



Mobile iOS Native Documentation





Table of contents

Getting Started	3
Architecture & Code Standard	3
General View Guidelines	3
Account Views	4
ContentView	4
MapView.....	4
IdentifyCharger	4
QR Code Reader	5
Account Settings Page	5
Invoices	5
Charging progress.....	5
ChargerAPI	5



Getting Started

This project is developed using Swift in the XCode environment. To start testing the program you first need to clone the GitHub [repository](#). The repository consists of different branches but practically the ones to focus on are the **main** and **dev** branches. The **main** is a branch that should, ideally, always contain working code, and working features. **Main** is also protected, which means that no code can be directly pushed to it without being reviewer approved first. Since Swift projects are conflict heavy, meaning files needs to be manually edited to prevent conflicts, the **dev** branch exists to make sure all the features can be integrated and work accordingly before being pushed to **main**.

When a branch is fetched as well as pulled, all that is needed now to press the *build* button, located in the top left corner of the IDE.

Architecture & Code Standard

The project follows the [MVVM](#), Model-View-ViewModel, architecture. And as for code standard, the project follows Clean code. More information regarding the matter can be read on the projects [Wiki page](#) in GitHub (recommended read).

For colors there is an **Extensions** folder the developers use to call colors and make sure the same colors are used throughout the project.

The **UsefulValues** contains values used often in the FlexiCharge application. Values such as **static let** `screenWidth = UIScreen.main.bounds.width` and **static let** `screenHeight = UIScreen.main.bounds.height` that are responsible for making a lot of design work as intended.

TextFields, SecureFields, buttons and more are commonly used with the same design across the entire application. Commonly used items are therefore added to the **CommonUsage** folder and called on from there. This ensures a unity in design and more scalable code.

General View Guidelines

The views are, as of so far, developed using [SwiftUI](#), a set of tools and APIs. But the most fundamental functionality to know is the concept of stacks. The different stacks places items in a certain direction; ZStack places items on the Z-axis, VStack in a vertical axis and HStack in a horizontal one.

Declaring variables follows the intended implementations. Use immutable **let** for constants or **var** otherwise. Values used for view items should be initialized using **@State**.



As previously stated, to follow the same design standard for repeat items example given buttons and text fields, using the assets from CommonUsage is advised.

Account Views

There are currently three views related to a user's account. Register account, log in and recover password. These views are built using the same type of styling for the text fields, font size, navigation bar, colors and more. For the password fields, a so call SecureField is used.

Currently when registering an account, there is no confirm password field. This is because of a bug in SwiftUI that causes elements to resize when having multiple SecureFields.

For the validation checks, ViewModels are added accordingly under the same folder as the given view.

ContentView

The ContentView is the “main view”. Here is a map displaying chargers both occupied and free to use. There is one big round button anchored at the bottom of the screen, and adjacently placed are a settings button, camera button for scanning a QR code and lastly a pin button. All these views are split into separate folders and called to in the ContentView.

MapView

The code for the map in and of self is in the folder called Map. In the file MapView, location is set up. How the map logic works is done in LocationManager. To start using the map **import MapKit** was used in the ContentView file.

The API is used for displaying chargers on the map, and in the MapView displays pins for its location.

Contrary to popular belief, the API call for getting chargers,

<http://54.220.194.65:8080/chargers> does not belong at the API file. Instead, loading the data happens in the ContentView. Aware of MVVM, this is yet still implemented the correct way to make the map update the correct way.

IdentifyCharger

To enter a chargerID, the user presses the big Flexicharge button at the bottom of the screen. The following folder, IdentifyCharger, has files for every function needed to make the button work properly.

- **IdentifyChargerView** is the view and **IndetifyChargerViewModel** has its logic. The main job for these files is to let the user enter a chargerID and then get prompted with its corresponding message. Example given “Charger occupied” or “Begin charging”.



- **ChargerList** is a list of chargers. Each row of this list is called from the **ChargerRowView**. Each row displays a name of the charger point and how many chargers that are available.
- Clicking on a row navigates to the **ChargerHubView**. A view where a charger and payment method can be selected.

As of now, the list and is filled with mock rows since it is not yet implemented in the backend.

QR Code Reader

To scan a QR code the `CodeScanner` library is imported in the `ContentView` with `import CodeScanner`. Furthermore, the design is located at `QROverlayView`.

Account Settings Page

The account button leads to a settings page where the user has several options. Example given to set up an invoice. Navigation to a view is done via the `NavigationLink`.

Invoices

One of the options in the account settings page is to set up invoice information and to view both upcoming and past invoices. What displays depends on two variables inside the `InvoicesView`: `var isInvoiceSetUp: Bool` and `var isInvoiceEmpty: Bool`. If a user does not have their invoice set up, the view will prompt the user they have not and display a button leading to a view from `SetUpInvoice` will be called. Else if it is set up, but there are no invoices they view will say that as well. Finally, if the user has set up their invoice, and the user has invoice(s), a list will display them.

Charging progress

After a charger and a payment method has been chosen, it gets reserved, or occupied, the charging progress view geographically placed at the top of the screen will show. This is loaded from, but still displayed on the `ContentView`, the `ChargingProgressView`. The file also makes use of the `ChargerAPI` to stop the charging.

ChargerAPI

Navigating to the `Models` folder and then to `ChargerAPI`, API requests for the charger are found. The `http://54.220.194.65:8080/reservations` is one example, used in the begin charging function: `func beginCharging(chargerID: Int) -> Any`. This is a put request, asking the backend to make a reservation for the charger, ultimately letting the user start a charging session.