



# X-Platform Squad

# Table of Contents

<b>Introduction</b>	<b>3</b>
<b>Architecture</b>	<b>4</b>
Naming convention	4
Services	4
LocalData	4
Models	4
<b>Testing</b>	<b>5</b>
Automated Tests	5
Identifying Widgets	5
Widget Test Limitations	5
Unit Tests	5
<b>Klarna Payment Integration</b>	<b>6</b>
<b>Workflow</b>	<b>7</b>
Connect to charger	7
<b>Live Metrics</b>	<b>7</b>
Backend Implementation	7
Receiving Live Metrics Data	8
<b>FlexiCharge Backend API</b>	<b>10</b>
Postman Collections	10
<b>Notes &amp; Requirements</b>	<b>11</b>
Run the app	11
iOS	11
Android	11
Additional tips	11
<b>Google Maps API KEY</b>	<b>11</b>
<b>Flutter Version</b>	<b>12</b>
Testing Registration Functionality	12
Flutter Secure Storage	12
Bug Tracking	12
App Flow Prototype	12
<b>QR Code Scanning</b>	<b>13</b>
<b>Code Comments</b>	<b>14</b>

# Introduction

This document contains information about the FlexiCharge Cross-Platform application. The application was started as part of the FlexiCharge project at Jönköping University. The project was continued by students in 2022 and is planned to be continued further during upcoming years.

# Architecture

The codebase utilizes an architectural design pattern called *Stacked Architecture*. Documentation about the Stacked architecture can be found here: [Flutter and Provider Architecture using Stacked - FilledStacks](#).

## Naming convention

- Files: snake\_case
- Classes: PascalCase
- General variables: camelCase
- Private member variables: \_camelCase
- Constants: SCREAMING\_SNAKE\_CASE

## Services

Used mostly when accessing the API and making requests, see more information here: [Services in Code and how to use them in Flutter - FilledStacks](#)

## LocalData

LocalData is the name of a class that enables us to save variables into an object that we can access from anywhere. A unified place to find common variables used throughout the entire application.

## Models

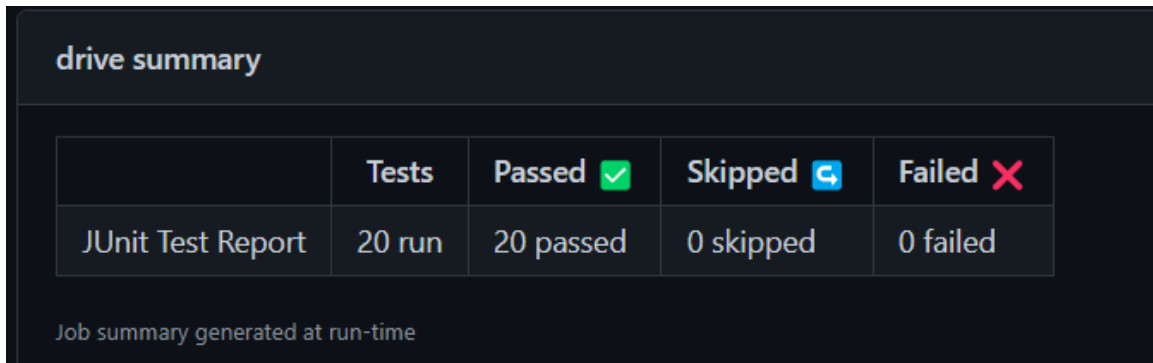
Some of the models define data structures which can be mapped to- and from json data. Other models are used for business logic. In order to follow the Stacked Architecture, it is possible that some Models should instead be Services.

# Testing

Test cases are located in the top level folder *test* and are divided into separate files for Widget tests and Unit tests.

## Automated Tests

All test cases are executed at each push and merge to the *main* branch in the GitHub repository. Integration tests are a goal to implement in the future. The tests are run by a GitHub actions workflow that starts a Flutter environment on a Linux runner which is able to perform the tests. The test results are displayed with the help of JUnit which can be seen in the image below.



The image shows a screenshot of a JUnit Test Report. At the top, it says 'drive summary'. Below that is a table with the following data:

	Tests	Passed ✓	Skipped ⚠	Failed ✗
JUnit Test Report	20 run	20 passed	0 skipped	0 failed

Below the table, it says 'Job summary generated at run-time'.

## Identifying Widgets

Thanks to Keys the test cases are able to find specific Widgets that should exist (or should not exist), and are able to click on button widgets etc. Each Widget that needs to be used in a test case is provided a value for the property *key*, the same value is then used in the test case to fetch the correct widget. Key Constants are contained in a class named *WidgetKeys* placed in the file `/lib/models/widget_keys.dart`.

## Widget Test Limitations

It can be difficult to test entire app flows, such as a login process which makes use of HTTP requests in the background. The HTTP client usually used in the application is replaced by a Mock client when widget tests run since they are not supposed to test anything related to external applications and integrations. A solution to this is to use a Mock library to simulate a fake HTTP response.

## Unit Tests

Since a mobile application of this sort contains little logical functions, only few parts of the codebase logic are testable. In order to easily write unit test cases, logical operations should be separated into small single responsibility functions with few dependencies.

## Klarna Payment Integration

There is no Klarna SDK for Flutter, however there exists an SDK for both Android and iOS. Therefore the solution for the Klarna payment feature has been implemented natively in both iOS and Android. The iOS integration handles most API calls itself, while the Android integration does not.

Due to endpoints missing or not being documented or not working, the Klarna payment feature has not been possible to implement completely. Some of these API calls are therefore skipped, so no actual charging is actually started despite the Klarna payment view and the charger status bar.

When making a reservation for a charger, use the following personal details in Klarna displayed in the image below:

Personal Number	410321-9202 19770111-6050 when testing direct debit in Playground
First Name	Testperson-se
Last Name	Approved
Street	Stårgatan 1
Zip Code	12345
City	Ankeborg
Phone Number	0765260000
Email	<a href="mailto:youremail@email.com">youremail@email.com</a>

The Credit Card Details displayed in the image below should be used for Klarna payment:

Credit card number:	4111 1111 1111 1111
CVV:	123
Exp:	12/25 (Or any other valid date in the future)

## Workflow

Documentation about how certain actions are performed within the app, steps needed and in what order etc.

### Connect to charger

When a valid charger ID has been entered we want to reserve the charger during the payment process. To enter a charger, we need the charger ID. We can get this from three different ways.

Scan QR code on the charger, which automatically enters the charger ID into the PIN input field. Choose a charger point (green marker on the map view) and select a charger from the list. This will also automatically enter the pin into the PIN input field

Enter the Charger ID manually. Then, we create a transaction session and receive a transaction object back from the FlexiCharge backend API. This transaction object contains a transactionID that we need when getting an authentication token from Klarna. The authentication token from Klarna can then be used to update our transaction object that verifies that our transaction was successful.

In *snappingsheet\_viewmodel.dart*, the *connect* function solves this

## Live Metrics

Live Metrics is used for displaying the current charging level in real-time. This means that when a car charging session starts, the application is supposed to connect to a websocket and continuously receive data about the current charging level percentage along with some related information such as price/kWh and time to fully charge.

## Backend Implementation

Live metrics are live on the server along with a temporary Klarna fix. This means:

- That you can start and stop transactions without having to go through Klarna.
- Since the server doesn't involve a charger; one must be simulated by using Postman or Insomnia.
- To test your app you don't have to connect with a user socket through Postman, instead do it from your app.
- Transactions should also be started from your app

## Receiving Live Metrics Data

Follow the steps below to receive live metrics data from the backend.

1. Find a serial number for a charger.
2. Connect to the following WebSocket:

```
ws://18.202.253.30:1337/charger/CHARGER_SERIAL_NUMBER_HERE
```

3. While connected to the WebSocket from previous step, also connect to this websocket:

```
ws://18.202.253.30:1337/user/USER_ID_HERE
```

4. Start a transaction by sending a request to the following endpoint:

```
{{url}}/transactions/start/:transactionId
```

..where `{{url}}` is the address to the backend API.

There will not be any response immediately, it will come when the next step is performed. The transaction must be related to the charger which's serial number has been used to connect to the WebSocket in an earlier step.

5. Send a "Remote Start Transaction"-request to the charger WebSocket:

```
[
  3,
  "100009RemoteStartTransaction1665649626046",
  "RemoteStartTransaction",
  {
    "status": "Accepted"
  }
]
```

Replace the second property with the unique ID which you got from the websocket when you first connected to it.

6. Send a "Start Transaction"-request to the charger WebSocket:

```
[
  2,
  "100009RemoteStartTransaction1665649626046",
  "StartTransaction",
  {
    "connectorId": 1,
  }
]
```



```
"idTag": 1,  
"meterStart": 1,  
"reservationId": 1,  
"timestamp":1234512345124123  
  
}  
]
```

7. Send a "MeterValues"-request to the charger WebSocket:

```
[  
  2,  
  "100008RemoteStartTransaction1665082232426",  
  "MeterValues",  
  {  
    "connectorId": 1,  
    "transactionId": 1,  
    "timestamp": 1664957184974,  
    "values": {  
      "chargingPercent": {  
        "value": 50,  
        "unit": "%",  
        "measurand": "SoC"  
      },  
      "chargingPower": {  
        "value": 0,  
        "unit": "W",  
        "measurand": "Power.Active.Import"  
      },  
      "chargedSoFar": {  
        "value": 0,  
        "unit": "Wh",  
        "measurand": "Energy.Active.Import.Interval"  
      }  
    }  
  }  
]
```

The property `values.chargingPercent.value` is used to set the charging level returned in the response. The reason why this is controlled by the client is that the implementation is not completed on the backend (2022-10-13).

# FlexiCharge Backend API

The FlexiCharge Backend API has been documented with Swagger and the documentation can be found here : <http://18.202.253.30:8080/swagger/>

## Postman Collections

The following collections can be used to test the backend API in Postman.

Collection	Link
Chargers	<a href="https://www.getpostman.com/collections/79c8b746a6f5e84f3b4b">https://www.getpostman.com/collections/79c8b746a6f5e84f3b4b</a>
Users	<a href="https://www.getpostman.com/collections/76955c65a73d97a81386">https://www.getpostman.com/collections/76955c65a73d97a81386</a>

The source code for the Postman collections can be found here:  
<https://github.com/knowitrickard/FlexiCharge-Backend/tree/main/postman>

# Notes & Requirements

Likely useful for next year's Cross-Platform Group

## Run the app

In order to run the Flutter project in an android or iOS simulator, there are some steps that need to be done before.

### iOS

1. Download Xcode.
2. Install Ruby in a command-line interface (CLI). To check if Ruby is installed, type `ruby --version` in the CLI. The version is likely to have been updated since October 2023, but version 2.7 still works.
3. Download CocoaPods by following the instructions on the CocoaPods website: <https://guides.cocoapods.org/using/getting-started.html> .
4. Go to the ios folder in the project and run the command `pod update` to update the pod to the latest version.
5. Go back to the project folder and run `flutter run` on the iOS simulator. If this does not work, try running `flutter clean` & `flutter run` to clean the project before running it.

### Android

If your project does not run on the Android emulator, it may be because of some dependencies in the `pubspec.yaml` file, which is located in the root project folder. These dependencies may need to be updated or replaced.

### Additional tips

- Make sure the simulator/emulator is running before you try to run the app.
- If you are using an iOS simulator, make sure the simulator is set to the correct device type in Xcode.
- If you are using an Android emulator, make sure the emulator is set to the correct Android API level in Android Studio.
- If the project still doesn't start, try running `flutter clean`, followed by `flutter pub get` and `flutter run`.

## Google Maps API KEY

The Google Maps API Key for Android & iOS needs to be updated. A new project in Google Cloud Platform should be created. The Maps API for both Android and iOS should be enabled in the project, the API key is the same for both. The API key is placed in

`/android/app/src/main/AndroidManifest.xml` as well as in `ios/Runner/AppDelegate.swift`.

## Flutter Version

In order to avoid working in a legacy codebase, it is plausible that a version upgrade of Flutter is desirable. Most packages can be updated automatically to the latest compatible version using the command `flutter pub upgrade` in a command-line interface (CLI). Certain packages may need to be updated through a manual alteration of the version specification in `pubspec.yml` after updating the Flutter version.

## Testing Registration Functionality

In order to test register & login functionality, a temporary email can be useful. Sites such as <https://temp-mail.org/sv/> can be used for easily gaining access to a temporary email.

## Flutter Secure Storage

User data is stored on the user device with the package Flutter secure storage. The package encrypts the data before storing it on the device. This package only supports strings hence all data is converted within the functions located in the class for UserSecureStorage.

## Bug Tracking

Bugs that are found in the code are documented with issues in GitHub and labeled with “bug”.

## App Flow Prototype

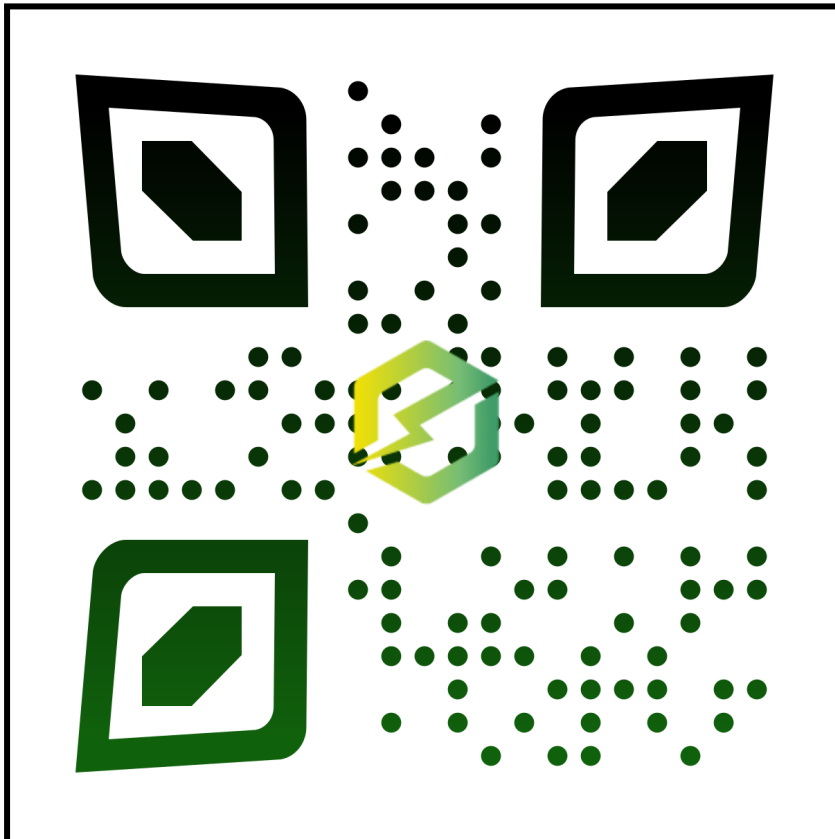
The views in the application and the navigation between the views has been illustrated in the flow chart diagram that was provided by KnowIT. The diagram can be found in the *documentation* folder in the repository.

## QR Code Scanning

Another option for finding a charger in the application is to scan a QR code on the charger when connecting the car. The following package is used for the QR scanner functionality:

[https://pub.dev/packages/mobile\\_scanner](https://pub.dev/packages/mobile_scanner)

A QR code has been generated at <https://www.qrcode-monkey.com/>. The QR code contains the value "100010" which is the ID of a charger. The QR code can be found below.



## Code Comments

Essentially the entire codebase is commented with the help of *Mintlify* which is an extension for Visual Studio Code. The comments are descriptive and inform the reader about the purpose and functionality of widgets, functions, classes or other data structures.