GITHUB:

UnlockIT

UnlockIT

(62417) Mobile Application Development With Swift

Anton Lage s191270

Jonas Jensen s205129

May 4, 2023

Contents

1 Introduction

This is the project report for the semester project in course 62417 - *Mobile Application Development with Swift*. The course focus is on the use of the Swift language, as well as the use of Apple's developer tools, such as Xcode, and many of its features. Doing a project, ensures that we as student obtain experience with the technology, through actual work, and that we do not just read about how to do things.

1.1 Project Description

The project we decided to do, is a digital key and access management system, which we named UnlockIT. It is a solution to businesses, where they can manage their employees, and their facilities, and configure who has access to which rooms in a building. This is all configured by one or more administrators, who can set security levels for each employee at a company, as well as configuring rooms to require a certain security level, for employees to unlock it. A room can then contain any number of locks (which would probably be one for each entrance). A room can also have a list of cleared employees, who can enter, no matter if their security level is high enough for that specific room.

Besides having the capability to manage employees and locks, the app also functions as the key itself. It should have used NFC to communicate with the physical locks, but for reasons which will be explained later, it uses WiFi instead.

2 Analysis

This being a course targeting the swift language, with focus on mobile devices (iOS), there was not a whole lot of things to consider when choosing the development environment.

What was more important to consider was our communication interface with the locks, and which platform we wanted to use for our backend.

The obvious choice was to use NFC, and we initially looked into going that way, but Apple does not allow people to use this capability in their devices, unless you pay for a full developer license. This was a shame, but there was not much we could do about it (except to pay up, but expenses like that is not included in our SU budgets). We then had two options: WiFi or Bluetooth. Jonas had another course, in which he had to do a project involving some micro controllers and some hardware. In that course he chose to develop a WiFi enabled lock as his project, so that we could combine the two. Beacuse of this, the choice for communication technology ended up being WiFi, but in the real world we would almost definitely have used NFC.

Another main thing we had to consider was our backend. The course included lectures about Firebase, and we thought it natural to go with this, and use it for our authentication system and data storage. None of us had used it before, but it was a great opportunity to learn, and familiarize ourselves with this platform.

The ”second part” of our backend would consist of a web application [4], acting as the broker between our two end-devices (lock and iPhone). We decided to go with this solution, since it is easier to maintain TCP connections through our web server, because it is always running. Furthermore, the authentication part of our overall system, includes symmetric encryption (AES-ECB mode) which seems to be not so easy to do in Swift. The reason for this, is because it is not considered a secure mode of operation in some cases, because identical cipher text will be encrypted into identical plain text, and Apple has a history of making it hard to do the ”wrong things” [3].

It should be mentioned that more ”appropriate” authentication mechanisms exist, either via hashing functions or symmetric cryptographic based authentication. However, this functionality doesn't exist on our microcontroller, where we are implementing our lock, which is why we are using AES-ECB mode. The feature that makes it insecure it also the feature that makes it possible to implement authentication - the property, that identical cipher text is encrypted into identical plain text, which makes it possible to compare two encrypted plain texts and check for equality.

3 Design

The app has been designed to function in two modes: Administrator mode and user mode. For now, the only thing a user can do, is to unlock locks. We talked about doing a booking functionality of some of the rooms, but this has not been implemented yet. This would include all the rooms configured to be bookable, like meeting rooms and other shared facilities, but so far the screen just says *Under Construction....*

The unlocking of a lock is done by pressing the big logo in the middle of the home screen, which brings up an unlock interface. This would be Apples built in NFC scanning card, but since we do not have access to that feature, we have made an emulator, that looks the part, and have a button for unlocking the WiFi enabled lock.

An administrator also have the option to unlock a lock, but in addition to that, he or she can also either manage users or manage rooms. Under manage users, an administrator has the options to create new users or to edit or delete existing users. Under manage rooms, the administrator has a few more options. Here the option to create, edit and deleting rooms exists, but there is a lot to that. A room can be configured with any number of locks, and a security level required to unlock those locks, which is the same for all the locks related to a room. Also, individual users can be authorized to unlock the locks to a room, regardless of their security level.

3.1 Overall System Architecture and functionality

The ”inner-working” of our overall system consists of three parts: Our lock, web application, and SwiftUI application. Our web application is created with Spring Boot in which we have created a REST API with three endpoints with the following functionality, respectively:

1. Activate lock (Fetch Encryption key from lock)
2. Initiate Challenge (Lock generates NONCE, encrypts it and transmits NONCE in plaintext)
3. Respond to challenge (Web server encrypts NONCE, transmits it and lock compares to deny or grant access)

All of the endpoints are called from our SwiftUI application, in which an ID is appended as a parameter, such that the web server can communicate with the appropriate lock. The flow of activating and unlocking a lock, can be seen in figure **1** and figure **2**, respectively.

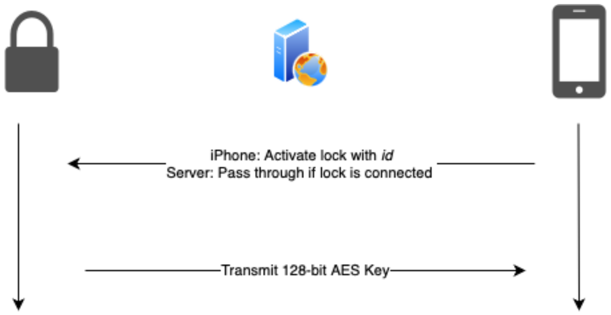

iPhone: Activate lock with *id*
Server: Pass through if lock is connected
Transmit 128-bit AES Key

Figure 1: Activate Lock Flow

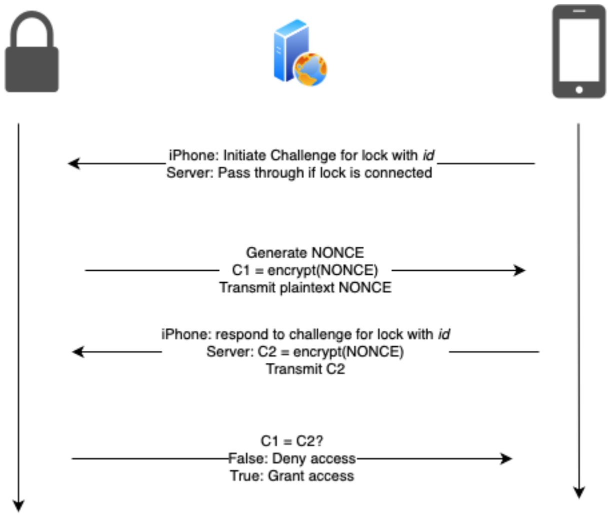

iPhone: Initiate Challenge for lock with *id*
Server: Pass through if lock is connected
Generate NONCE
C1 = encrypt(NONCE)
Transmit plaintext NONCE
iPhone: respond to challenge for lock with *id*
Server: C2 = encrypt(NONCE)
Transmit C2
C1 = C2?
False: Deny access
True: Grant access

Figure 2: Unlocking Lock Flow

4 Implementation

4.1 Data classes

For this application there are a couple of crucial data classes that have been created also to be compatible with Firebase persistence. All of the ”crucial” data classes are passed down as environment objects from the top-level view as can be seen in listing 1.

```
@StateObject private var appStyle = AppStyle()
@StateObject private var user = User()
@StateObject private var roomsModel = RoomsModel()

var body: some Scene {
      WindowGroup {
          MainView()
              .environmentObject(appStyle)
              .environmentObject(user)
              .environmentObject(roomsModel)
      }
}
```

Listing 1: Top level view.

4.1.1 User Model

Much of the app revolves around a user, and what that user can do and is allowed to do. The user object is accessible throughout the app as an environment object. To represent this user we have written a class, with fields for user data and state, and some methods for various different things. When a user is logged in to Firebase Auth, the User class is synchronized with the user data contained in our Firebase Firestore database.

As mentioned, the user class also contains some different state variables, for the UI part of the app to look at, and render a View accordingly. These state variable includes if the user is logged in, if the user is an admin, if it is the first time a user is logging in and finally if the user has been locally authenticated. The methods contained are related to persistence, where the users credentials are stored locally on device, in `User Defaults` or `Keychain`, as well as local authorization of the user.

To handle the local authentication of the user, the User class has two methods. One method utilize the `LocalAuthentication` API to user either `FaceID` or `TouchID` to authenticate the user, and if successful, update the authentication state to authenticated. The other method simply clears the authentication state, so that the user will have to be re-authenticated the next time a restricted part of the app is accessed.

4.1.2 Rooms model

Another crucial data class for this application is the RoomsModel class. To minimize the complexity of the code, we have decided to set this as an environment object in the application, which means the class itself must conform to ObservableObject. It only has one published property, which is a list of room structs, that all contain the relevant properties of a room in our context.

```
struct Room: Identifiable, Codable, Equatable  {
    @DocumentID var docId: String?
    var id: UUID = UUID()
    var locks : [Lock] = [];
    var newLock : Lock = Lock()
    var description : String = "Room Description";
    var authorizedUsers : [String] = [];
    var roomType : RoomType = .Meeting
    var bookable: Bool = true;
}

@MainActor
class RoomsModel : ObservableObject, Identifiable {
    @Published var rooms : [Room] = []
    private lazy var firestore = Firestore.firestore()

    ...
}
```

Listing 2: RoomsModel class.

4.2 Firebase

As mentioned we use Firebase for part of our backend service. Firebase offers a lot of different services and functionality, but we only use two things: Authentication and Firestore database.

4.2.1 Authentication

Each user is created by an administrator in the Authentication user-base. Here, a user is created with an e-mail and a password, and a `UUID` is generated by the service. When the administrator creates a user, the user is created with a default password (imagine this having some sort of company policy, e.g. creating users with their initials as a first password, when they are first employed). When a user then logs in for the first time, the user is prompted to change his or hers password to something they choose themselves.

In our app, the user model has a field for indicating if the user is logged into Firebase's authentication system, which is subscribed through the Firebase API. This means, that whenever the logged in state changes in Firebase, it also changes on the device, and SwiftUI can use that variable for determining which views to generate.

4.2.2 Firestore Database

For our database we have selected to use Firestore. Firestore is a document based database, and is organized using sections and documents. The way we have organized ours is in the following treestructure:

- Companies
    - DTU

    - Users
      - User A
      - User B
      - ...
    - Rooms
      - Room A
      - Room B
      - ...
  - AU
  - Another Company
  - ...

Each user and each room is its own document, where user documents are named with the related users uid. Rooms are named with the name given to them when the administrator creates them.

Reading, creating, updating and deleting documents in the User section of the database is handled by a controller called FirebaseUserController. This is a fully throwing and fully asynchronous class, with methods for doing the aforementioned operations. This means, that the UI thread can safely call any of the functions, without having to worry about freezing the UI, and that if any errors occur, that the app will not crash. The controller also handle signing into the Firebase authentication system, as well as creation of new users in the Firebase authentication system.

When reading, writing, updating or deleting a room, the app uses the same approach as when persisting users (async/await). It should however be mentioned, that some functions for creating/updating a document relies on the firebase API, which takes an encodable object and isn't an async function.

4.3 Persistence

In this app we have the need to store the email and password which was last logged in with. This is only necessary for a very specific scenario, which is when an administrator creates a new user. When doing so, the account is automatically signed into in Firebase's authentication system. To sign back into the administrators account, using stored credentials is necessary.

Credentials are stored and loaded through calling methods in the User class.

4.3.1 User Defaults

The logged in user's email is stored and read in `User Defaults`, using the very simple API:

```swift
UserDefaults.standard.set(email, forKey: credentialKeys.emailKey)
UserDefaults.standard.string(forKey: credentialKeys.emailKey)
```

Listing 3: User Defaults API Snippet.

4.3.2 Keychain

The Keychain is a little bit more tricky to work with, but not much. To simplify things, we wrote a `KeychainManager` class, to wrap the interaction, and isolate it from the rest of the code. The `KeychainManager` has two methods. One method for storing a value for a key, both provided as a function parameter, and another method for fetching a value for a given key, also provided as a function parameter. Both functions throws if, any errors are detected.

4.4 Lock Communication

As mentioned earlier we have created a web application acting as a broker between the lock and the Swift Application. The three endpoints mentioned in section 3.1 are called in an async/await approach with proper error handling. Note that the errors thrown in the function must be handled in the view where the function is called. First we make sure, that the response type received from the server after the asynchronous call on line 10 is of type HTTPURLResponse (line 12). Next the actual status code is checked which is a property of the httpResponse, which should be in the range of 200-299, if everything went right. At last, we try to decode the response with encoding UTF-8 and return the String in case no errors were encountered. The code for calling the other two endpoints is exactly the same, except for the URL itself.

```swift
1  func activateLock(id: String) async throws -> String {
2      guard let url = URL(string: "http://139.144.66.167:8080/api/activate?id=\(id)") else {
3          throw URLError(.badURL)
4      }
5
6      let request = URLRequest(url: url)
7
8      let session = URLSession.shared
9
10     let (data, response) = try await session.data(for: request)
11
12     guard let httpResponse = response as? HTTPURLResponse else {
13         throw URLError(.badServerResponse)
14     }
15
16     guard (200...299).contains(httpResponse.statusCode) else {
17         throw URLError(.badServerResponse)
18     }
```

```

    guard let responseString = String(data: data, encoding: .utf8) else {
        throw URLError(.badServerResponse)
    }

    return responseString
}
```

Listing 4: Lock Communication Snippet.

4.5 Localization

This app has been thoroughly prepared for localization, and have been localized, so that the app supports both English and Danish (which are the only languages fully implemented in the developers).

4.5.1 Preparation

By preperation, what is meant is, that the whole app has been gone through, and all text which might be presented through the UI to the user, has first of all been enabled for localization, and has had a comment attached to it, explaining the text's context. This entails that all string literals which might be presented in the UI has been converted from the old method of using string literals, to the new (both show below). This has been done to allow for both to localize the contents of the string, but also attach a comment to the contexts. Also, Labels and Buttons have been converted from the old form, to the new (both shown below), again in order to attach comments to their content for context [1]

```
// Old Methods:
stringVar = "some text"
Label("text", icon: "iconName")

// New Methods:
stringVar = String(localized: "some text", comment: "explaining context")
Label {
    Text("text",
         comment: "localization comment")
} icon: {
    Image(systemName: "iconName")
}
```

Listing 5: Localizing content.

4.5.2 Pluralization

Besides translation of the static text in the the app, rules for pluralization has also been set up, for the single piece of text in our app, which contains a count of something. More specifically, it is the label stating how many total users exists, in the `ManageUsersView`. The rules for pluralization has been created for both Danish and English, and lists as following:

	English	**Danish**
Zero	No Users	Ingen Brugere
One	1 User	1 Bruger
Other	%lld Users In Total	%lld Brugere I Alt

Table 1: Pluralization rules for number of total users

5 Tests

To test the app, we have implemented both some unit tests and some UI tests. It is however safe to say, that we do not have 100% code coverage, but we believe that we have written enough tests to demonstrate that we know how, and that we have familiarized ourselves with Apples testing frameworks.

To run the tests, and to test stuff like localization, we have configured two testplans, with some different configurations. When running a testplan, all the selected test are run. Configurations can vary in things like region and language settings, simulated location, memory settings, concurrency settings and so on. The two testplans we have configured is for testing both English and Danish things, and they run different test. The english one runs the most, and the danish one run a subset of the same tests, but where it looks for danish text instead.

Besides automatic testing, we have of course also tried out the app manually, whenever we have implemented new things.

5.1 Unit Tests

With unit tests, we test a couple of different things. We test that the correct things happen when local authorization is performed, and cleared. We test that the users can be sorted correctly, between administrators and non administrators. We test that storing and fetching from the keychain works correctly. And finally we test the RoomModel.

When doing our unit testing, we have to provide know data, and we have to *Mock* objects, that we do not intend to test. For example the local authorization API. When testing this, we just want it to act as if the user is being successfully authenticated, but without it really happening. To do this we override the methods we use in the API, and pass that to the methods using the API.

5.2 UI Tests

With UI testing, we are testing that the apps UI behaves correctly, based on which state the user is in. This is, for example, that the login options are available when the user is not logged in, and that the controls to change a password are available when it is the first time a user is logged in. We also test that only administrators have the option to access administrator controls. We also test, with the help of the testplans, that our localization of the pluralrules work.

When UI testing, we cannot access environment objects directly, to alter the state of the application. Instead, we have to pass arguments and set environment variables, which can then be read by the app during test execution [2]. For example, we do no want the app to use the Firebase login, but we instead want to control if an administrator or non administrator is logged in, if it is the first time a user is logged in or that maybe no user at all is logged at all, in depending on which test is running.

To help manage this sort of configuration, we have created some helper classes, which can be used to instruct the app on how it should be configured.

6 Conclusion

In conclusion this project has been a success. First of all, and most importantly, we have both familiarized ourselves with Xcode and the Swift language. Besides this, we have managed to develop a functioning app, incorporating most of the things Ian has talked about during his lectures. The app is obviously not production ready, and we have had to deviate from the optimal solution with NFC, and use WiFi instead, but in general, we are satisfied.

This project has stood out from many of our other projects, in the sense that we have had more focus on the completeness of the app, in this project than normal, and have focused less on implementing as many features as possible. An example of this is the fact that multiple languages are supported. That is not something we would normally have focused on for a school project, but getting to know a lot of the tool provided by Apple and Xcode has been rewarding.

As mentioned, we feel that we have gained experience with most of the technology taught in this course. We have gotten familiar with Swift UI, and how state management is done in relation to the UI. We have implemented asynchronous code, getting familiar with both async/await syntax, and completion handlers. We have implemented error handling in much of our functionality. We have tried writing extensions to existing types. We have utilized many of apples provided API's, for interacting with the phone itself, including local authorization, user defaults, keychain, localization, and more, as well as how to communicate from the phone via web requests. We have tried using third party libraries, here for communication with Firebase Firestore and Authorization. And we have gotten to know Apples testing framework, XCTest, by writing both unit tests and UI tests, as well as their testplan solution.

References

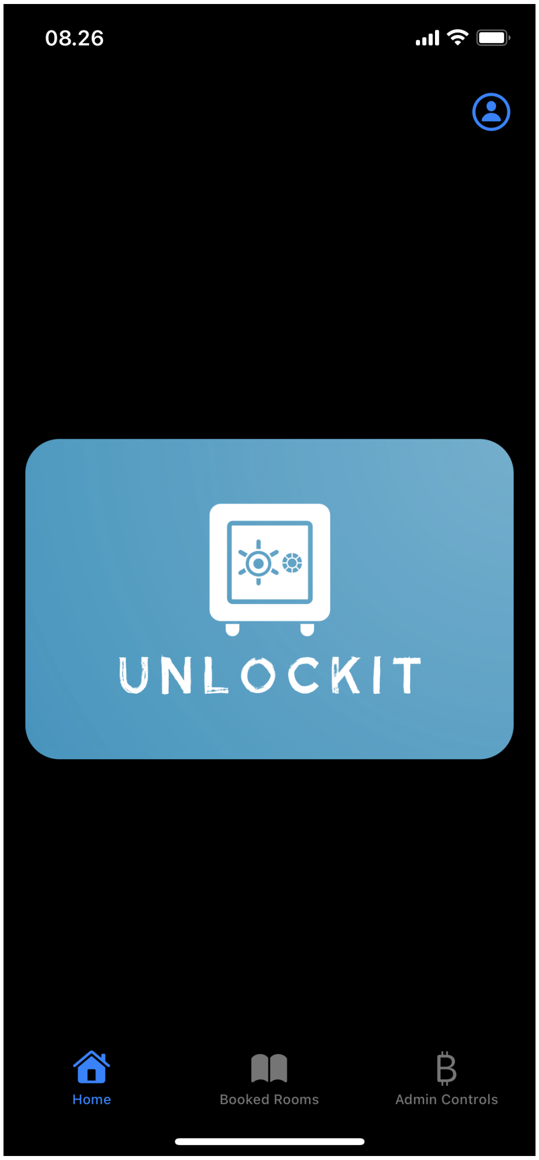
[1] Preparing Views For Localization

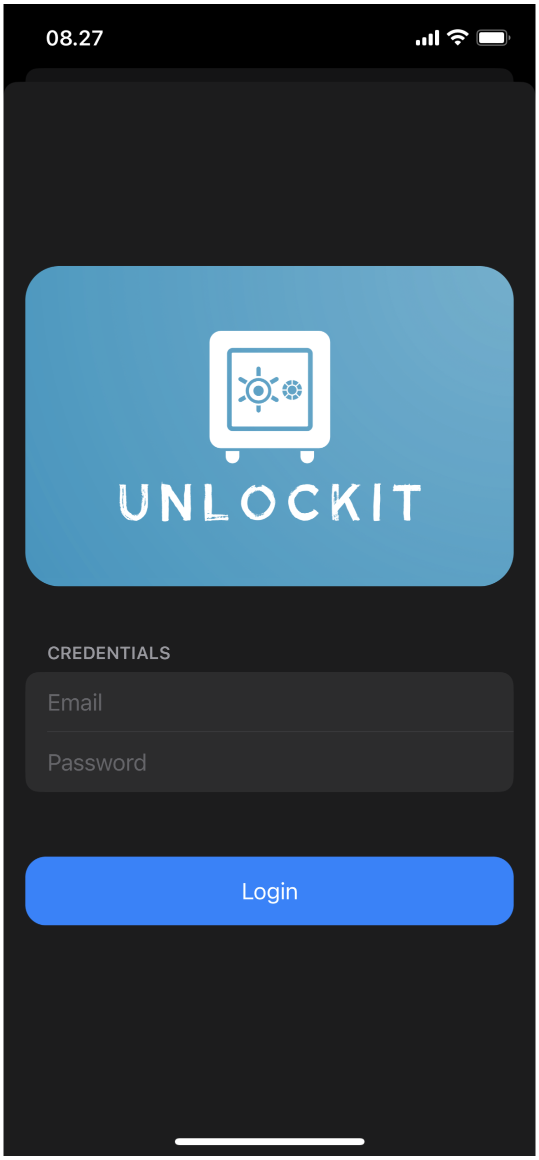
[2] Getting Started with XCUI Test in Swift

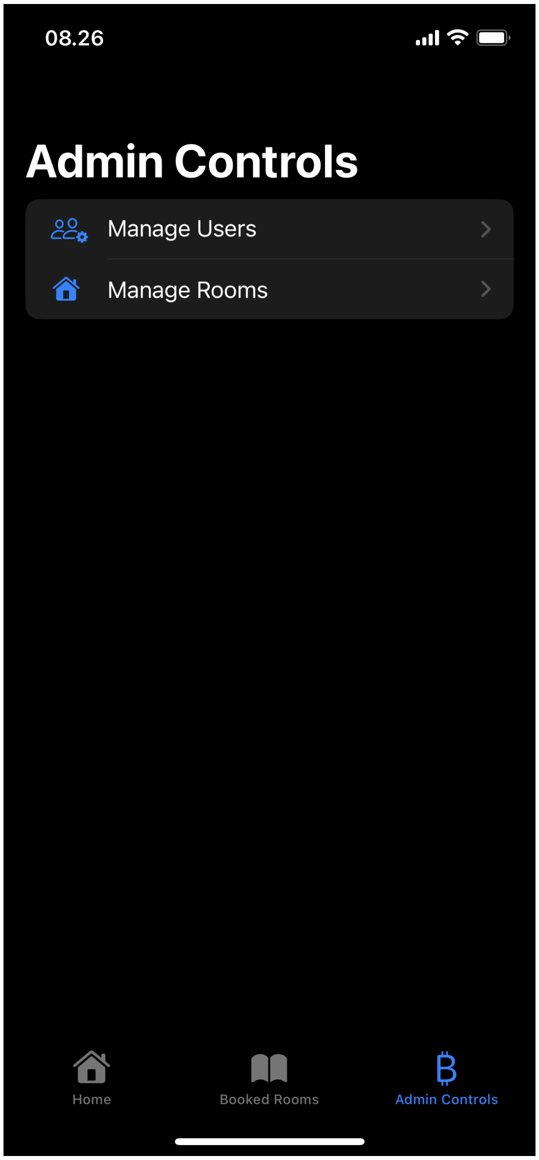
[3] AES ECB (128 bits) — Apple Developer Forums

[4] UnlockIT broker web application

A Screenshots





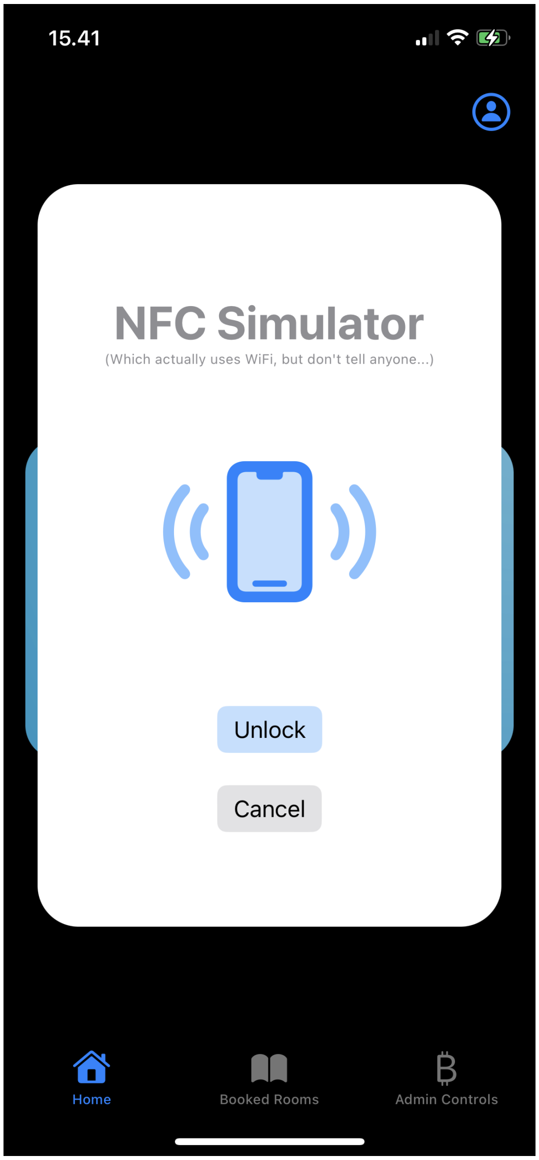


08.26

Admin Controls

Manage Users

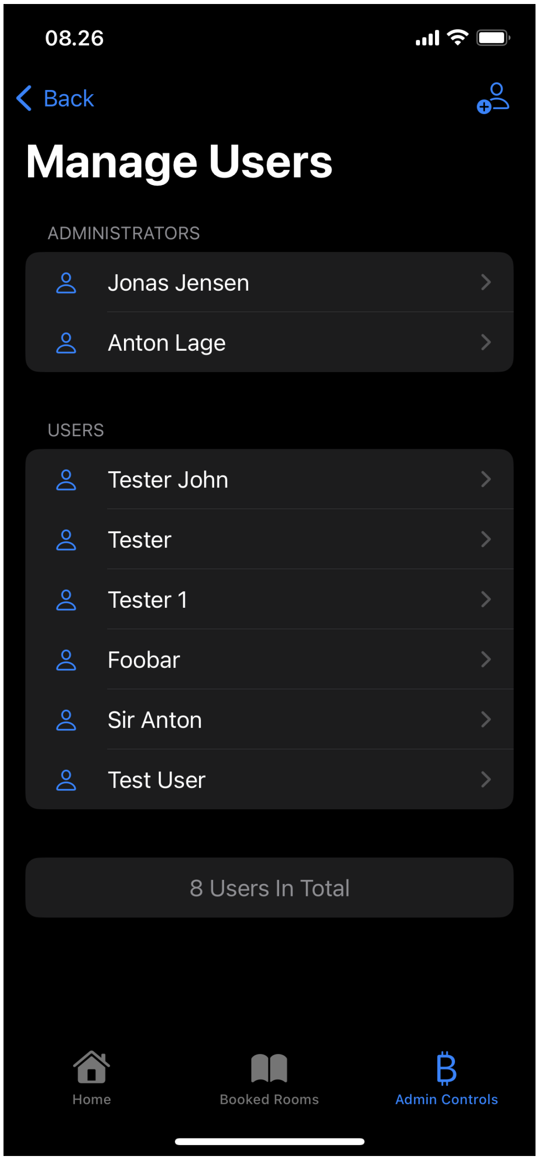
Manage Rooms



15.41

NFC Simulator

(Which actually uses WiFi, but don't tell anyone...)



Manage Users

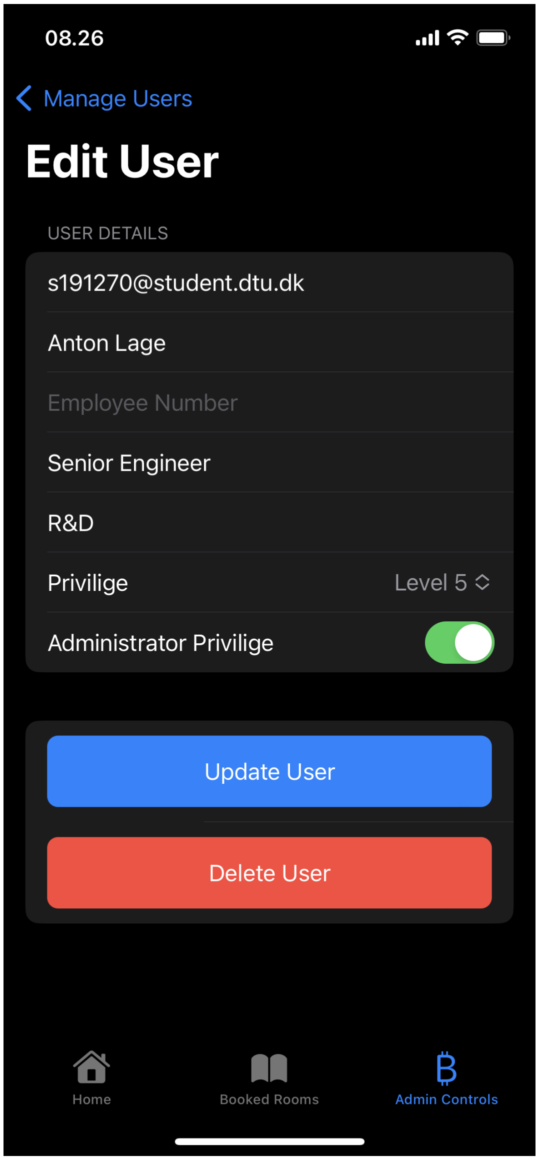
ADMINISTRATORS

- Jonas Jensen
- Anton Lage

USERS

- Tester John
- Tester
- Tester 1
- Foobar
- Sir Anton
- Test User

8 Users In Total



Edit User

USER DETAILS

s191270@student.dtu.dk

Anton Lage

Employee Number

Senior Engineer

R&D

Privilige: Level 5

Administrator Privilige: on