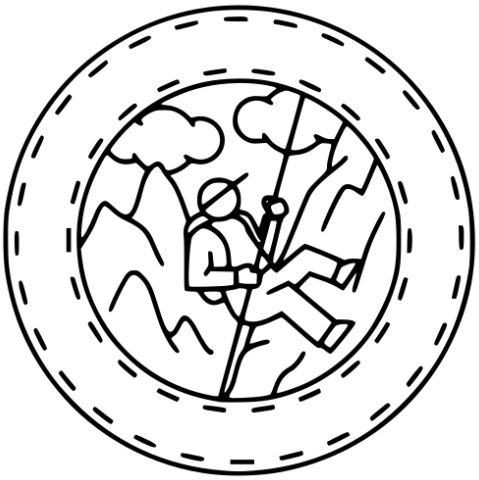
Uni Eibar-Ermua
Multiplatform Software Development

Final Project

Climby mobile application



CLIMBY

Author: Unai Zelaia-Zugadi

Tutor: Ane Erenaga Zabala

2023-2024

INDEX

1. Introduction

Climby is a comprehensive mobile application designed to cater specifically to the needs of climbers worldwide. Serving as a centralized platform, it facilitates the exchange of climbing-related information, including route updates, gear recommendations, and personal experiences. This technical documentation aims to provide users with an in-depth understanding of the application's functionalities, enabling them to make the most out of their climbing endeavors while utilizing Climby effectively.

2. Context

In the world of climbing, access to timely and accurate information is crucial to have a good experience when going to new routes. I know several climbers (who I'm building the app for) that have had bad experiences when going to routes because the knowledge is shared by word of mouth and is not always reliable. That is why the need for a reliable source of knowledge is needed in the world of climbing.

Information technology is a great solution for this problem, allowing people to share knowledge about the conditions, location and state(how the route is equipped is very important to know what to bring) of a route or area in real time. Through a mobile application, this information can be delivered in an easy way to consume even to people who are not technology savvy. With this application climbers can use the power of technology to connect with fellow climbers around the world allowing for a great space for knowledge sharing.

Climby serves as a collaborative hub for all climbers around the world to share their beautiful hobby. By leveraging information technology, Climby empowers climbers to make informed decisions, mitigate risks, and ultimately enhance their climbing experiences while contributing to the broader climbing community's knowledge pool.

3. Objectives

My objectives with this project revolve mainly around the backend, since that is the part I enjoy the most doing. I do not want to overlook, however, the Android application and I strive to create a decent app that will show a decent amount of information in an easy-to-navigate way.
First, for the backend I want to create a professional grade backend that will be fast, robust and secure. To do this I want to search for all the best practices and ways to secure an application possible. I also want to somehow Dockerize the application so it will be containerized and ready to use anywhere. For the frontend I strive to create a easy to use

and navigate Android application in a MVVM architecture(Model-View-ViewModel) with persistence in case there is no internet connectivity. I want the application to use the API provided in a way that all that it reads from it, it will be stored inside the applications database itself. Since this is an application for the outdoors and remote places, it makes sense to do it this way.

In the end my objectives revolve around the use of data, data-flow, data.consistency and persistence, more than making a good UI and UX or a feature packed application. I want to center around the robustness of the software first, and then create anything extra.

4. Resources

4.1 Human resources:

- **Analyst**: This person will interview and research the target audience of the project and gather their needs to then make a comprehensive analysis of what the features of the application should be.

- **Designer**: The designer will be in charge of the main architecture of the project, and will design the features themselves to make sure that they are feasible to implement and will decide how to implement them and which technologies to use. The designer will serve as a bridge between the Analyst and the developers of the application.

- **Graphical designer**: The importance of design in an application is very important to its success and a good graphical design makes an app very attractive to the audience. The graphic designer's role will be to make the overall aspect of the application attractive to the target audience.

- **Frontend developer**: The role of the frontend developer will be to program all the interface elements of the application. The role consists of programmatically creating the design given by the graphical designer and connecting said design to the backend of the application so the information will be properly visualized.

- **Backend developer**: The backend developer will deal with the database design and data manipulation part of the application. It will be the role of the backend developer to create a way for the data to be accessed by the application(mostly by API development) and to ensure a good data-flow.

- **Devops engineer**: The role of a devops engineer is to organize the task of all the other developers and organize the team to work correctly and synchronized. They will use project management tools and will organize the team to work in the best way possible to reach the deadlines.

- **Systems administrator**: The systems administrator will be responsible for setting up the servers and tools needed for the application to work and will deploy the application to the public. They will use the tools they deem necessary(albeit cloud or on-premise servers) to ensure uptime and a fast response to the application resources.

4.2 Physical resources:

- **Computers**: The main thing when developing an application is to have computers for the team to work on. The computers will be equipped with the necessary tools for each one of the roles(design software for the designer, IDE for the developers, messaging software for the whole team etc.).

- **Servers**: Cloud or on-premise, servers are necessary to run the application. They will host all the backend components of the application.

- **Mobile phones**: Since this is a mobile application, the mobiles are a crucial part of this app. Some might even say the most important. Phones for debugging will be used in development to make sure that the app works in real-life and will also be used by our users to use the app.

- **Distribution service**: Even if it isn't a "physical" thing, distribution is very important for the success of the app. We will use well-established distribution services such as the google-play store or Apple's app store to distribute our application. Apart from this, we will aim to distribute our own binaries so users who don't want to use said stores will be able to download a binary from ourselves.

5. Analysis of the application

5.1 Requirements analysis

5.1.1 Functional Requirements

1. User Authentication and Profile Management

- Users should be able to create accounts and log in securely.
- User profiles should allow climbers to provide relevant information such as climbing experience, preferred climbing styles and gear preferences.

2. Route Information

- Provide a searchable database of climbing routes worldwide.
- Include detailed information about each route, including location, difficulty level, route type (sport, trad, bouldering) and recent updates.
- Allow users to filter routes based on various criteria such as location, difficulty, and route type.

3. Route Conditions and Updates

- Enable climbers to report and view real-time updates on route conditions, including weather conditions, rock quality and any changes in the route's accessibility.
- Implement a system for climbers to rate and comment on route conditions, providing valuable insights for other climbers.

4. Gear Recommendations

- Offer recommendations for climbing gear based on the specific requirements of each route, including ropes, harnesses, shoes, and protection.
- Allow users to share their gear recommendations and experiences with specific routes.

5. Community Interaction

- Enable climbers to share their climbing experiences, trip reports and photos to inform others.

6. Safety Features

- Provide safety guidelines and tips for climbers, including information on proper equipment usage, emergency procedures, and first aid.
- Implement an emergency notification system to alert authorities or other climbers in case of accidents or emergencies.

5.1.2 Non-functional requirements

1. Performance

- The application should have fast response times and minimal loading times, even when accessing large datasets or multimedia content.

- It should be optimized for both online and offline use, allowing climbers to access essential information even in areas with poor internet connectivity.

2. Security

- Implement robust security measures to protect user data, including encryption, secure authentication protocols, and regular security audits.
- Ensure compliance with relevant data protection regulations, such as GDPR or LOPD, to safeguard user privacy.

3. User Experience

- Design an intuitive and user-friendly interface that caters to both experienced climbers and beginners.

4. Scalability

- Build the application on a scalable infrastructure that can accommodate a growing user base and increasing volumes of data.
- Implement monitoring and performance optimization tools to ensure smooth operation under heavy loads.

5.2 Technology-platforms

There are several technologies that would work for this project, it all depends on the experience of the developers on each of the technologies and how established and mature these technologies are.
For this project, it is clear that we will need 3 main components: A database to hold the information of our platform, a backend which will interact with our database and will do all the data manipulation and a frontend which will be a mobile application. In this next analysis I will list several options for each one of these technologies and reach a conclusion as to which one to use. Additionally, I will also make an analysis of deployment methods for this application:

5.2.1 Database technologies

PostgreSQL

Pros:

- Robust support for relational data modeling.
- ACID compliance ensures data integrity and consistency.
- Extensible architecture and scalability features.

Cons:

- Structured data modeling may be limiting for certain use cases.
- Scaling can be complex with very large datasets.

MongoDB

Pros:

- Flexible document-based data model, suitable for unstructured or semi-structured data.
- Scalable architecture with built-in sharding and replication.
- High performance for read-heavy workloads.

Cons:

- Lack of ACID transactions may pose challenges for maintaining data consistency.
- Schema-less design can lead to data redundancy and inconsistency if not managed properly.

Firebase Realtime Database

Pros:

- Real-time data synchronization between clients and server.
- Scalable NoSQL database with automatic scaling and offline support.
- Seamless integration with Firebase Authentication and Cloud Functions.

Cons:

- Limited querying capabilities compared to traditional databases.
- Data structure may need to be denormalized for complex queries, leading to redundancy.

5.2.2 Backend technologies

Go with GORM and Gin Framework

Pros:

- Lightweight concurrency model and efficient runtime.
- Simplified database interactions with GORM.
- Fast HTTP routing and middleware support with Gin.

Cons:

- Limited ecosystem compared to more established languages/frameworks.
- Steeper learning curve for developers unfamiliar with Go.

Java with Spring Framework

Pros:

- Extensive ecosystem with robust libraries and frameworks.
- Mature tooling and community support.
- Well-suited for enterprise-grade applications with complex requirements.

Cons:

- Heavier memory footprint and slower startup times compared to Go.
- Overhead of boilerplate code, especially in larger projects.

Python with Django

Pros:

- Rapid development with Django's batteries-included approach.
- Clean and readable code syntax, facilitating collaboration.
- Rich ecosystem of third-party packages and Django extensions.

Cons:

- Slower performance compared to Go or Java for CPU-intensive tasks.
- Django's monolithic architecture may lead to tight coupling between components.

5.2.3 Frontend technologies

Android App with Java

Pros:

- Native performance and seamless integration with Android platform.
- Extensive documentation and resources for Java development.
- Access to platform-specific APIs and UI components.

Cons:

- Platform-specific development may require additional effort for cross-platform compatibility.
- Java's verbosity may lead to longer development cycles compared to more concise languages.

React Native

Pros:

- Cross-platform framework, allowing code reuse across iOS and Android.
- Declarative UI components for rapid development and iteration.
- Large ecosystem of third-party libraries and community support.

Cons:

- Performance may not match native development, especially for complex animations or intensive computations.
- Reliance on JavaScript may introduce runtime errors and debugging challenges.

Flutter (Dart)

Pros:

- Cross-platform framework with native performance and UI rendering.
- Hot reload feature for fast iteration and debugging.
- Single codebase for iOS, Android, and web applications.

Cons:

- Relatively new framework with a smaller community compared to React Native.
- Limited access to platform-specific APIs and libraries compared to native development.

5.2.4 Deployment technologies

On-premises Deployment

Pros:

- **Control**: Complete control over infrastructure configuration.
- **Data Sovereignty**: Ensures compliance and data privacy.
- **Security**: Direct management of security measures.

Cons:

- **Costs**: High initial investment and ongoing maintenance.
- **Scalability**: Limited scalability and agility.
- **Maintenance Overhead**: Requires dedicated IT resources.

Cloud Deployment (IaaS)

Pros:

- **Scalability**: Elastic scalability for optimal performance.

- **Flexibility**: Customized configurations and resource management.
- **Cost Savings**: Pay-as-you-go pricing and economies of scale.

Cons:

- **Vendor Lock-in**: Dependency on specific cloud providers.
- **Security Concerns**: Data security and compliance challenges.
- **Performance Variability**: Influence of network and shared resources.

Containerized Deployment (Using Kubernetes)

Pros:

- **Portability**: Consistent deployment across environments.
- **Scalability and Orchestration**: Automated scaling and resilience.
- **DevOps Practices**: Enables CI/CD and agile development.

Cons:

- **Complexity**: Requires specialized skills and expertise.
- **Resource Overhead**: Higher infrastructure costs.
- **Networking Challenges**: Complexity in networking configurations.

5.2.5 Chosen technologies

The final chosen technologies will be:

- **PostgreSQL**: It's the most familiar database for our developers and its reputation is immaculate as the go-to choice for most companies when wanting a relational database. It is also ACID compliant, which is important when wanting to build a concurrent database.
  Furthermore, PostgreSQL has extensive documentation and a very active community, which makes it easy to overcome problems that may arise during development.

- **Go backend**: Even if go is not the most known language for our developers (who are more familiar with Spring) it has impressed our team with its simplicity and easy concurrency model.
  Coupled with GORM (Go Object Relational Mapping) and the Gin HTTP framework, it makes writing backend code a very enjoyable and easy task, while remaining simple and easy to write.
  Go is also very fast and scalable, with very low memory usage and a speed that python or node could never match, which in part makes it more efficient and that aligns with our team's focus on the outdoors and nature: Using efficient technologies to have less environmental impact.

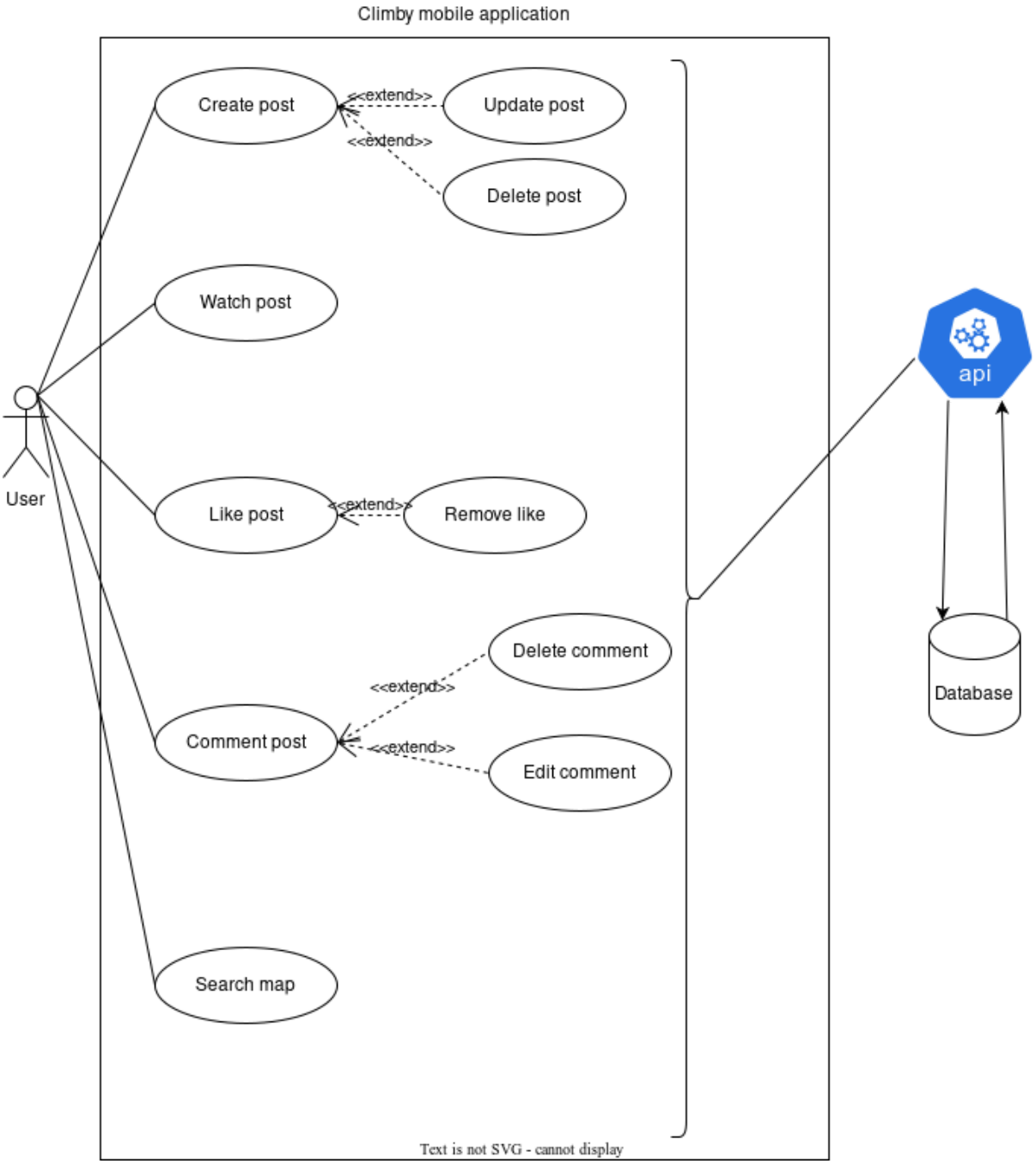
- **Native Android app**: A native android app is the go-to when wanting to write an android app without the need to port it to other mobile OS's or the web. It just works

very well. Since our product is not going to be released on any iOS phones nor the web, making this choice is very easy. Our developers are widely familiar with this technology and enjoy working on it, so it is a very easy choice. The question of using Java or Kotlin is still in the air at the time of writing this, since both languages can be used to build a native android app in a similar way.

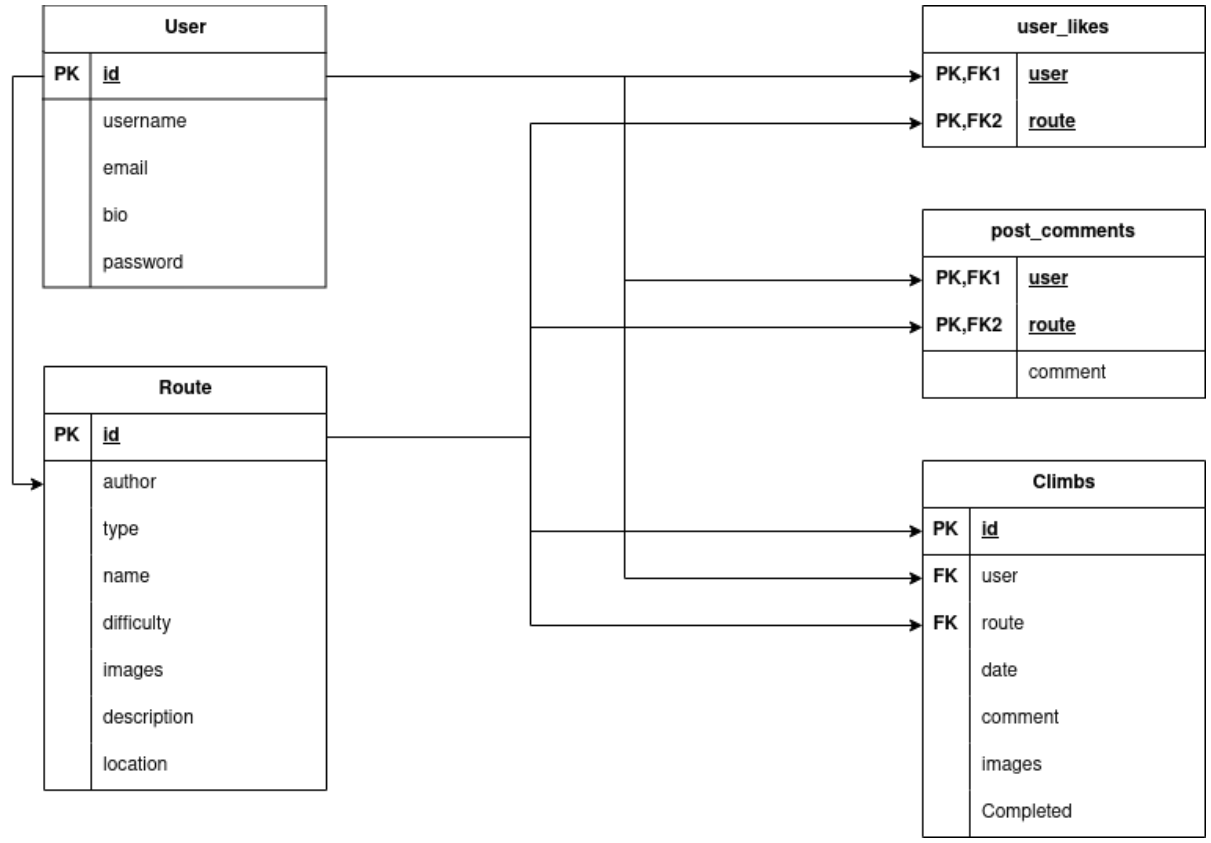
6. Application design

6.1 Use case diagram

This use-case diagram shows the interaction between our user-actor and the system. The user is the only actor in the system, since there is no implementation for an admin at this stage in the application and the only type of actor in the application itself is the user.


Climby mobile application
Create post
<<extend>>
Update post
<<extend>>
Delete post
Watch post
User
Like post
<<extend>>
Remove like
Delete comment
<<extend>>
Comment post
<<extend>>
Edit comment
Search map
api
Database
Text is not SVG - cannot display

6.2 Database design



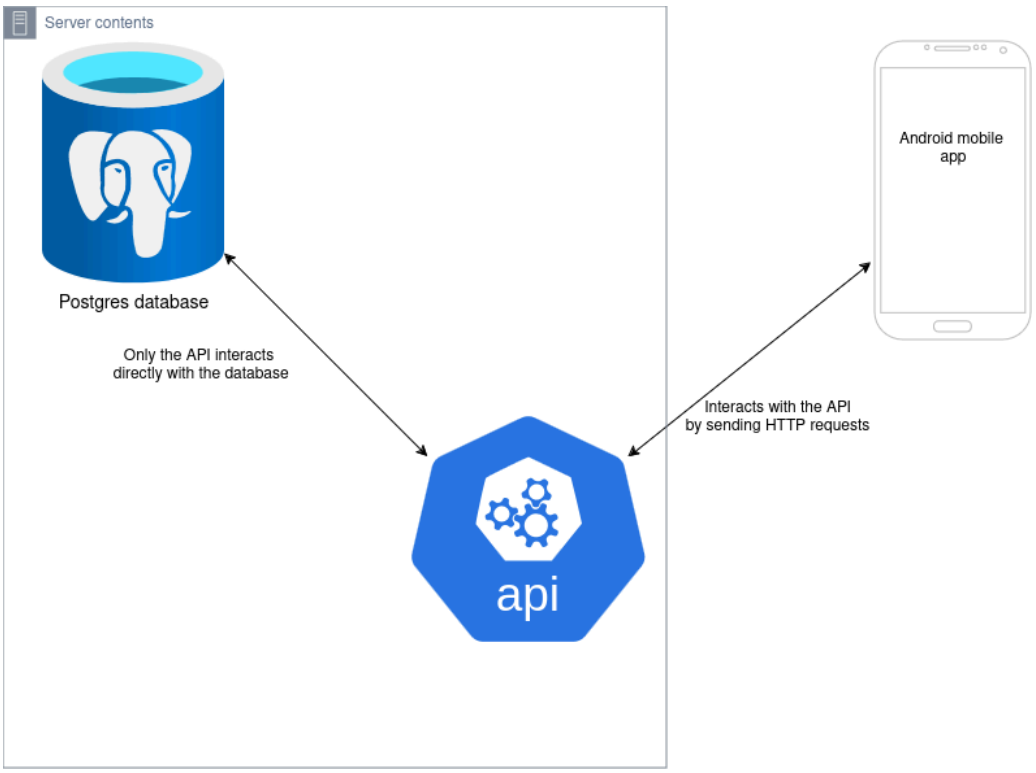
There are 3 main tables in this database: User, Route and Climbs. The User table has 1:n relations to Route and Climbs because one user can post many routes and one user can post many climbs. The Route table also has a 1:n relation with Climbs since 1 route can be climbed many times.

Then there are 2 tables that come from n:m relations. The user_likes table is created from the n:m relation between user and route, since 1 user can like many routes and one route can be liked by many users. The same happens with the route_comments table. One user can comment on many routes and one route can be commented on by many users, hence the n:m relation.

This database design is subject to change at any moment.

6.3 Application architecture overview

The next image shows how the application is going to be structured. Like in the previous explanations, it shows 3 main parts: The database, the backend(REST API) and the frontend(mobile app):



The application is a simple, three tier architecture for a mobile application. The three tiers are:

- **Mobile App:** The mobile application layer represents the user interface that users interact with. It's a native application written for a specific operating system, in this case the operating system is Android.

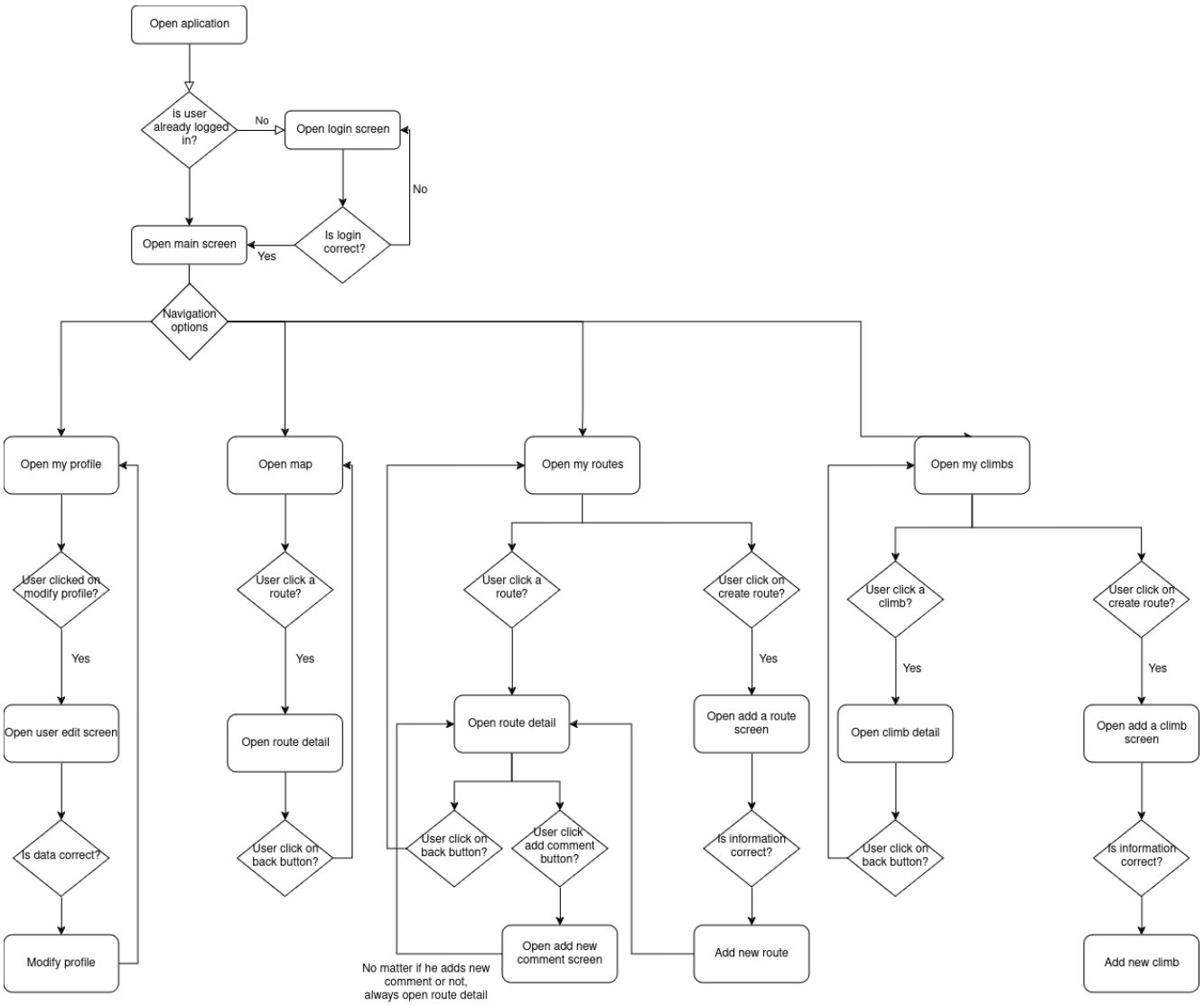
- **API:** The REST API layer acts as an intermediary between the mobile app and the database. It receives requests from the mobile app, processes them, and then sends them to the database tier. The API also performs tasks such as user authentication and authorization.

- **Database:** The database tier stores all of the application's data. In our case, it's a Postgresql database.

  The mobile app communicates with the API layer via HTTP requests. The API layer then communicates with the database layer to add, modify or retrieve data as necessary.

6.4 Activity Diagram

The activity diagram shows the main flow of the application. This activity diagram is subject to change at any time during application development.



7. Tasks

This section outlines the essential tasks required to develop the technical foundation of our mobile application. Here, we'll focus on the whole process of writing an application from its design, all the way to crafting the data storage, processing engine, and communication channels that will power the app's functionality, as well as the visual portion that the user will interact with.

Task	**Estimated duration**	**Start date**	**Finish date**	**Responsible**
App design				
Application’s needs analysis	2 hour	11/03	11/03	Analyst + client
General app design (subject to change, continuously changing)	1 hours	12/03	12/03	App designer
Selection of technologies	1 hours	12/03	12/03	App designer
Database model design	2 hour	12/03	12/03	App designer + Backend developer
Frontend screen design	2 hour	17/04	17/04	App designer + Graphics designer
Frontend UX design	1 hours	17/04	17/04	App designer + Graphics designer
Backend development				
Environment configuration	0.5 hours	18/03	18/03	Backend developer
Writing model programmatically in backend	3 hours	18/03	18/03	Backend developer
API endpoints development	8 hours	19/03	22/03	Backend developer
Security implementation	3 hour	25/03	25/03	Backend developer

Endpoint testing	1 hours	26/03	26/03	Backend developer
Frontend development				
Environment configuration	0.5 hours	05/04	05/04	Frontend developer
Implementing graphic design programmatically into app	5 hours	08/04	11/04	Frontend developer
Frontend functionality and wiring to the backend	6 hours	12/04	19/04	Frontend developer
UI testing	1 hours	22/04	22/04	Frontend developer
Deployment				
Server configuration	1 hour	01/05	01/05	Systems administrator
Application deployment	0.5 hours	01/05	01/05	Systems administrator
General				
Documentation	3 hours	06/05	09/05	Devops engineer + other developers
Project management	9 hours	11/03	end of project (continuous, projected on 20/05)	Devops engineer
Creating presentation	1.5 hours	13/05	14/05	Devops engineer + other developers
Presentation rehearsal	1 hour	15/05	17/05	Presenter of the project
	ESTIMATED TOTAL TIME	**START DATE**	**PROJECTED END DATE**	
	53 hours	**11/03/2024**	**20/05/2024**	

8. Tracking and evaluation

In this section I will talk about how I managed during the development of the application and several problems that have arised during it.

At the start of the project I laid down the basics of my application. After some thought, I came up with a database schema, which for me is the base of any project. This database changed during development, when I realized some of the fields and tables could be removed, merged or simplified in some way.

Then I focused on writing the backend of the application to the best of my abilities. I wanted to write a professional-grade backend, with token authentication and cookies, as well as middleware support for authentication. In this part of the project I'd say I was quite successful. My project is mainly based in the backend as that is the part of development I want to focus my career on. I have to say that the development of the backend was quite problem-free and straight-forward. The only real problem I had was that, when sending a request without a cookie to an authenticated endpoint, some of them would return data anyway and I didn't know why. There turned out to be some missing return statements in the error checking part, which was fast and easy to fix.
I knew I wanted something more from my backend, so I thought about what is the modern way to deploy server-side applications and the answer was quick to come: Docker.
I decided that dockerization was necessary to make a real professional-grade backend, so I got to it. At first I managed to write the dockerfile and docker-compose without much trouble. The docker-compose downloads and sets-up the database for the backend without needing anything else, so I was happy with that. After many years of installing VMs and having to configure databases, I thought that this was the way forward for writing backend systems. But there was a problem: My small application written in Go, when creating a docker image for it, weighed in at more than 1.3Gb. Not acceptable. So I started to investigate how to make it smaller. After some reading and trying, I managed to bring down the image of my application to a mere 40Mb. More than 1Gb shaved. I managed to do this using a lighter base image (alpine linux), multi-stage building and only copying the necessary files to the image. After all that, I got my easily deployable backend ready to accept calls from the application.

The second big part of the project was the android application. And I had much more trouble making it work than the backend. I started with laying down the base for the MVVM architecture with Room for persistence. Writing the DAO interfaces and all the queries, the repos and the viewmodels for each of the models. After doing all that, the application inexplicably broke. Room couldn't read the @Entity tags properly and it would not run. I started a new project after several hours of troubleshooting without a fix. This time, I realized some slight issues in my models, so I corrected them in this new version of the app. To my surprise, when executing for the first time after writing all the models, DAOs, repos and viewmodels, it ran first-try and without issue.
After having this base for my applications data, I started writing the DataAccess class to interact with the backend and the fragments to show the data at the same time. If I needed some data from the backend while writing a fragment I wrote a function for it and I worked that way for the remainder of the project. I had 2 major issues with the android application(

apart from how ugly it is): The parsing of LocalDateTIme with the backends Date.Date data type and sharing data between activities. The first issue is still not fully resolved. When creating a new instance of a climb, if that climb was not created from the device itself and is being downloaded from the backend, it will fail to parse and not display it. The second issue was an easy fix in the end, but it took a lot of time for me to find the solution(which again, was quite easy). I resolved that by creating a static class SharedData that holds the data that has to be shared across activities and all of the application in general. It turned out to work very well and the data is shared across all the app without issue.

In the end, I had few specific issues that took me some meaningful time to resolve.I had some minor issues that were easy to fix that I did not include in the text because it took me a maximum of 5 minutes to fix. Overall I'd say that the development went quite smoothly, with some hiccups here and there. Talking about the objectives I set for myself at the start of the project, is a mix. In the backend department, I think I was able to meet my objectives and create a really robust and well structured backend, but in the frontend department I fell quite short of my objectives. The application is quite simple for what I hoped to do and there are features that I did not implement due to lack of time, but the good part is that all the backend code is there to support new features if they are to be implemented in the future.

9. Installation and deployment of the software.

This is going to be a short section because, with the use of docker, the time and complexity to deploy the application is extremely short.

First, we will need to deploy the backend. To do that you will need a server or computer of some sort with Docker installed and Docker engine running. After making sure that your system has Docker, simply navigate to the root folder of the API folder and execute the command

```
docker compose up -d
```

The docker-compose will then automatically build the image for the backend, download the latest postgres image, run the postgres image configure with the data specified in the .env file, wait for postgres to initialize and run the API. All in one command.

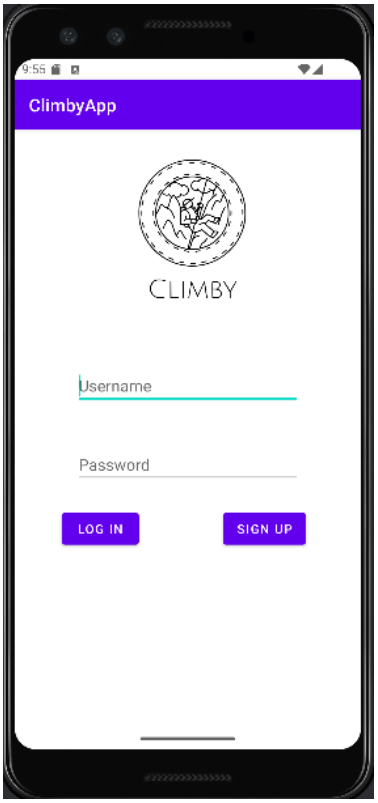
For the android application it is maybe even easier. Just go to the .apk file and download and install it on your mobile phone.

With those 2 simple steps, climby is ready in the server and in the client.

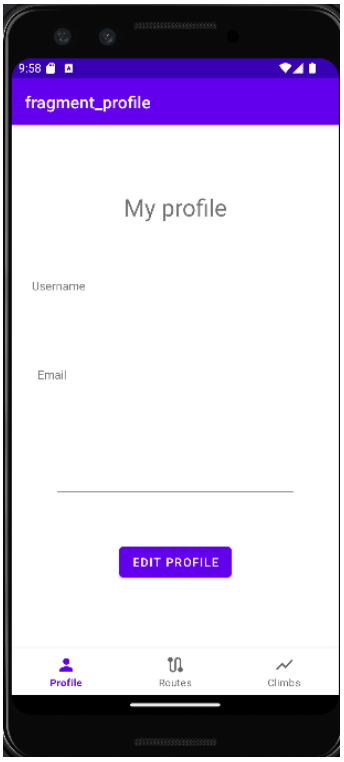
10. Guide on how to use the software

This guide is only for the client side android application.

After installing the application, you will be greeted with the login screen:

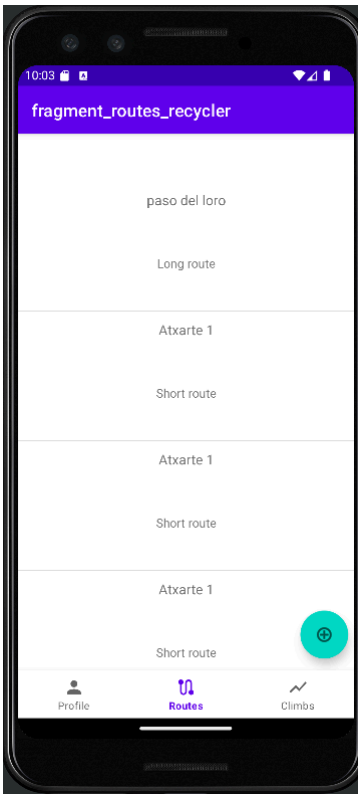


If you have an account you can go ahead and enter your credentials and log in, but if you don’t have one click on the sign up button, enter your data and click on sign up again. You will be automatically logged in and your user will be created. After logging in you will be greeted with your profile page:



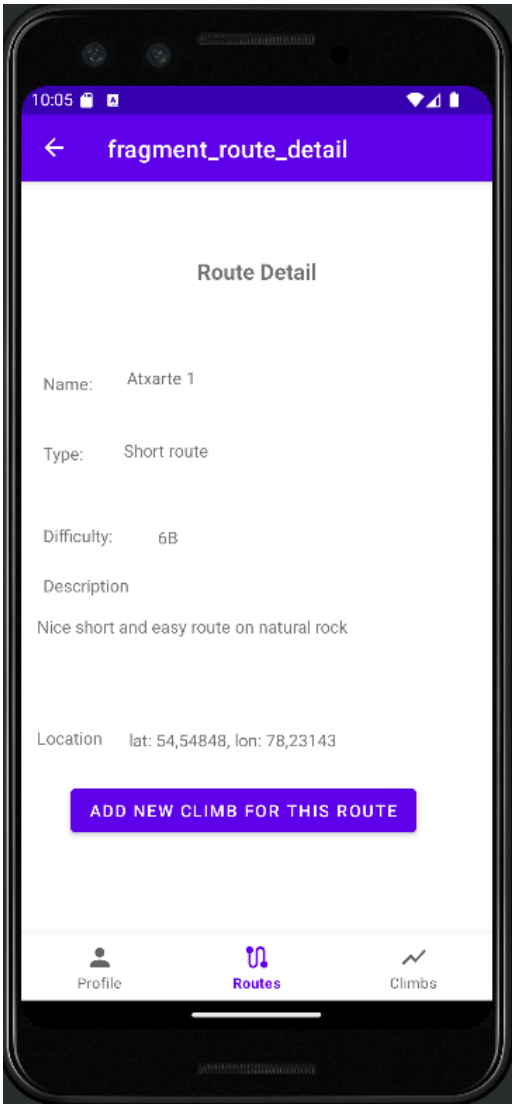
There is an issue with the application and it does not display the data, but this could be fixed in the future.
From this screen you will notice a bottom navigation bar with 2 other destinations: Routes and Climbs.

When clicking on routes you will be greeted with a screen with all routes that all users have created:

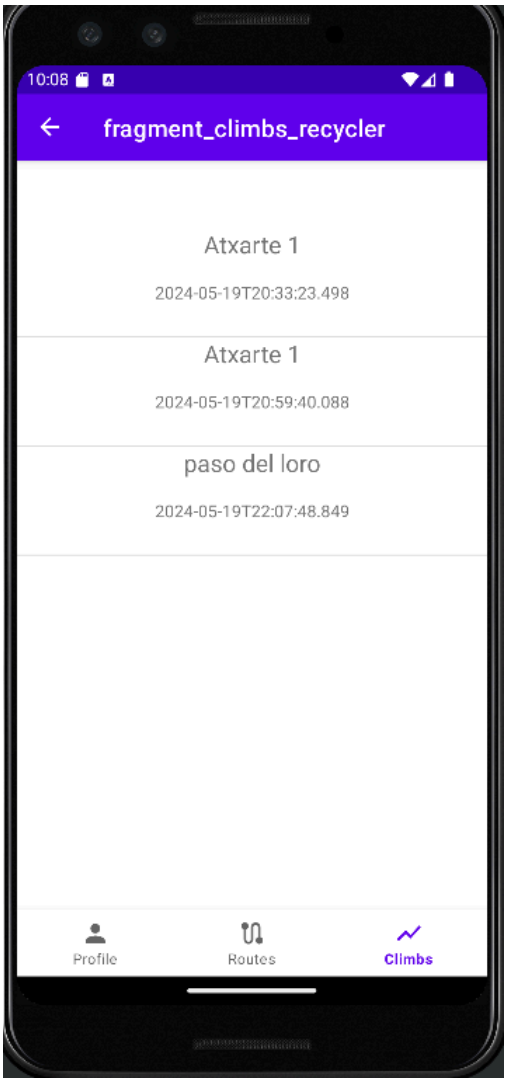


You can click on any of this routes and it will send you to the details of the route.
Alternatively, the + button on the bottom right will get you to the “add a new route” screen, where you will be able to add a new route yourself.

When on the route detail screen you will be able to add a climb to that route:



And if you click the button you will add a new climb. These climbs can be seen in the climbs navigation option of the bar. It’s different from the routes part in that you will only be able to see your own climbs. It look like this:



And when clicking on one of the climbs, you will be able to see the details of it.

11. Conclusions

In the end, my conclusions are mixed.
On one hand I have learned to write quality backend code while learning a programming language that I've a big interest for(Go) . I learned about database migrations and how to programmatically define tables and relations so I don't have to write SQL or do it manually. I learned how to use ORMs and how useful they can be when used correctly. There are several techniques I have learned in this project about backend code that I will bring to any other future projects I am in. Things I had no idea how they worked before like middlewares or environment and database initializers.
One of the big takeaways for me in this project is the use of Docker. Using Docker and composing a set of different systems has made it quite easy for me to continuously deploy my backend and be fast when wanting to make changes to it. I also learned about techniques to reduce weight and load of docker images, and since Docker is so popular, it is going to be very useful for me in the future.

On the other hand, I cannot say the same about the Android application. I used techniques and architectures that I knew before and I have barely anything to take away from the experience of writing Android applications, other than I do not like doing it. In my opinion, there is very little you can do while writing Android applications. Everything must be written as it needs to be and there is very little room to improvise. The Android system, while I am sure is very robust and tried and tested, makes you write with its own rules: A labyrinth of classes and functions specifically designed for Android that you will need to navigate correctly. If there is anything that I took away from writing this Application is that I do not want to do it ever again.