningful feedback to users. We use an error handling mechanism provided by Spring Boot called `@ControllerAdvice` combined with the `@ExceptionHandler`. In the `@ExceptionHandler` the exception that is being treated in that function is specified so this way we can generate appropriate responses for every kind of error. By using this exception handler, the application can intercept, and process exceptions throw during request processing. It is also very helpful to keep the code clean and readable by avoiding the use of try-catch blocks.

To represent an error at the HTTP level a media type “application/problem+json” is used. When the client receives a response with this media type, it knows that the operation was not successful and the explanation for this error is available in the response body.

4.9. Server-Sent Events

In this application, there are moments that we wish to have information in real time. One example of this is when the client sends a connection request to the monitor, the latter receives a notification with this information. To apply this mechanism a protocol has been set. When the client side pretends to receive update at real time it must subscribe using the “/users/subscribe”. In response the server returns a `SseEmitter` that will keep the connection open for sending events. In this response it goes the data but also the type of data that is being transmitted, so the client has prior knowledge of what events can and will be transmitted. In the server side a `ConcurrentHashMap` is used to keep track of the instances of `SseEmitter`. When the server must send any information to the client, it uses the instance of `SseEmitter` that is associated with that user and sends the event. This way we can push real time updates to the client side over a single long-lived HTTP connection.

Chapter 5

Client Implementation

The following chapter explains the implementation and thought process behind the client side, that is responsible for the visual element of the project.

Description of the architecture:

- **Activities:** Activities are responsible for managing the lifecycle of the application and providing the environment where screens are displayed. In the context of Jetpack Compose, activities host the Compose composer.
- **Screens:** Screens represent individual user interface representations in the application. Each screen is typically composed of one or more composable functions, which define the structure, layout, and visual interaction of the screen. A screen can display data, respond to events, and interact with the ViewModel.
- **ViewModels:** ViewModels are responsible for providing and managing the data required for screen display and interaction. They contain the business logic and are responsible for fetching, transforming, and providing data to the screens. ViewModels are observable and notify screens when data is updated.

The interaction between these components occurs as follows:

1. The Activity hosts the Jetpack Compose composer and manages the overall flow of the application.
2. Each Screen is implemented as a composition of elements using Jetpack Compose functions.
3. The screen interacts with the corresponding ViewModel to obtain the required data for screen display and updates.
4. The ViewModel fetches the necessary data through services, performs transformations and makes them available on the screen through observable properties.
5. The screen observes the ViewModel's properties and is automatically notified when the data is updated, causing the user interface to update as needed.
6. The screen can also send events to the ViewModel, such as button clicks or user interactions, which can trigger additional actions or updates to the data.

This architecture allows for a clear separation of responsibilities, facilitates component reusability, and promotes well-organized and testable code.

5.1. Jetpack Compose

We follow a well-known architecture called **MVVM** (Model-View-ViewModel). This architecture separates responsibilities into distinct layers, making code organization and maintenance easier.

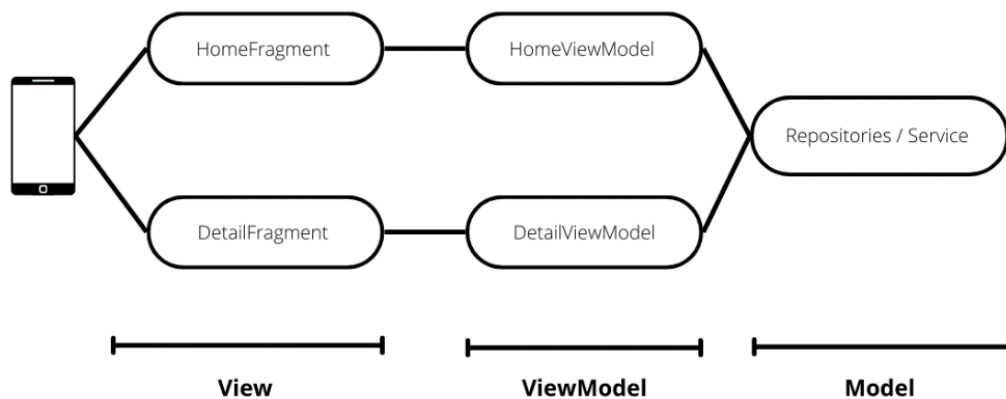


Figure 3 – Diagram about pattern in MVVM

5.1.1. View

The View is responsible for presenting the user interface to the end user. It displays the visual elements and interacts with the user, collecting input and responding to user actions. In MVVM, the View is typically implemented as an activity, fragment, or other UI component. It communicates with the ViewModel to retrieve data and update the UI.

5.1.2. View Model

The ViewModel acts as a mediator between the View and the Model. It contains the presentation logic and holds the state of the View. The ViewModel receives input from the View, processes it, and interacts with the Model to fetch or update data. It prepares the data in a format that the View can easily consume. The ViewModel is independent of the View, meaning it can be unit tested without the need for UI components.

We are using Flows to save all kinds of data that arrive in the request to the server.

Why Flows?

A Flow is a type that can emit multiple values sequentially, as opposed to suspend functions that return only a single value.

Using Flows in ViewModels has several benefits, offering a powerful and reactive way to handle asynchronous data, manage state, compose, and transform data streams. This results in more robust, flexible, and responsive ViewModels, providing a better development experience and a more responsive UI for users.

Each time a given StateFlow has a new value, it will trigger an update to Flow. In this way, it is possible to force recompositions on the screens.

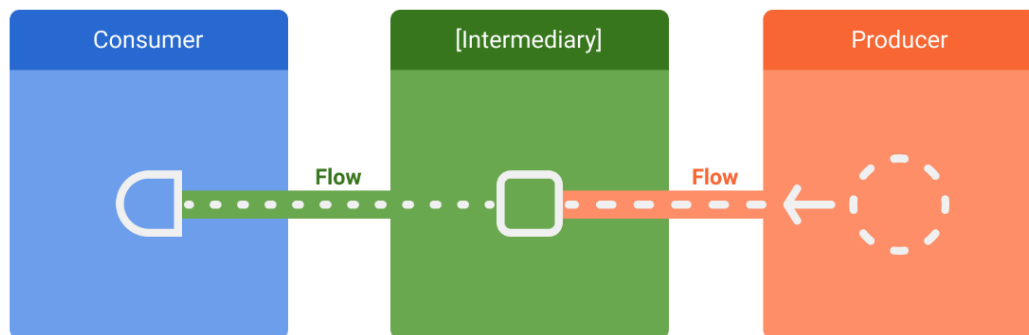


Figure 4 – Entities involved in streams of data

5.1.3. Model

The Model represents the underlying data and business logic of the application. It encapsulates the data sources, such as databases, APIs, or local storage, and contains the business rules and algorithms. The Model is responsible for fetching, manipulating, and persisting data. It provides the necessary data to the ViewModel, which then updates the View accordingly.

5.2. API access

We are using a popular open-source HTTP client library for Android apps – *OkHttp*. Provides efficient communication with RESTful APIs, retrieves data, sends requests with multiple parameters, and handles responses asynchronously. *OkHttp* simplifies the process of API integration by providing a reliable and efficient networking solution for Android applications.

5.3. Shared Preferences

We are using Android's `SharedPreferences` to store some user information when creating an account or logging in.

`SharedPreferences` is an Android framework class that provides a way to store and retrieve key-value pairs of primitive data types. It's commonly used for storing small amounts of application data that need to persist across app launches or device restarts.

5.4. WorkManager

WorkManager provides a reliable and flexible API for scheduling and executing tasks in the background. It ensures that scheduled work continues to run even if the app is closed, or the device is rebooted. This makes it ideal for handling tasks that require long running or periodic execution, such as data synchronization, database updates, or network requests.

The application is using 2 Workers:

- **RefreshTokensPeriodicWorker:** Performs a *PeriodicWorkRequest* job. Is responsible for periodically refreshing authentication tokens in the background, ensuring that the user remains authenticated and able to make secure API calls in the application. A repetition interval of 50 minutes was defined for this periodic execution, since, by the server, the user's access token is only valid for 1 hour.
- **VideoSubmissionOneTimeWorker:** Performs a *OneTimeWorkRequest* job. Is responsible for execute the task of submitting a video exercise for a one-time submission in the background. In this way, it is possible for the user to record his exercises even without a network connection, and then, when there is one, the worker will send the videos in the background.

Both Workers have the constraint of needing network connection to make the request to the server. If there is no connection when they are going to do the task, they wait for the user to connect to the network.

5.5. Exceptions handling

For errors that arrive in a response from the server, we throw an `APIException` exception, which has a property of type `ProblemJson`. This exception is caught in the `AppViewModel` (View Model that we created to use in all View Models in the application) and updates a `Flow` variable with the new value. As all View Models have access to this variable, a function was created that checks the state of this `Flow`, and if there is an update, it is detected and a `Composable` is built to show this error in the form of an `AlertDialog()`.

We can also detect whether the user has a network connection or not. If he doesn't have it and wants to make a request to the server, we inform him in the same way as before.

5.6. Internationalization

So that the application has a broader prospect of future users, internationalization was implemented, having now both English and Portuguese.

However, it's important to note that all types of data stored in the SQL database do not support languages other than English, such as exercise names and descriptions.

Chapter 6

ML Kit Pose Detection

The android application static machine learning Model, the vision techniques and the logic used to represent the exercises will be presented in this chapter.

6.1. Model

The **ML Kit Pose Detection** is an on-device, cross platform (Android and iOS), lightweight solution that tracks a subject's physical actions in real time. We used it to build an application that gives a great and easier experience for users.

The base of this API is built on the **BlazePose** [15] pipeline which allows developers to build great solutions with reduced effort and doesn't require ML (Machine Learning) expertise.

The **ML Kit Pose Detection** API utilizes a two way step process for detecting poses:

- Firstly, it combines an ultra-fast face detector with a prominent person detection algorithm, in order to detect when a person has entered the scene. The API is capable of detecting a single (highest confidence) person in the scene and requires the face of the user to be present in order to ensure optimal results. If the face of the user is not present nor the full body the API is not capable of ensuring good results in terms of tracking the landmark points correctly and its coordinates might return incorrect results.
- Secondly, the API produces a **full body 33 point** skeletal match to the detected person, as seen in figure 5, that includes facial landmarks (ears, eyes, mouth, and nose), along with hands and feet tracking. These points are rendered in 2D space and do not account for depth. The API also contains a streaming mode option for further performance and latency optimization. When enabled, instead of running person detection on every frame, the API only runs this detector when the previous frame no longer detects a pose. We disabled this option and used only the operating mode described below, where we detect a person at all times/ in every frame.

The operating mode used in this application was the **“Fast”** mode where we expect a higher frame rate and a faster detection of the person, the other operating mode **“Accurate”** wasn’t used because of its improved accuracy the frame rate would decrease, making the application slower and less user-friendly.

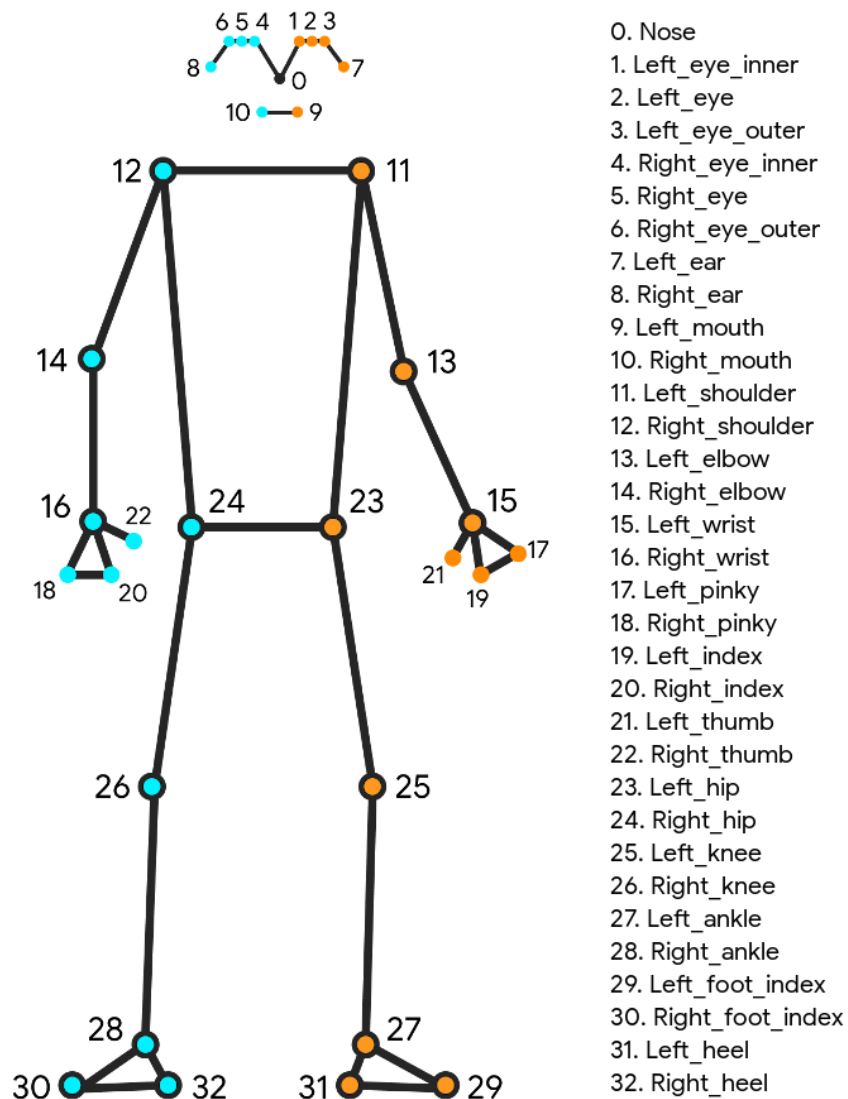


Figure 5 - 33 point skeletal match on human body

With the help of the camera and video input, the following figure 6 represents the algorithm to find and detect a person pose (landmark detection and pose estimation).

When a person is detected, we have access to a “*InFrameLikelihood*” score per landmark point to help verify if a user is in frame. This score is calculated during the landmark detection phase and a low likelihood score suggests that a landmark is outside the image frame.

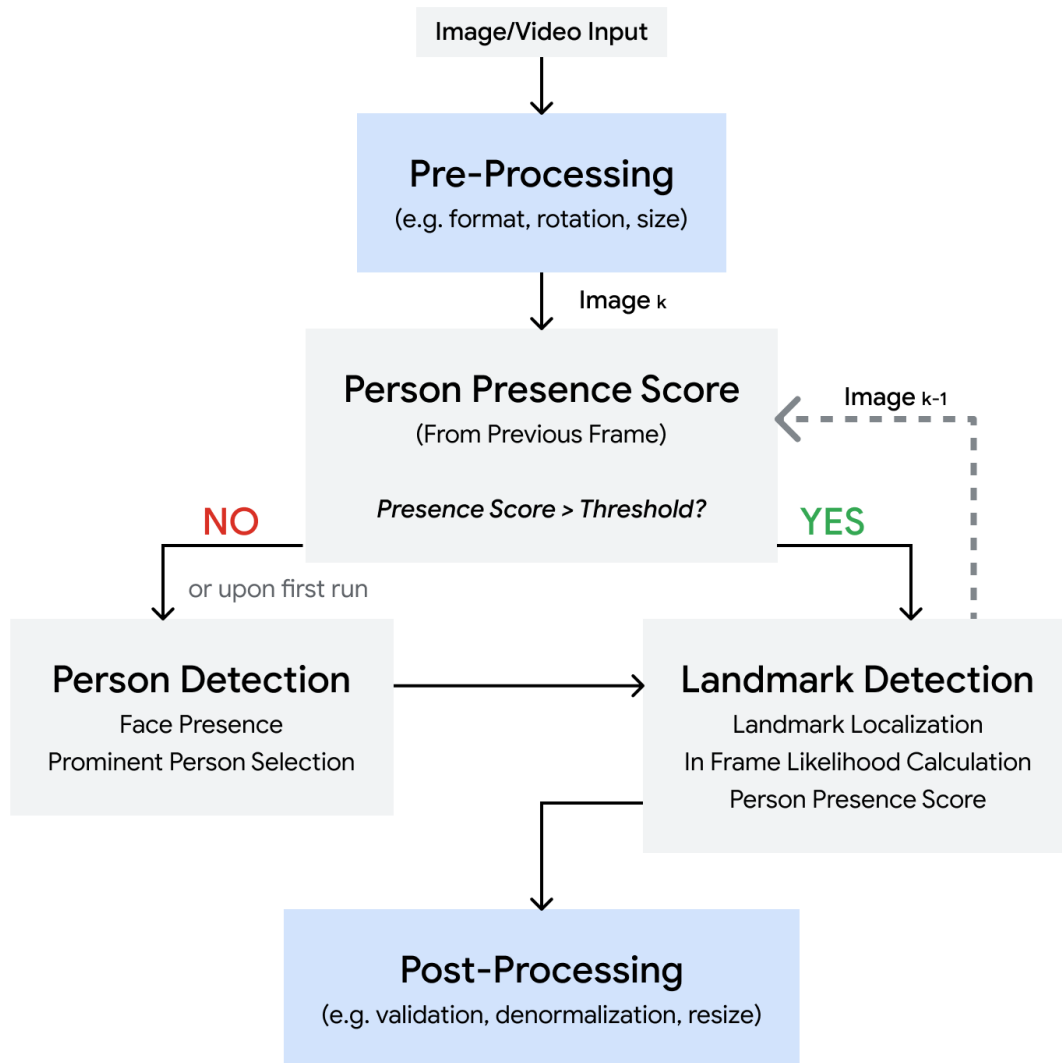


Figure 6 - ML Kit Pose Detection Pipeline

We need to highlight that in this project the use of the **ML KIT Pose Detection machine learning model** was merely static, meaning that we used the model already built with the necessary artificial intelligence techniques to detect the presence of a person and draw the 33 point skeletal match on the human body and as we didn't use the learning ability of the model to classify new poses.

We decided to use only the vision techniques and landmarks points on the body, which allowed us to create an algorithm to classify our own exercises with specific poses, using the coordinates (x and y as its only 2D) and the joint angles.

6.2. Exercises

As said before we made our own algorithm to classify the different exercises in our library. As proof of concept, we've applied this algorithm to three different exercises:

- Squat
- Push up
- Shoulder Press

each one with different needs.

To guarantee that the physiotherapist can define the exercise logic without knowing software engineering we've created two independent layers. The definition of these two layers creates an abstraction, that allows us to separate the definition of the exercise from the execution.

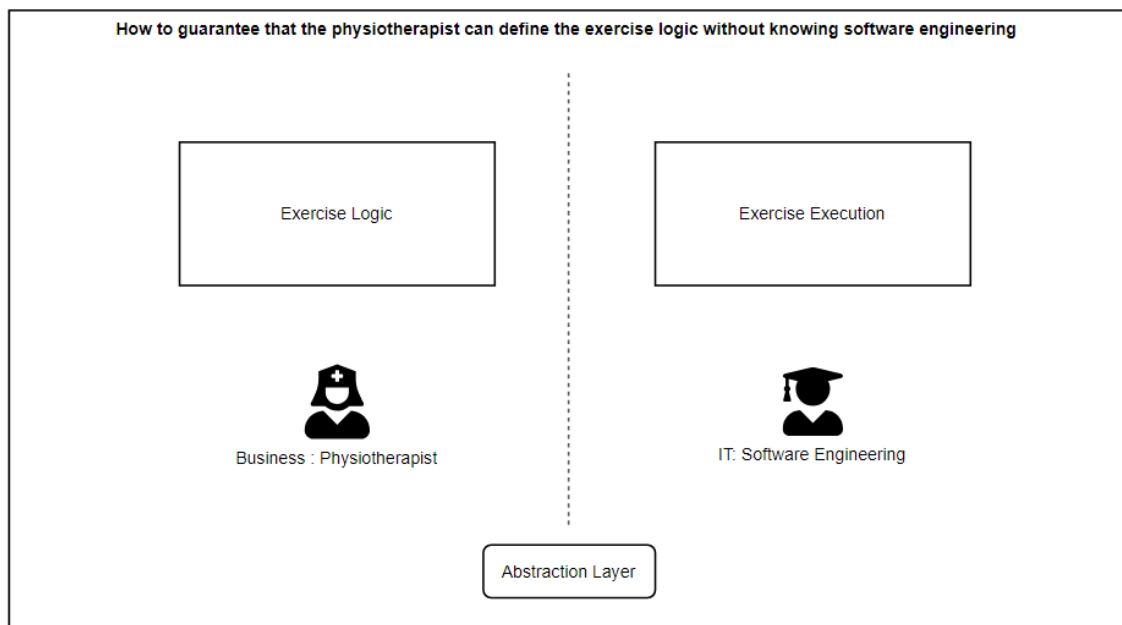


Figure 7 - Exercise Architecture with two Layers

- **The Exercise logic** definition part, as seen on figure 7, 8 and 9, where we define the angle, landmark points and conditions necessary for the execution of the specific exercise allowing the creation of multiple exercises using this method, each one with their own conditions angles and points.
- **The Execution** of the defined exercise logic part, as seen on figures 9,10 and 11 where we merely execute the logic defined previously for any exercise.

As explained in chapter 8.1 (Future work) the idea was to have a screen where the monitor could input an exercise with more detailed conditions (Used to create the exercise logic) to personalize a specific exercise to a client with a specific problem or limitation.

```

/**
 * Class to represent different exercises
 */
class ExerciseLogic(
    val firstPoint: PoseLandmark,
    val midPoint: PoseLandmark,
    val lastPoint: PoseLandmark,
    val rightHandPos: Float,
    val leftHandPos: Float,
    val ratio: Float,
    val currentHeight: Float,
    val minSize: Float,
    //conditions
    val condOne: Int,
    val condTwo: Int = 0,
    val condThree: Double = 0.5,
    val condCurrentWeight: Int?,
    val condMinSize: Int?,
    //line text of conditions
    val lTextCondOne: String,
    val lTextCondTwo: String,
    val lTextCondThree: String
)

```

Figure 8 - Exercise Logic Class

```

//Calculate whether the distance between the shoulder and the foot is the same width
val ratio = ratio(leftShoulder!!.position.x, rightShoulder!!.position.x, leftAnkle!!.position.x, rightAnkle!!.position.x)

//to divide each exercise logic
when(exercise.exeTitle){
    "Squats" -> {
        //Calculate whether the hand exceeds the shoulder
        val yRightHand = differenceBetweenCoordinates(rightWrist!!.position.y, rightShoulder.position.y)
        val yLeftHand = differenceBetweenCoordinates(leftWrist!!.position.y, leftShoulder.position.y)

        //defining specific exercise logic
        val exerciseLogic = ExerciseLogic(rightHip!!,rightKnee!!,rightAnkle,yRightHand,yLeftHand,ratio,
            currentHeight: rightShoulder.position.y + leftShoulder.position.y, minSize: rightAnkle.position.y - rightHip.position.y,
            condOne: 5, condTwo: 0, condThree: 0.5,
            condCurrentWeight: 2, condMinSize: 5,
            lTextCondOne: "Please stand up straight", lTextCondTwo: "Please hold your hands behind your head",
            lTextCondThree: "Please spread your feet shoulder-width apart")

        doExerciseLogic(exerciseLogic)
    }
}

```

Exercise Logic

Exercise Execution

toDraw = !toDraw

Figure 9 - Squat exercise logic and execution example

```

/**
 * Implements the exercise logic
 */
@ 47531
private fun doExerciseLogic(exerciseLogic: ExerciseLogic){

    val angle = getAngle(exerciseLogic.firstPoint, exerciseLogic.midPoint, exerciseLogic.lastPoint)

    if (((180 - abs(angle)) > exerciseLogic.condOne) && !isCount) {
        reInitParams()
        lineOneText = exerciseLogic.lTextCondOne
    } else if (exerciseLogic.leftHandPos > exerciseLogic.condTwo || exerciseLogic.rightHandPos > exerciseLogic.condTwo) {
        reInitParams()
        lineOneText = exerciseLogic.lTextCondTwo
    } else if (exerciseLogic.ratio < exerciseLogic.condThree && !isCount) {
        reInitParams()
        lineOneText = exerciseLogic.lTextCondThree
    } else {
        val currentHeight =
            if(exerciseLogic.condCurrentWeight != null) exerciseLogic.currentHeight / exerciseLogic.condCurrentWeight //Judging up and down
            else exerciseLogic.currentHeight
    }
}

```

Figure 10 - Do Exercise Logic function

```

    if (!isCount) {
        shoulderHeight = currentHeight
        minSize = exerciseLogic.condMinSize?.let { exerciseLogic.minSize / it } ?: exerciseLogic.minSize
        isCount = true
        lastHeight = currentHeight
        lineOneText = "Gesture ready, control each rep"
    }
    if (!isDown && (currentHeight - lastHeight) > minSize) {
        isDown = true
        isUp = false
        downCount++
        lastHeight = currentHeight
        lineTwoText = "start down"
    } else if ((currentHeight - lastHeight) > minSize) {
        lineTwoText = "downing"
        lastHeight = currentHeight
    }
    if (!isUp && (upCount < downCount) && (lastHeight - currentHeight) > minSize) {
        isUp = true
        isDown = false
        upCount++
        lastHeight = currentHeight
        lineTwoText = "start up"
    } else if ((lastHeight - currentHeight) > minSize) {
        lineTwoText = "upping"
        lastHeight = currentHeight
    }
}
}

```

Figure 11 - Do Exercise Logic function continuation

The **doExerciseLogic** function receives the created instance of any exercise to do, and then applies the conditions defined previously in the exercise logic layer. Starts by checking if the client is in the right position using the angle, the points and the conditions, if not it will present some information on the screen warning the client to put himself in the right position. When in the right position we will judge the height of the client according to the maximum and minimum size conditions received and proceed to start counting repetitions.

The exercise repetitions defined in the plan associated with the specific client, are just a recommendation or a goal designated by the monitor, as we can have clients that have certain limitations or are simply not strong enough to complete the number of repetitions established. The completion of a set of repetitions will be inferred by the start and stop recording actions talked in the next sub-chapter.

6.3. CameraX and record video

We used the **CameraX VideoCapture** API [16] to capture and record videos of the user performing the exercises. Below in the figure 12 we have a base and conceptual capturing system which uses a video recorder and an audio record to record the video and audio streams respectively, does its multiplexing and compression operations and save the result in the local storage.

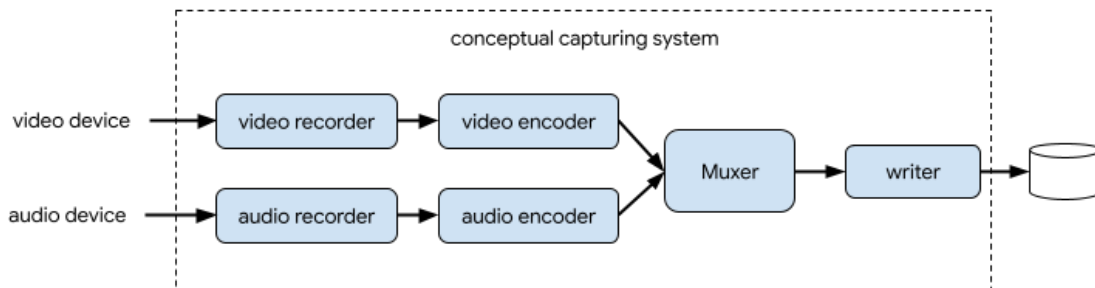


Figure 12 - Conceptual Capturing System for the CameraX

In our specific case we used the **VideoCapture** API, as seen below in figure 13, where we need permission to use the camera, microphone and access to the internal storage.

After that we need to create a **VideoCapture** instance, and a **SurfaceProvider** to the video origin, and finally an output (recorder) that has the process of multiplexing and

compressing the video and audio streams operations and save its result in the internal/local storage.

In this project as we use the *Google Cloud Storage* platform, when stopping the recording of a video while it is getting saved locally, we also send it to the server with a worker (explained on chapter 5.4), that can with a valid internet connection sends immediately the video, and in the case where a user doesn't have internet connection or lost it, the video recorded will be sent immediately after regaining the connection.

This API abstracts the complexity of the capturing mechanisms and gives us a straightforward and simple API for video capturing.

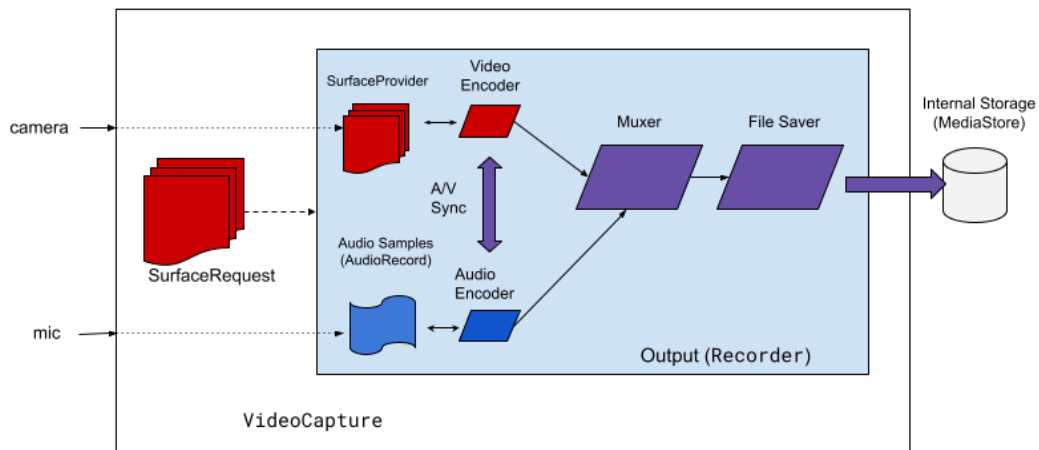


Figure 13 - Use case for CameraX with VideoCapture

The recorded video action can be divided into two parts:

- **The start recording action**, when clicked on the recording button activates this action and starts a timer with the time passed since clicked.
- **The stop recording action**, when clicked again on the same recording button activates this action where it stops the timer and shows a countdown timer for the resting time, as well as presents a message telling the client to rest.

After one **recorded video action** is completed, the video is sent to the local storage and the cloud storage explained previously and increases the sets made. We control the sets made with this action as the client can decide when to stop the video and the repetitions to do are only a recommended number (explained in the previous sub-chapter Exercises).

Chapter 7

User Interface and Functionalities

The android application graphical user interface (UI) will be presented in this part. The views below were created with the goal of delivering information as practical and user friendly as possible.

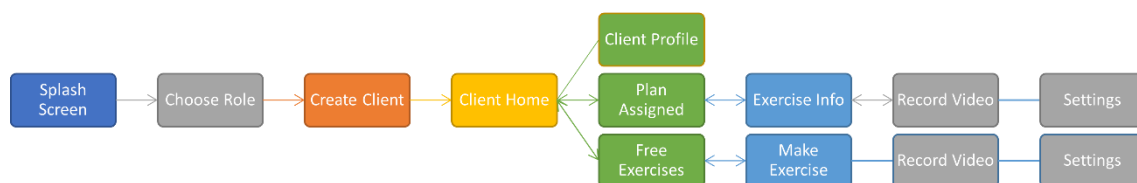


Figure 14 - Client Navigation

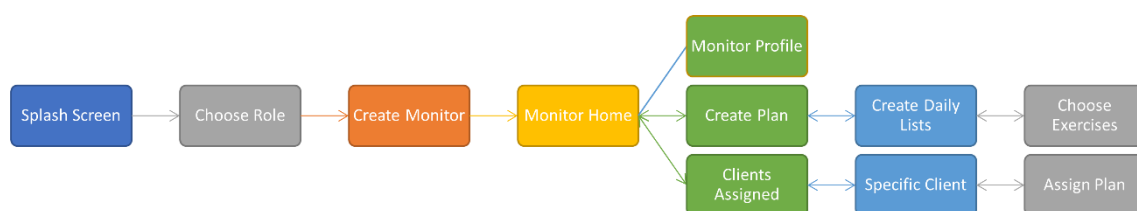


Figure 15 - Monitor Navigation

7.1. Splash Screen

The application's Splash Screen is represented by the figure 16. This is the initial screen that appears when the application starts. After a delay of a specified time (we chose 3 seconds), the application redirects the user to the appropriate screen based on the user's role.

This screen serves to enhance the appearance of the application, generating a positive impact on the users.

It appears while the application is loading and does work in the background so that the necessary data is immediately displayed on the screen without the user having to wait.

Here it is checked whether the user is logged in or not. If not, he is simply redirected to the Choose Role Screen where they can create an account or login.

If the user is logged in, his role is checked. If it's a client, a request is made to the server to request the monitor associated with it, along with the current daily exercise plan, if the client has one. Otherwise, if it's a monitor, a request is sent to the server to receive clients associated with that monitor and requests from clients that want to connect to him.

At the end of time, they will be redirected to Client Home Screen and Monitor Home Screen, respectively.



Figure 16 - Splash Screen

7.2. Authentication Screens

The screens in this section are only for users who are not already logged in on their mobile device.

7.2.1. Choose Role

This screen is responsible for allowing the user to choose whether he wants to be a Client or a Monitor. The user also has the possibility of already having an account created and, in that case, he only needs to log in by clicking on Login.

When a role is selected, the code navigates to the Sign Up Screen, passing the selected role's name as a parameter.

In case login is clicked, the user is redirected to the Sign In Screen.

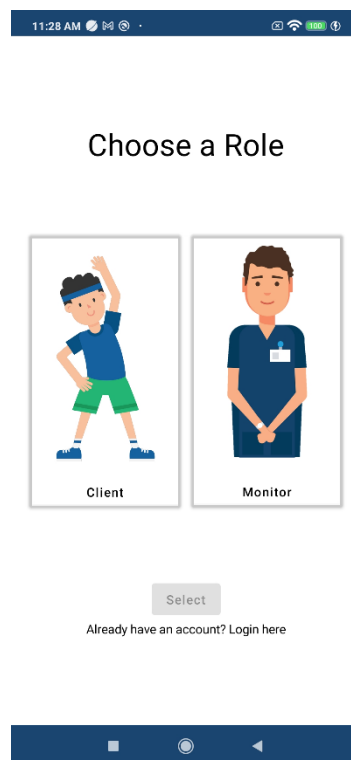


Figure 17 - Choose Role Screen

7.2.2. Sign Up

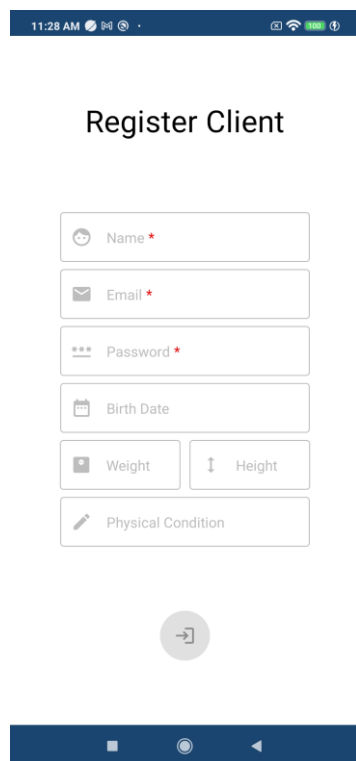
This screen is responsible for user registration. Depending on the selected role in Choose Role Screen, the specific screen, Register Client or Register Monitor, will be displayed.

When creating any of the roles, 3 parameters are mandatory: Name, Email and Password.

In the case of the client, he can provide more information about himself, such as his date of birth, weight, height, and his physical condition (if any). In this way, a monitor will be able to assign a more personalized plan according to the client's data.

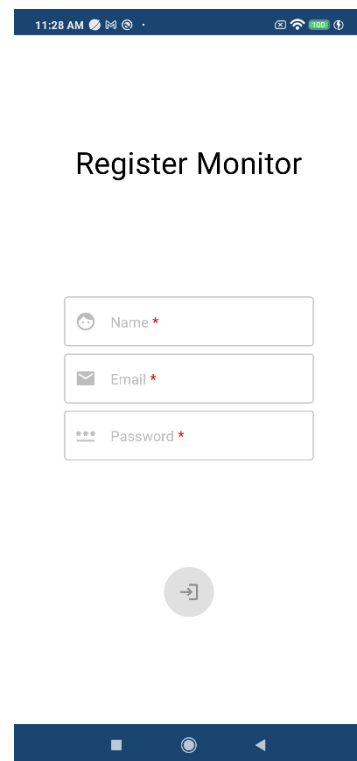
In the case of the monitor, although this data is only requested when creating the account, he will have to submit a document that proves he has the necessary qualifications to monitor a client, such as the credential of a physiotherapist.

After sending the request to the server and it returns a successful response, the user is redirected to his home screen and a session is created on the device to store certain essential user information, such as his id, username, access token, refresh token and his role (Client or Monitor).



The image shows a mobile app screen titled "Register Client". At the top, there is a status bar with the time "11:28 AM" and various icons. Below the title, there are several input fields: "Name" with a red asterisk, "Email" with a red asterisk, "Password" with three asterisks and a red asterisk, "Birth Date" with a calendar icon, "Weight" with a scale icon, "Height" with a height icon, and "Physical Condition" with a pencil icon. At the bottom, there is a large circular button with a right-pointing arrow. The bottom of the screen features a dark blue navigation bar with three icons: a square, a circle, and a triangle.

Figure 19 - Register Client Screen



The image shows a mobile app screen titled "Register Monitor". At the top, there is a status bar with the time "11:28 AM" and various icons. Below the title, there are three input fields: "Name" with a red asterisk, "Email" with a red asterisk, and "Password" with three asterisks and a red asterisk. At the bottom, there is a large circular button with a right-pointing arrow. The bottom of the screen features a dark blue navigation bar with three icons: a square, a circle, and a triangle.

Figure 18 - Register Monitor Screen

7.2.3. Sign In

This screen is responsible for logging in a user.

It's only necessary to provide the email and password as access credentials. Then, a request is sent to the server and if the data match, a successful response is returned. After receiving the necessary user information data from the server, a session is created on the device, and the user is redirected to his home screen.

If a user who is logged in loses authentication for any reason, he will be notified and redirected back to this screen.

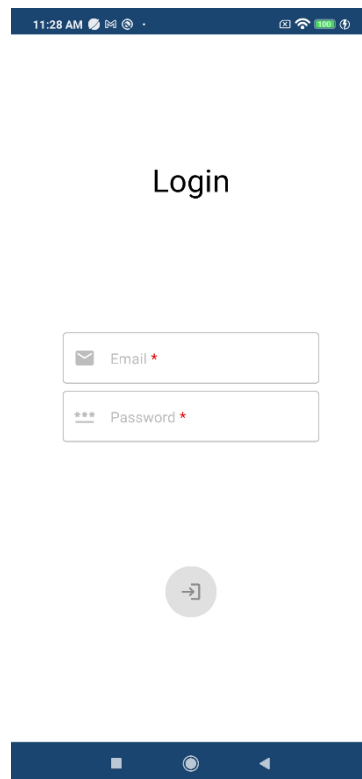


Figure 20 - Login Screen

7.3. Client Screens

The screens in this section are only accessible by clients.

7.3.1. Bottom Bar

The **Client bottom bar** is a nav bar present in the bottom of every screen, excluding the CameraX live Screen, where the client is logged in.

It's divided into three parts:

- Client home
- Exercises List
- Client's profile

As seen in figure 21



Figure 21 - Client Bottom Bar

7.3.2. Home

This is the first screen that a client sees when he opens the application. It's responsible for managing the day to day of a client.

Initially a client doesn't have an associated monitor, as shown in figure 11, so he must make a request to connect to a particular monitor. For that he will have to click on the respective button, and will be redirected to the Search Monitors Screen, where he can search for monitors that are available.

Below the monitor is the area reserved for controlling the client's current plan. He can see his daily exercises and if wants to do one, he can click on it. Then the client will navigate to the Exercise Information Screen, if it has not yet been done, if it has already been done it will go to Client Exercise Done Screen.

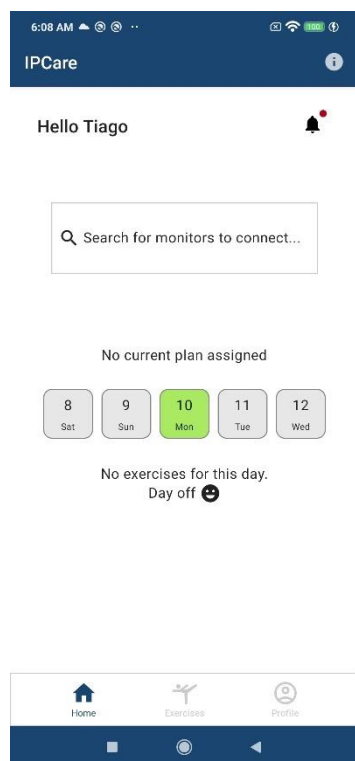


Figure 22 - Client's Home Screen without a monitor or plan

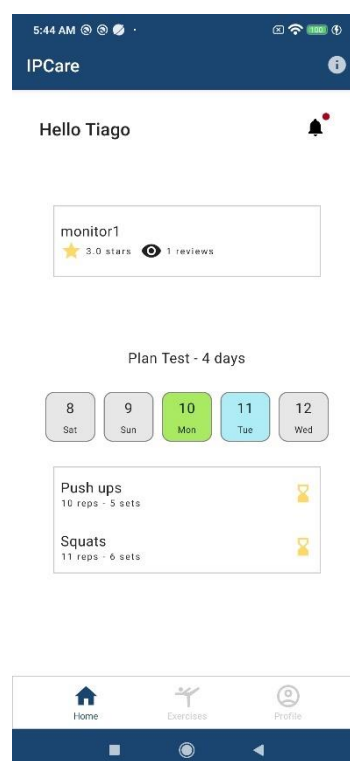


Figure 23 - Client's Home Screen with an associated monitor and plan

7.3.3. Search Monitors

This screen is for the client to be able to search for monitors to connect. This search is carried out by name and only validated monitors appear in the list, those who have already gone through the validation process of their document.

When clicking on a monitor, the client will navigate to the Monitor Details Screen, where he will be able to make the connection request there.

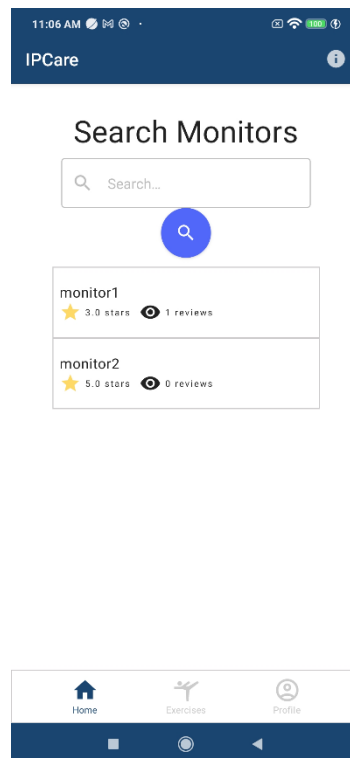


Figure 24 - Search Monitors Screen

7.3.4. Monitor Details

On this screen, the client can see the details of a specific monitor.

If this monitor is the client's monitor, the client will be able to evaluate the experience he had with it only once. This will contribute to the rating of that monitor.

Otherwise, the client is looking for monitors, and therefore, it can send a connection request with a small message to better explain the situation in which it finds itself.

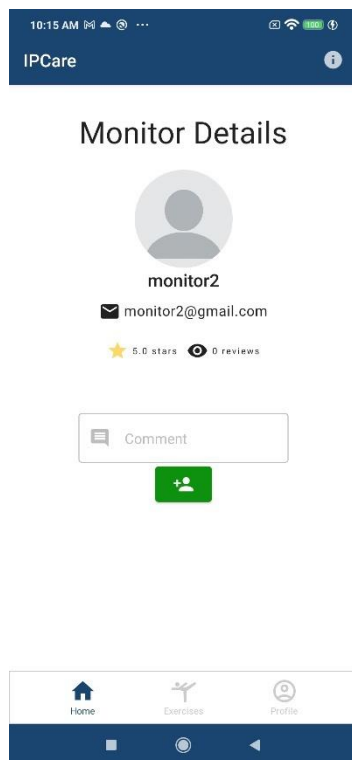


Figure 25 - Monitor Details Screen: not the monitor of client

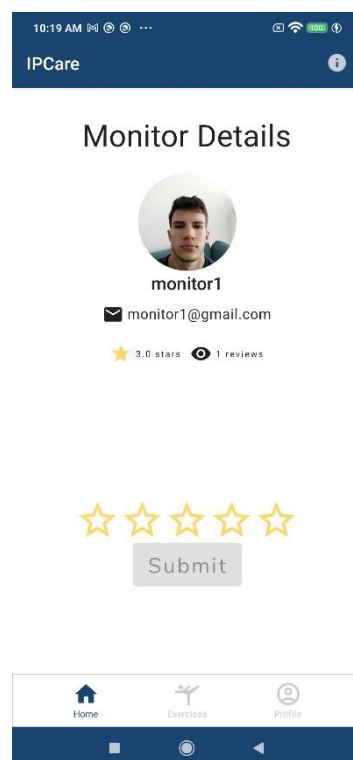


Figure 26 - Monitor Details Screen: monitor of client

7.3.5. Exercises List

This screen displays the entire library of possible exercises for the application to do. Pagination is accepted, as we can see from the screens below.

These are considered Free Exercises, as no client needs to be associated with a monitor to do them.

Setting the number of sets and reps is only for the purpose of the client to influence doing the exercise and we can complete the exercise when it reaches a certain number of sets.

If the client clicks on the "+" it will navigate to the Exercise Information Screen.

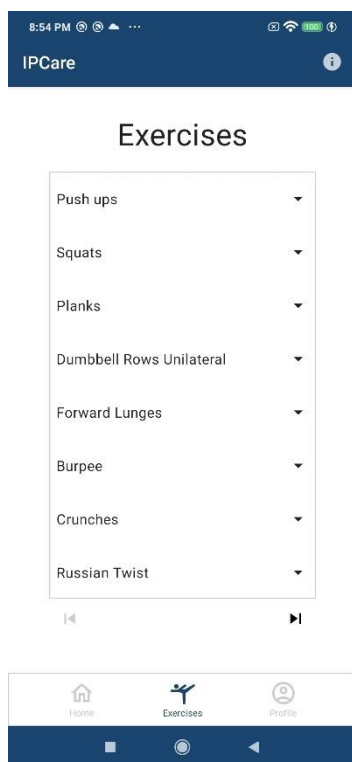


Figure 28 - Exercises List Screen

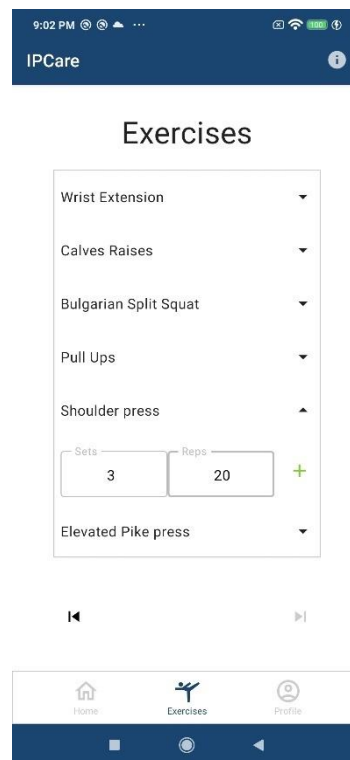


Figure 27 - Exercises List Screen
with exercise selected

7.3.6. Exercise Information

This screen is intended to show the information of a certain exercise, its title, its description and a preview for the client to know how to perform it.

The customer can then proceed with the recording by clicking on the "Record Video" button, which will navigate to the CameraXLive Screen.

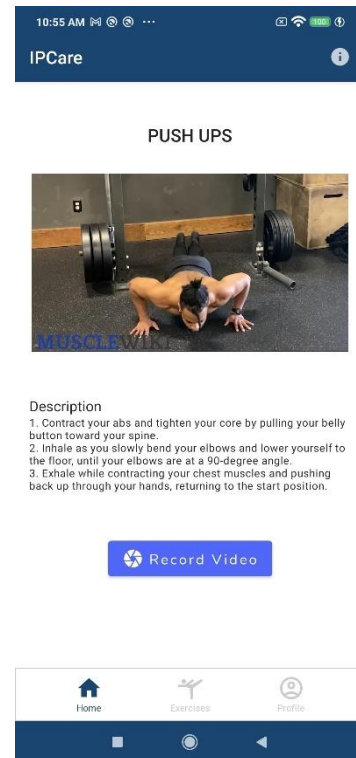


Figure 30 - Exercise Information Screen

7.3.7. Client Exercise Done

When a client finishes an exercise, it's possible to see the videos of each set of that exercise through this screen.

He can also see the feedback that the monitor has given to the videos, which in this case it hasn't yet.

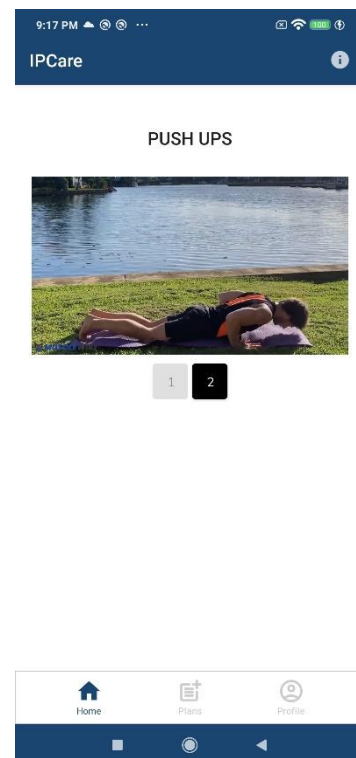


Figure 29 - Client Exercise Done Screen

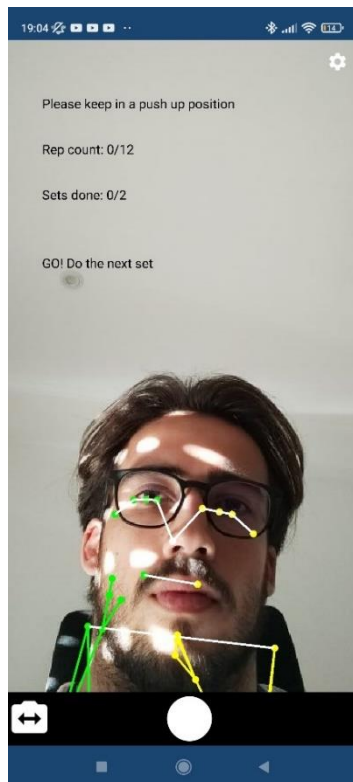
7.3.8. CameraX Live and Settings

The **CameraX live screen** will present the information regarding the pose in the form of a skeletal match with 33 points always updating in every frame as explained in chapter 6.1 Model. As well as the information about the pose, the recommended repetitions to do, sets to do and done, an increasing recording timer and a countdown timer to present the rest time. Changes the timers with the start and stop recording actions as seen in the figures 31,33 and 34.

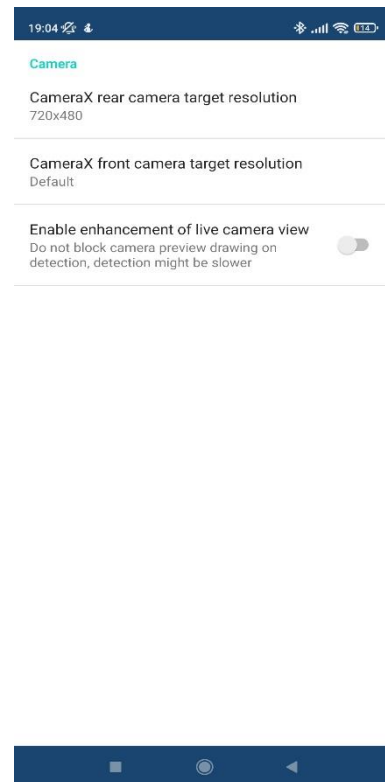
The **Settings screen** presents 3 options:

- Target resolution of the back camera
- Target resolution of the front camera
- Enhancement of camera quality, but makes the detection slower

As seen in figure 32.



*Figure 31 - Use case for
CameraX start*



*Figure 32 - Use case for
Settings*

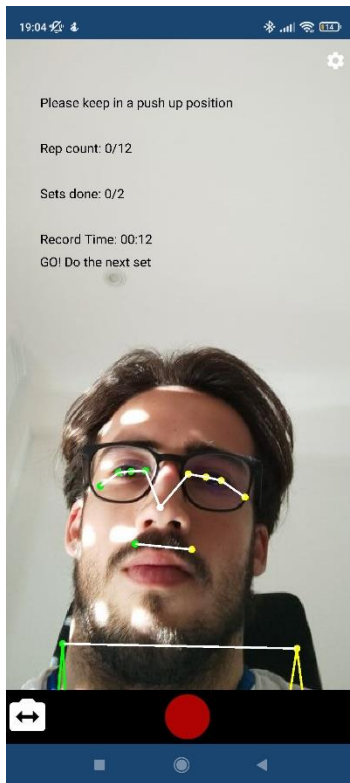


Figure 33 - Use case for CameraX recording

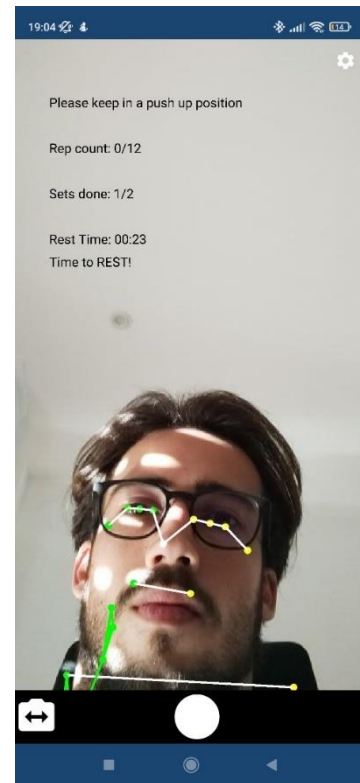


Figure 34 - Use case for CameraX resting

7.3.9. Profile

The **Client profile screen** presents information about the client:

- A photo of the client which can be altered
- Its name
- Its email address
- Birth Date
- Weight
- Height
- And a small explanation about his problem

It also has the option to logout out of the application, as seen in figure 35.

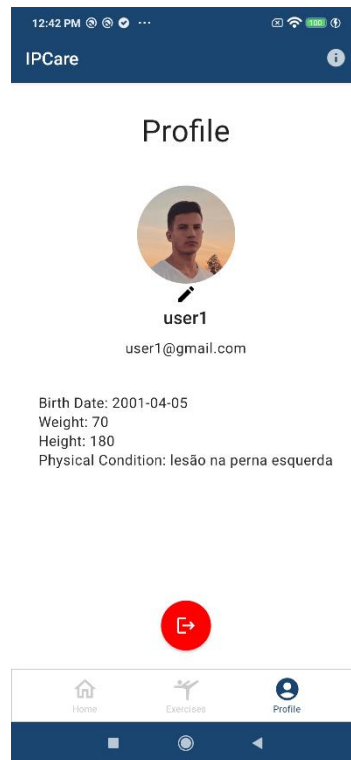


Figure 35 - Use case for Client Profile screen

7.4. Monitor Screens

The screens in this section are only accessible by monitors.

7.4.1. Bottom Bar

The **Monitor bottom bar** is a nav bar present in the bottom of every screen, where the monitor is logged in.

It's divided into three parts:

- Monitor home, figure do monitor home
- Create a Plan, figure
- Monitor's profile, figure

As seen in figure 36.

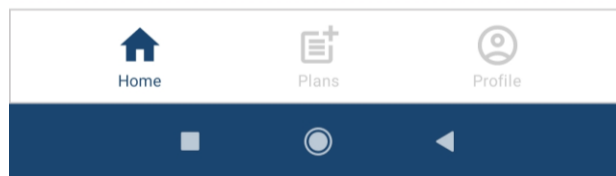


Figure 36 - Monitor Bottom Bar

7.4.2. Home

This is the first screen that a monitor sees when he opens the application. It's responsible for managing the day to day of a monitor.

When a client wants to connect to a monitor, this request will land on his notifications, where the monitor can accept or reject it. He can also see a message that the client has written at the time of the request, as shown in figure 37.

The clients associated with the monitor appear in a list, where the client can click on one of them to see its details, navigating to the Client Details Screen.

Below the list of clients is the area reserved for controlling the daily exercises performed by each client. The monitor can see the number of daily exercises to do for each client and if you want to see the complete plan that that client is associated with, you can click on it. Then the monitor will be redirected to the Plan Details Screen.

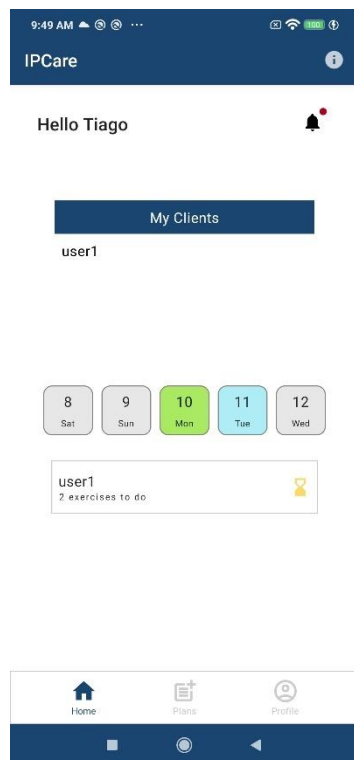


Figure 37 - Monitor's Home Screen

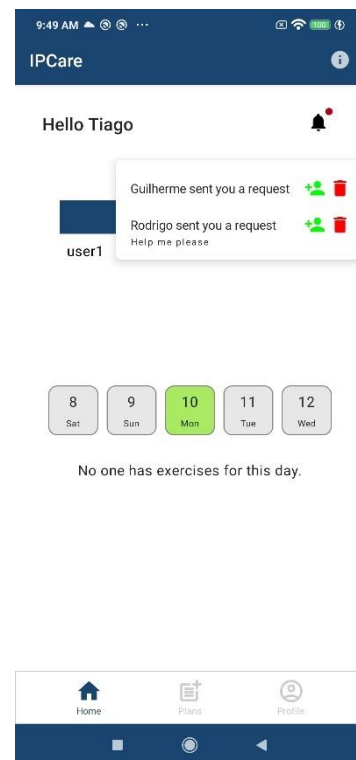
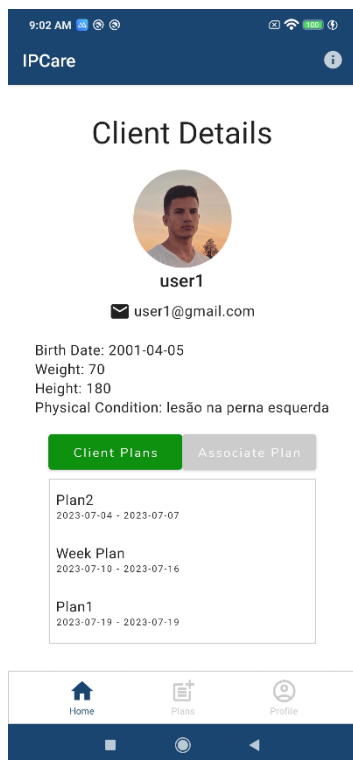


Figure 38 - Monitor's Home Screen with requests

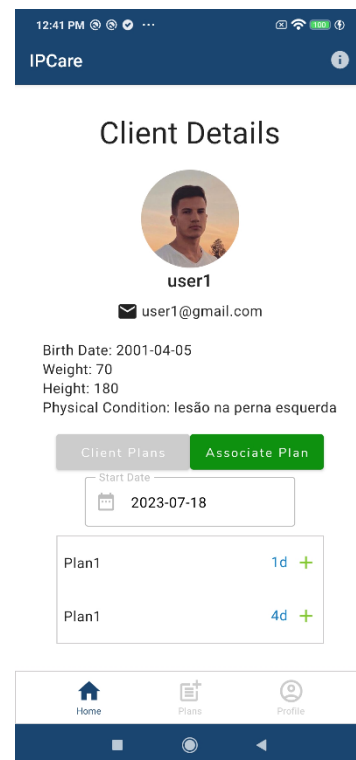
7.4.3. Client Details

On this screen, the monitor can see the details of a specific client, as well as the plans that this customer has assigned by the monitor, as shown in figure 39. If the monitor loads one of these plans, he will navigate to the Plan Details Screen and he will be able to see it in full.

It's also possible to associate a plan to that client from that monitor's plan library, choosing a specific start date, as shown in figure 40. It will not be possible to associate a plan to a client on a given date if he has an ongoing plan on that date.



*Figure 39 - Client Details
Screen with list of plans*



*Figure 40 - Client Details
Screen for associating a plan*

7.4.4. Create Plan

On this screen the monitor can create his plans. To create it, it is necessary to assign a name to the plan, add the necessary days, with the respective exercises, defining the number of sets and reps for each one, as shown in figure 41.

It is possible to add rest days, if the monitor doesn't add exercises to a given day.

The request to the server is only made at the end when the monitor presses the "+" button at the bottom.

In the end, this plan is added to the monitor's plan library, which can later be assigned to a specific client, as seen in figure 40.

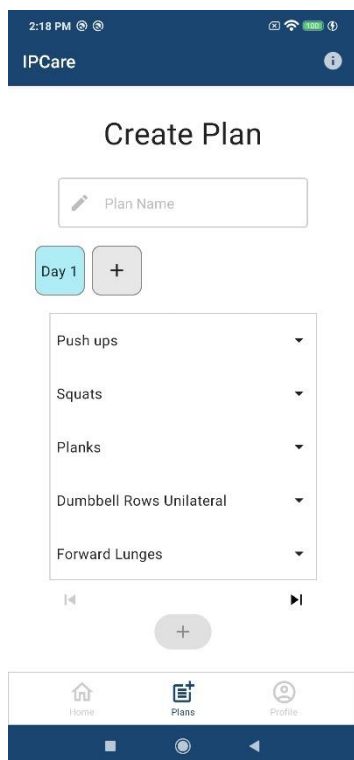


Figure 41 - Create Plan Screen

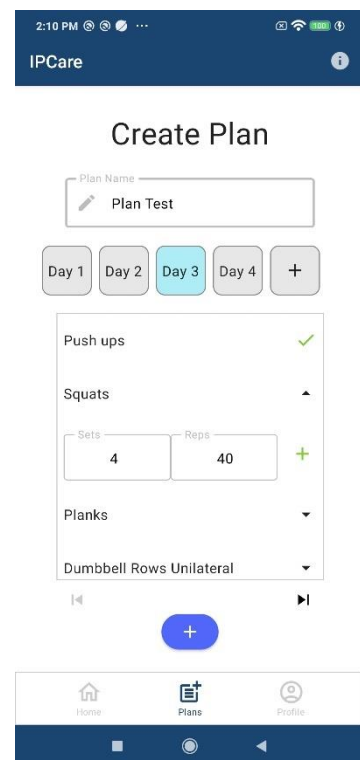


Figure 42 - Create Plan Screen with the daily exercises

7.4.5. Plan Details

Here it is possible for the monitor to check the status of the plan assigned to a certain client. He can also control the exercises that the client has already done or still must do.

An exercise is only considered really done if all the sets associated with it have been done.

If an exercise has already been performed by a client, the monitor can see its recording by clicking on the corresponding exercise, thus navigating to the Client Exercise Screen.

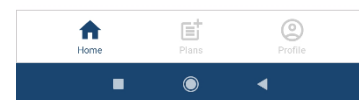
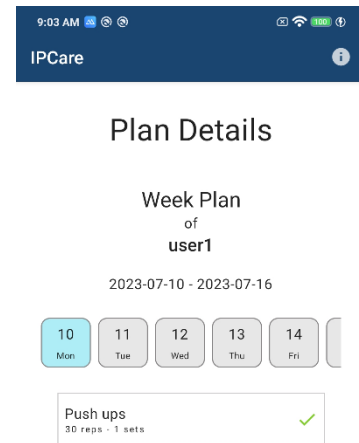


Figure 43 - Plan Details Screen

7.4.6. Client Exercise

When a client finishes an exercise, it's possible to see the videos of each set of that exercise through this screen.

The monitor can also leave a comment on each set, such as congratulating the customer for a job well done or making some criticism.

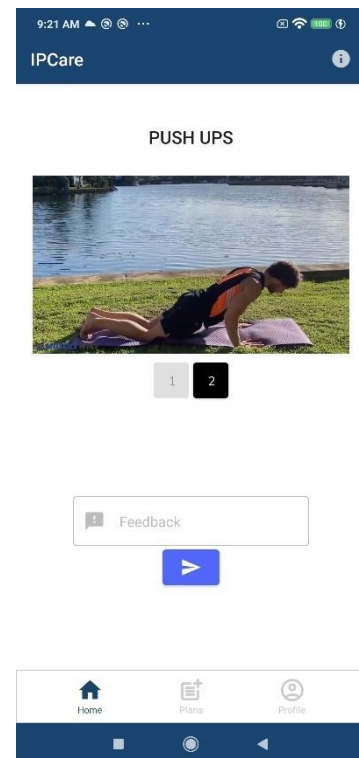


Figure 44 - Client Exercise Screen

7.4.7. Profile

The **Monitor profile screen** presents information about the monitor:

- Profile photo that can be altered at any time
- Name
- Email address
- Rating
- Reviews
- Submit credential option

The monitor will only have access to the other app functionalities once he has submitted his credential and it has been verified. As seen below in figures 45 and 46.

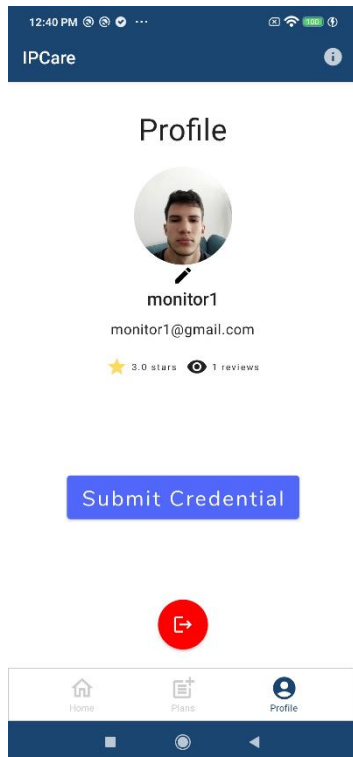


Figure 45 - Use case for Monitor Profile screen

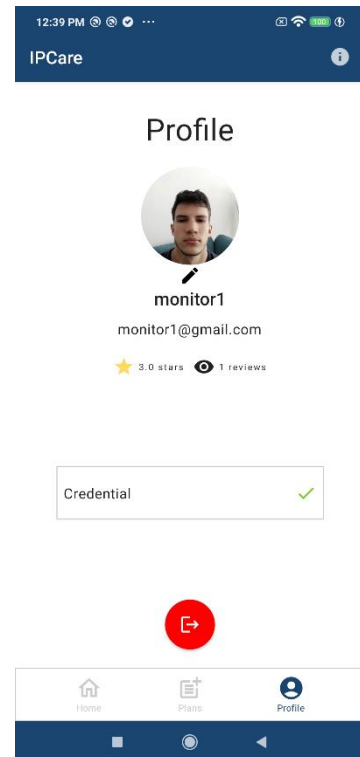


Figure 46 - Use case for Monitor Profile credential valid screen

7.5. About Screen

The **About screen** presents information about the authors of the app, and links to our **GitHub Accounts** and a link to our general **GitHub** application repository.

This screen is available in every screen that has the top bar, as seen below in figure 47.



Figure 47 - Use case for the About screen

Chapter 8

Conclusion

In conclusion, our Android application, the Intelligent Personalized Care project, aims to address the challenges posed by sedentary lifestyles and the lack of access to rehabilitation services for individuals with musculoskeletal conditions. By leveraging advanced pre trained machine learning models and other technologies, we strive to provide personalized and effective remote care, promoting physical exercise, and facilitating rehabilitation in a convenient and accessible manner.

The motivation behind our project stems from the alarming statistics provided by the World Health Organization, revealing that a significant portion of the global population suffers from painful musculoskeletal conditions. Furthermore, the shortage of healthcare professionals dedicated to rehabilitation exacerbates the need for alternative solutions. We recognize the importance of preventing injuries and physical issues caused by weakened muscles, while also acknowledging the crucial role of physical activity in combating chronic diseases and improving quality of life.

Through the development of our Android application, we offer a comprehensive solution that combines tele-exercise, tele-rehabilitation and vision techniques to enhance physical agility and recovery. Moreover, our application provides patient monitoring and follow-up features, enabling physiotherapists to track their patients' progress effectively.

This report outlines the architecture, functionalities, and technologies utilized in our project. It delves into the server and client components, describing their implementation, structure, authentication processes, and exception handling. As well as the way

In summary, our project strives to bridge the gap between the growing sedentary lifestyle and the need for rehabilitation services. By leveraging technology and combining it with personalized care, we aim to empower individuals with musculoskeletal conditions to take control of their health, enhance their physical well-being, and ultimately improve their overall quality of life.

While our Android application offers remote care and rehabilitation support, it is essential to recognize that it cannot fully replace traditional physical therapy. The app serves as a valuable tool to supplement in-person sessions by providing convenient and personalized guidance. However, direct interaction with healthcare professionals remains vital for comprehensive assessments, hands-on interventions, and tailored treatment plans. Physical therapists possess the expertise to perform manual techniques, assess progress, and make real-time adjustments based on individual needs. They also offer crucial emotional support and motivation. Our app aims to enhance accessibility and continuity of care, but it is not intended to replace the essential role of physical therapists in delivering comprehensive and personalized treatment.

8.1. Future Work

After all the mandatory requirements were made, we decided that some follow-up work must be done to evolve from proof of concept to product.

- Client side subscribe to **Server-Sent Events** for at real time notifications, polling no longer necessary
- Add more exercises to be analyzed by **ML Kit Pose Detection API**
- Add **text to speech** technology to the exercises for a better and easier user experience.
- The videos need to be **compressed** before sending to the server
- The application is in English, in the future **more languages should be added**
- **The addition of tests** is always an ongoing work
- **Chat service**, this way monitors and clients can chat in real time
- **Create a Screen** where the monitor could input an exercise with more detailed conditions, to personalize a specific exercise to a client with a specific problem or limitation.

References

- [1] World Health Organization – Musculoskeletal health. <https://www.who.int/news-room/fact-sheets/detail/musculoskeletal-conditions>
- [2] Kotlin programming language - <https://kotlinlang.org/>
- [3] PostgreSQL. <https://www.postgresql.org/>
- [4] Google Cloud Storage, <https://cloud.google.com/storage>
- [5] JDBI. <https://jdbi.org/>
- [6] Spring. <https://spring.io/>
- [7] URI. https://en.wikipedia.org/wiki/Uniform_Resource_Identifier
- [8] Jetpack Compose. <https://developer.android.com/jetpack/compose?hl=pt-br>
- [9] ML Kit – API Pose Detection. <https://developers.google.com/ml-kit/vision/pose-detection?hl=pt-br>, 2023. [Online; accessed 20-03-2023].
- [10] Ktlint, <https://pinterest.github.io/ktlint/0.49.1/>
- [11] Okhttp, <https://square.github.io/okhttp/>
- [12] CameraX, <https://developer.android.com/training/camerax>
- [13] Ngrok, <https://ngrok.com/>
- [14] Entity Association, https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model
- [15] BlazePose, <https://ai.googleblog.com/2020/08/on-device-real-time-body-pose-tracking.html>
- [16] CameraX VideoCapture API, <https://developer.android.com/training/camerax/video-capture?hl=pt-br>

