TÉCNICO LISBOA

Mobile and Ubiquitous Computing - 2024/25

Instituto Superior Técnico

ChargIST

Developed by:

Group 02
112265 – Daniela Camarinha
112269 – Laura Cunha
112270 – Rodrigo Correia

Contents

1. Introduction . . . 3

2. Implementation . . . 3
   2.1. Architecture . . . 3
   2.2. Mandatory Features . . . 3
      2.2.1. Components . . . 3
      2.2.2. Back-end . . . 4
      2.2.3. Resource Frugality . . . 5
      2.2.4. Context Awareness and Privacy . . . 5
      2.2.5. Caching . . . 6
   2.3. Additional Components . . . 6
      2.3.1. User Ratings . . . 6
      2.3.2. User Accounts . . . 6
      2.3.3. Localization . . . 7

3. Test & Setup . . . 7

4. Conclusion . . . 7

Appendix A: ChargIST Application Screens . . . 8

1. Introduction

ChargIST is a mobile application for Android, that explores several key aspects of mobile development, including location awareness, efficient use of limited resources, and social behavior. It provides mobile support for electric vehicles (EVs) and their owners, allowing users to find EV chargers in their surroundings, view information about the chargers and their capabilities, check their availability, and access additional information about nearby services such as air/water stations, restrooms, food options, and more.

2. Implementation

To fulfill the project's objectives, we have implemented all the required components of the assignment, including both the mandatory and three optional features, as detailed in the sections below.

Note: Appendix A presents screenshots of the various application screens, each accompanied by a brief description of its corresponding functionality.

2.1. Architecture

To implement our app, we decided to adopt the following package-based architecture:

- **data**:
  - **api**: Contains external APIs used by the app, such as *Nominatim* for query search and *LibreTranslate* for dynamic translation of user comments.
  - **database**: This is the *Room* database setup, with an abstract class acting as the app's database holder. It includes **entities**, which represent the *Room* tables, and ***DAOs***, which define the database access methods.
  - **model**: Contains the data classes that represent objects to be mapped into *Room* entities.
  - **repository**: Repositories that interact with the *DAOs* and allow operations to be performed on both *Supabase* and *Room*.
  - **supabase**: Contains ***DTOs*** (Data Transfer Objects) that map the tables from the *Supabase* database.
- **services**: These are called from the main entry points of the app and handle real-time synchronization between the local *Room* database and the remote *Supabase* database, according to the defined policies. Services include synchronization for favorites, maps, search, session, and view tracking.
- **ui**: Contains all user interface components, including screens and themes.
- **viewmodel**: Each screen has its own *ViewModel*, which acts as an intermediary between the screen and the repository layer.

2.2. Mandatory Features

The mandatory features include the core library management functionalities, ranging from basic features and UI, to backend, resource frugality, caching, and context awareness.

2.2.1. Components

For the mandatory components, we have the following:

- **Map:** For the map, we used the *OpenStreetMap* API. We chose this API because it doesn't require usage credits. We use its markers to represent both the charging stations and the user's current location, along with tooltips (or balloons) to provide context-aware information. The map itself is fully interactive, it's draggable and zoomable for easy navigation and includes a button to center the view on the user's current position.
- **Search Bar:** On the map, there is a search bar that allows users to search by address or station name. A dropdown appears with options matching the query, and clicking on an option navigates the map to the corresponding location. It uses the *Nominatim* library (from the same developers of *OpenStreetMap*) to fetch addresses based on the search query. The search bar also includes a cancel (X) button to clear the search.
- **Profile:** On the map screen, there is a button that takes the user to their profile. There, they can view their name, username, and profile picture. Users can also edit their name and profile picture. If logged in as a guest, they have the option to create an account or log out.
- **Favorites:** The user's list of favorite stations also appears in the profile. Favorite stations are marked with a red marker on the map to distinguish them. On the station details screen, there is a heart-shaped button that allows the user to add or remove the station from their favorites.
- **Station Details:** By tapping on a station marker or on a station card displayed in favorites or the map's swipe-up panel, the user can access a screen with the station's details. This screen shows the station's image, name, clickable location coordinates (that goes to the location on the map), payment methods, and available chargers with a brief summary including type, price, speed, and availability. It also displays nearby services, reviews and comments, average ratings, and a review histogram, along with an option to add a new review, this will be explained in detail in the Section 2.3.1. There is a favorite button to add or remove the station from favorites, as well as options to edit its displayed

components. Additionally, station cards include not only the name of the station, as well as a location button that, when clicked, centers the map on the station's location.

- **Add Station:** On the map screen, there is a button that allows users to add a new station by completing a four-step form. The steps include uploading a station photo (either taken with the camera or selected from the gallery), entering the station name, and selecting the location either by using the current location or picking a spot (on a draggable and resizable map where clicking places a marker). Users can select available payment methods and add chargers to the station. Chargers are added as bundles defined by type and speed, for which the user selects the type (*CCS2* or *Type 2*), speed (fast, medium, slow), price, and amount. These bundles can be edited or removed. Nearby services can also be added from a displayed list, though this step is optional. The final step is a confirmation screen to review all data, and users can navigate back and forth between the steps until they confirm.
- **Edit Station Details:** On the station details screen, there is a pencil icon that allows users to edit the station. This opens a form similar to the one used when adding a station, enabling users to update the station's details as well as its chargers.
- **Charger Details:** On the station details screen, there is a list of charger bundles available at that station. By selecting one, the user is taken to the charger details screen, where they can view and update the availability status, as well as see the type and speed of the charger. A charger can be marked as occupied or free, and there is also a button to report damages (vandalism, not charging, or physical damage). Additionally, the user can mark the charger as repaired, if it presents an issue.
- **Filter & Sort:** In the map's swipe-up panel, there is a filter button that allows users to filter stations by several criteria, such as stations with only available chargers, charger type, charger speed, charger price, available payment methods at the station, nearby services, maximum distance, and maximum travel time to the station. Users can also remove applied filters. The swipe-up panel also offers sorting options by ascending and descending price, more or fewer available chargers, closer or farther stations, faster or slower chargers, and shorter or longer travel time. According to the chosen sort type, the station cards that remain visible after filtering will display the value of the category used for sorting specific to each station.
- **Global:** For accessibility, all screens allow navigation back by popping the last screen from the stack using a back arrow icon. The app supports dark mode with logos and images adjusted to be visible in both light and dark themes, as well as adaptable text colors.

2.2.2. Back-end

In order to enable explicit data sharing and crowdsourcing across multiple devices, our backend uses ***Supabase*** to store and process shared data. We chose it over alternatives like *Firebase* or a traditional *REST* API because it better meets our needs, primarily due to its **real-time** capabilities, which allow us to handle events and provide seamless updates within the application. Additionally, since *Supabase* is built on *PostgreSQL*, we can use **standard SQL**, implement custom logic, and benefit from an open-source solution. This flexibility was a key factor in our decision.

To support our application, we structured the following **tables** in *Supabase*:

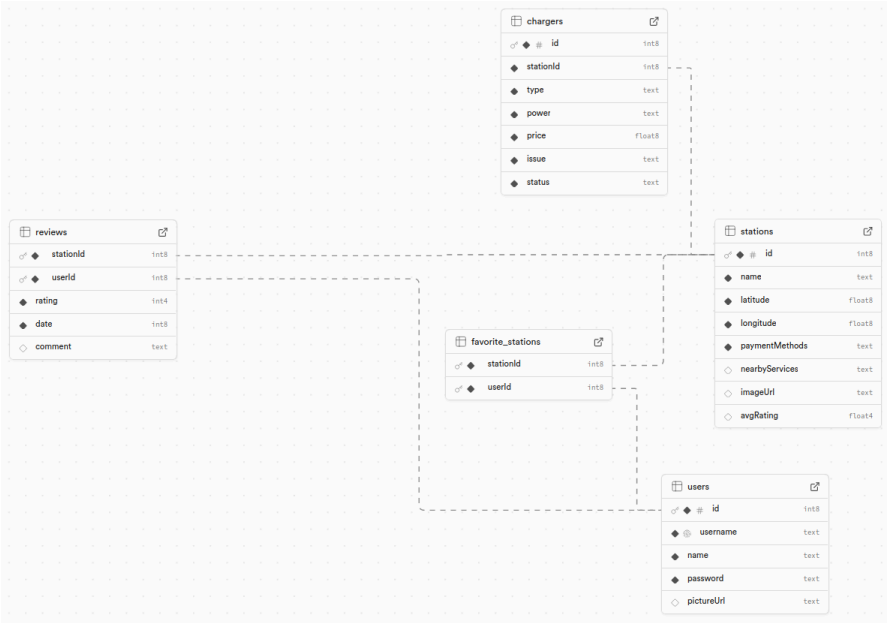


Figure 1: ***Supabase* tables**: users, stations, chargers, favorite stations, and reviews

We also emphasize security, **passwords** are **hashed** using **Argon2** before being stored in the database to ensure they are never saved in plain text.

Images, whether uploaded from the gallery or captured using the camera, are stored in a bucket of ***Supabase* Storage**. This service enables us to upload image bytes directly and store them as bitmaps in designated buckets. Each image is named using the corresponding user or station ID, and a public HTTP link to the stored image is then saved in the appropriate column within the database tables.

Since *Supabase* supports SQL, we created **SQL/PLSQL scripts** to initialize and populate the database. We also implemented triggers to automatically update the average rating of a station whenever a new review is added.

2.2.3. Resource Frugality

To save network and device resources, we made the following decisions for resource frugality:

- **Realtime Updates**: Using *Supabase Realtime*, a user's list of favorites is **automatically updated across all devices** where they are logged in. Additionally, if a user is viewing the details of a station and its associated charger bundles, and another user makes changes to that station, those updates will also be reflected in real time. We assume a **last-writer-remains** consistency model: if two users edit the same station simultaneously and one finishes first, the second user's changes will overwrite the first's once they are submitted. This approach works seamlessly over both **Wi-Fi** and **mobile data**, ensuring synchronization across devices while using the network efficiently, and avoiding constant polling, which drains the battery of the mobile device.
- **Pre-loading**: To use the network efficiently, the application performs smart pre-loading of key content when the user is connected to **Wi-Fi**, as described in the Section 2.2.5 strategy. Since Wi-Fi usage incurs no cost to the user, the app pre-loads not only the visible details of nearby stations but also those of all stations previously stored in the local *Room* database. We limit this to only the *Room* entries to avoid consuming too much local storage, balancing the trade-off between information availability and device capacity. When the user is on **mobile data, no pre-loading** is performed to conserve bandwidth, getting only some columns of the station's table in the *Supabase*.
- **Images Loading**: Particularly large content such as images can be further optimized to avoid consuming costly metered data. For **profile pictures**, which are not considered critical, updates are only reflected after the user logs back into the app, avoiding unnecessary requests to the remote database for non-essential changes. For **station images**, the image URLs are updated in real time via *Supabase* to ensure station data stays current. However, the images themselves are cached by the Android system, meaning that the app will only reload the image when Android determines the cache is outdated, similar to browser behavior, since the image URLs are HTTP links, a full refresh may take a moment to reflect the latest content. For **profile pictures in reviews**, the behavior is the same as with station images, when retrieving reviews, the app uses the image available in the database or relies on the Android cache.
- **Visible Information**: To avoid overloading both the network and the user's device storage, it's essential to minimize unnecessary resource usage, so the app only downloads data related to UI elements as they become visible to the user. For instance, **only stations within a 30 km radius** of the user **and those currently visible on the map** screen are downloaded.
- **Repeated Information Avoidance**: To prevent unnecessary data transfers, *Supabase*'s **real-time** feature is helpful. Although the first request sends the entire table, subsequent requests only send the rows that have been modified, that is useful for the stations and reviews tables, the data is cached after the initial load, and only updates are received afterwards.
- **Incremental Loading**: For the list of stations and reviews, we **fetch the data incrementally** from the database, similar to paginated loading. As the user **scrolls**, more batches of them are fetched and displayed.
  In the **reviews** section, when a user opens the station details, only **three reviews** are loaded initially, either from the local *Room* database or the server if needed, being only fetched from the database only when the user taps '***View More***' button and **scrolls** further, with reviews being loaded in **batches of 3** per scroll.
  Likewise, when swiping up the **station cards** from the map, station data is progressively retrieved from the database as the user **scrolls and taps the "*Load more*" button**. This triggers a **backend function** that filters the stations according to the user's selected **filters** and returns them in **batches of 5**, so the data is loaded on demand, reducing unnecessary data transmission and keeping the app lightweight and responsive.

2.2.4. Context Awareness and Privacy

In order to address the context awareness and privacy requirements of our application, station markers within a **5 km radius** on the map will **automatically display a label** showing the station's rating. If the station is outside this range, the label will not appear. This approach allows users to quickly obtain relevant information without cluttering the map.

2.2.5. Caching

As many users operate under unstable or metered data connections, to ensure a smooth and efficient user experience, the *ChargIST* application is designed to minimize unnecessary network usage and maintain functionality during short-term outages.

To support this, we implemented a local persistence layer using Android ***Room***, enabling the application to store data on the device, avoiding repeated downloads, speeds up access, and allows continued use even when offline.

Room acts as a structured **local database**, mirroring key components of the remote *Supabase* backend. It includes *@Entity* classes to represent tables and *@Dao* interfaces to perform SQL queries. These are used by the repository layer to interact seamlessly with both local and remote sources.

The **local database (*Room*)** includes the following tables:

- **chargers**: id, stationId, type, power, price, status, issue
- **favorite_stations**: stationId, userId
- **reviews**: stationId, userId, rating, comment, date
- **stations**: id, name, latitude, longitude, paymentMethods, imageUrl, avgRating, nearbyServices

These tables correspond directly to those in *Supabase*, ensuring data consistency. Synchronization occurs only when necessary, as explained in the Section 2.2.3, based on predefined policies, reducing data consumption.

Additionally, when the device is connected to **Wi-Fi**, which incurs no cost to the user, the application performs smart **pre-loading** of key content, that includes not only the visible details of nearby stations, but also those of all stations previously stored in the *Room* database. As a result, users can later access this information without relying on mobile data, even when offline. This approach improves performance, reduces network dependency, and enhances the overall user experience, especially in low-connectivity scenarios.

2.3. Additional Components

For additional features, in order to complement the mandatory ones, we chose three: **user ratings**, **user accounts**, and **localization**, which we explain below.

2.3.1. User Ratings

In the station details, there is a **ratings and reviews** section. It shows the **average rating** and the **total number of submitted reviews**, as well as a **histogram** indicating the number of reviews by rating category. There is a "*Write Review*" button where users with an account can submit a **comment (optional)** and rate the station from **1 to 5 stars**.

If a comment is added, a review card (showing stars, comment, date, and a translate option) will be displayed in the section. If a user submits a new review after already having submitted one, the previous review will be overwritten.

The station details screen shows the three most recent reviews and includes a "*View More Reviews*" button that leads to a scrollable screen where all reviews for the station are available.

The review's logic is included in both the *Station Details* screen and the *Reviews screen*, which is accessed by pressing the "*View More*" button. The logic for handling reviews is implemented in both corresponding *ViewModels*, the *ViewSync* service, as well as in the database layer through the relevant repository, *DAO*, and *DTO* classes, and the tables for the *Supabase* and *Room*.

2.3.2. User Accounts

The user has the possibility to **create an account** by entering their name, username, password, confirming the password, and uploading a profile picture. If the user already has an account, they can simply **log in** with their username and password, and each user/account has a unique ID (*Long*) that allows tracking.

If a user logs in as a **guest**, that means without an account, they have access to all app functionalities except writing reviews, which requires logging in. Favorites are stored locally using *Room*, and only when a **guest upgrades to a registered account** are these **favorites saved** to the database to ensure they are not lost.

Moreover, the authentication system ensures that once a user logs into a device, their **session is persisted locally**. As a result, users do not need to log in again when reopening the application on the same device, improving convenience and streamlining the user experience.

In addition, since the application integrates with ***Supabase***, we take advantage of its **RealTime** service to receive live updates on data changes, enabling seamless **synchronization across devices logged into the same user account**.

Users can create accounts and log in from multiple devices, and any **changes**, such as adding a station to favorites, are instantly **reflected across all devices associated with the same account**.

The user's logic is included in *Profile*, *Register* and *Login* screens and its correspondent *ViewModels*, as well as in the database layer through the relevant repository, *DAO*, and *DTO* classes, and the tables for the *Supabase* and *Room*.

2.3.3. Localization

Since users can be multilingual, the app supports translation into **English**, **Portuguese**, and **Spanish**, and stores these strings in a way that makes adding new translations easy without requiring refactoring the application.

Static app strings are stored in *XML* files within the **res/values folders**, allowing Android to automatically load the appropriate translation based on the device's language setting.

For **dynamic** translations, such as review comments, each review card includes a **translate button** that translates the original comment into the device's language, which can be any supported language. To achieve this, we use the *LibreTranslate* library, which must be hosted locally using *Docker*, enabling dynamic translations by sending requests to the API and displaying the received translated responses. The logic for this dynamic translation is implemented in the *ReviewViewModel*.

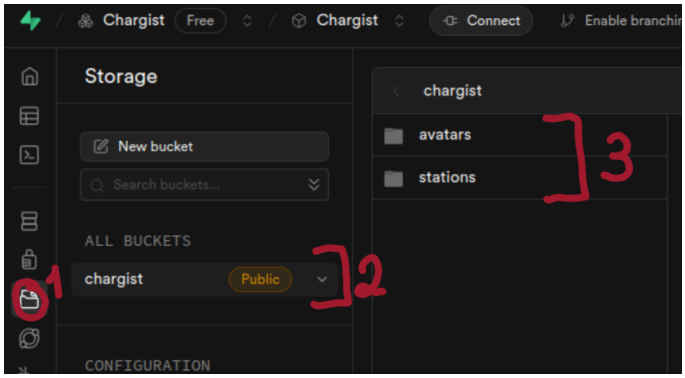
3. Test & Setup

To test the application, you need to:

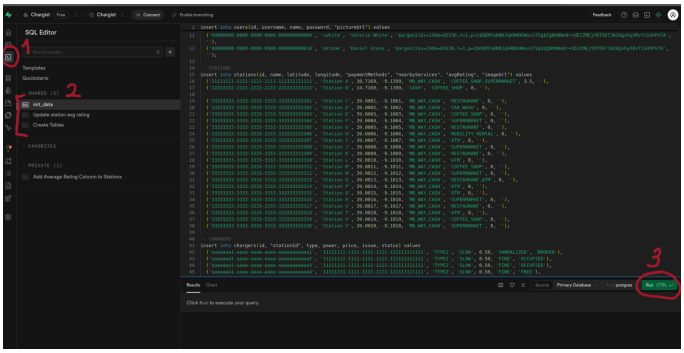
- Run the ***Docker*** container for dynamic translation from the root of the project using the command: ***docker compose up***.
- **Open the project in an up-to-date version of Android Studio**, by importing the *app* directory of the project as a project in Android Studio.
- You are now ready to use all the features of the application normally.

The **database is already set up**. However, if you want to **configure your own API key**, follow these steps:

- Create an account on ***Supabase***, set up a project, and update the API key in the ***SupaClient.kt*** file.
- Create a **bucket** in ***Supabase*** and add the folders "*avatars*" and "*stations*" to store user and station images, respectively, as shown in the image Figure 2.
- Copy the **SQL scripts** from the "*scripts_supabase*" folder (located in the root of the project) into the **Supabase SQL editor** and execute them, as shown in the image Figure 2.



Steps to set the buckets into _Supabase_.



Steps to set the sql files into _Supabase_.

Figure 2: Steps to setup buckets and SQL scripts into *Supabase*.

4. Conclusion

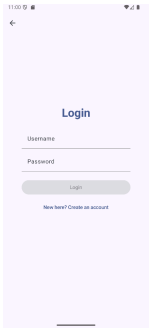
Throughout the development of *ChargIST*, we successfully implemented all mandatory features and additional functionalities, effectively addressing the challenges of mobile and ubiquitous computing. Our solution enables users to search, manage, and interact with electric vehicle stations in a way that is both efficient and user-friendly.

The architecture we designed promotes maintainability and scalability, while our choice of technologies, such as *Supabase* for real-time backend support and *Room* for local caching, ensures consistent performance in both connected and offline scenarios. We also addressed key concerns such as privacy, context awareness and resource frugality to enhance the user experience.

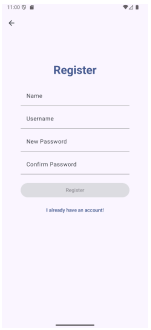
Ultimately, *ChargIST* demonstrates how mobile technologies can be leveraged to create smart, context-aware solutions that support sustainable mobility and improve user interaction with their surroundings.

Appendix A: ChargIST Application Screens

Open Screen

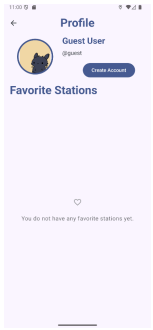


User Login Screen

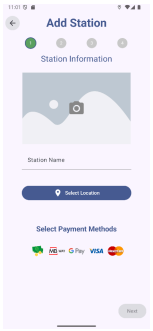


User Register Screen

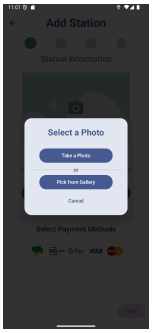
Map Screen



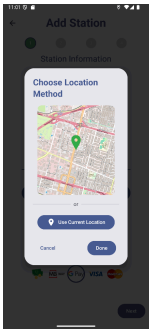
Guest Profile Screen



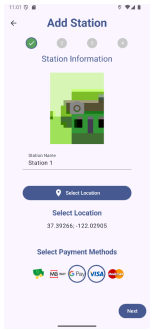
Add Station Screen - Step 1



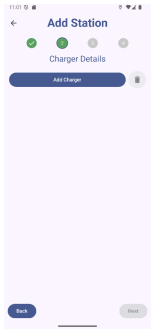
Add Station Screen - Step 1 - Pick Image



Add Station Screen - Step 1 - Pick Location



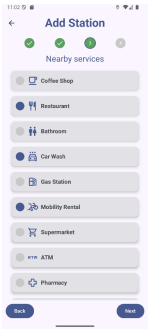
Add Station Screen - Step 1 - Filled



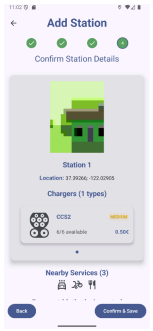
Add Station Screen - Step 2

Add Station Screen - Step 2 - Pick Charger

Add Station Screen - Step 2 - Filled

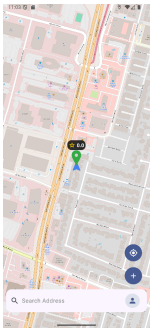


Add Station Screen - Step 3 - Filled

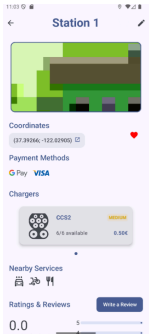


Add Station Screen - Step 4

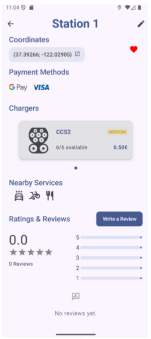
Add Station Screen - Step 4 (cont.)



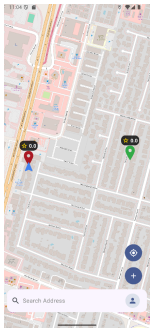
Map with station



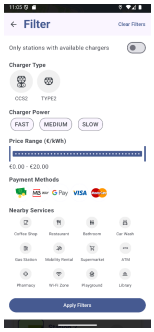
Station Details Screen



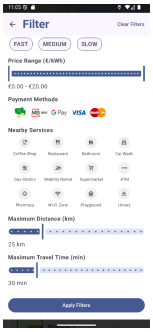
Station Details Screen (cont.)



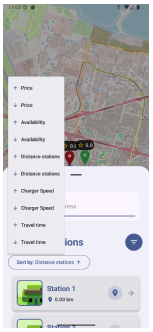
Favorite Stations on Map Screen



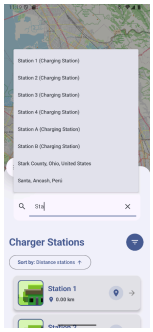
Stations Filter



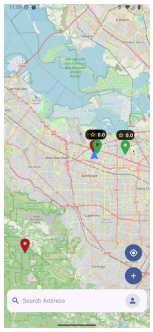
Stations Filter (cont.)



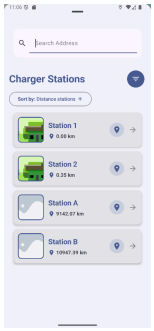
Stations Sort



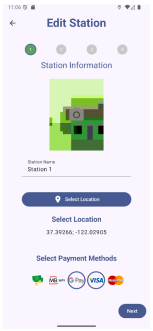
Stations Search



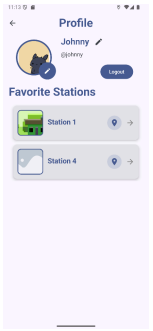
Context Awareness with 5kms radius



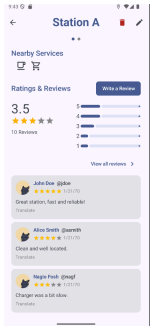
Swipe up menu with station's list



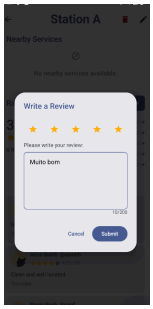
Edit Station Screen



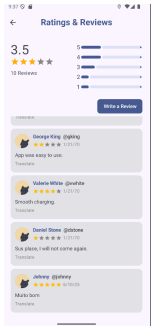
User Profile - Upgraded from guest



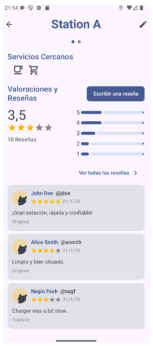
Reviews Section



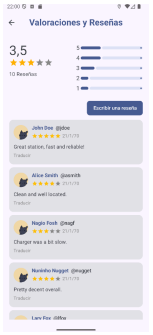
Write Review



Added Review



Translated reviews and Station Screen to Spanish



View more reviews

Figure 4: Captures of the different screens and funcionalities of ChargIST.