Bachelor's Thesis

Geolocalization and routing in complex multi-floor hospital environments

UC Odisee

Department of Engineering Technology - Electronics and Information Technology,

Specialisation Information Technology

Joachim Cardoen

2018

I want to thank my foster mother, Miek Roels for her devotion to guide me through the process of writing this thesis. I want to thank her and both Nathan Verhoeven, one of my closest friends, as well as Cedric De Roover (Senior Consultant Manager at IBM) to correct my thesis where necessary. I also want to thank my mentor Sven Sanders for answering my e-mails day and night whenever I had got stuck.

Abstract

The aim of this project is to realize a mobile application for patients of the hospital CHC Saint-Jean in Liege that will enable them to register, view appointments and find their way in the hospital. First the existing application for consulting scheduled appointments and registration is reworked to a native iOS mobile application (mobile app in short). This feature is extended with the option to view the information of the hospital and set reminders for any appointment in the near future. Second an indoor location framework is added to the reworked mobile app that shows the route to the place of appointment. After implementing the geolocation and routing, an optimization method is implemented: the ant colony optimization algorithm. The ant colony algorithm determines the shortest path to the place of appointment based on some factors such are: amount of visitors in some corridors and hallways and the amount of stairs a patient has to climb (a patient's mobility). Using the mobile app the hospital can improve their safety procedures by notifying users of the application of any problems in the hospital, these problems can be but are not limited to: fire, malfunctioning of the elevator and electrical outage.

Contents

Acronyms

API application programming interface. 4, 8–10, 19

EU European Union. 19

PoC proof of concept. 8, 10, 17

SDK software development kit. 8, 9

UI user interface. 17

UML unified modelling language. 9, 19

UX user experience. 17

Chapter 1

Summary

Chapter 2

Project Specification

2.1 Project Description

The goal is to develop a proof of concept (PoC) in both iOS and Android for a hospital. Firstly the existing application is reworked from using the Ionic framework to a native mobile application (Swift for iOS and Kotlin for Android). In addition to this part, geolocalization is implemented in the native mobile app using the MapWize service [11] and the IndoorLocation framework [9], both service provide working software development kit (SDK) for iOS and Android. Finally the application is revised by the team of interns and the developers at IBM and uploaded onto the Apple Store and the Google Play Store.

2.1.1 Technical Design Specs

The communication with the hospital happens with a server provided by IBM and the hospital's application programming interface (API). This means that the mobile device interacts with a intermediary server from IBM which in its turn communicates with the API of the hospital. This model is an example of a highly reusable architecture. If another hospital needs to be attached to the IBM server, only a small 'translator' for the endpoints of the additional hospital's API needs to be created whilst the structure of the IBM server remains the same.

2.1.2 Features

The main features of the project are specified below [8]:

1. Login with hospital provided credentials;
2. Synchronization of appointments with the hospital;
3. Ability to set reminders for an appointment;
4. See the hospital's location (and venues) as well as contact details;
5. Allow geolocalization inside the hospital;
6. Provide feedback after an appointment;
7. Localization in French, English and Dutch;
8. Available on both iOS and Android

9. Distributed in the Apple Store and Google Play Store;

2.1.3 Technologies to research

Throughout the development of the PoC, several technologies are used, such are: Android SDK, authentication, RoomDB for offline storage, IBM BlueMix API, unified modelling language (UML), dependency injection, MapWize, IndoorLocation and Cisco CMX.

2.2 Development Guidelines

To attain uniformity in the codebase of iOS and Android a 'Development Guidelines' document is written, this document can be found as an appendix.

Chapter 3

Technical Study

3.1 Integrated Development Environment

The application is written using Android Studio. Android Studio is an supported by Google and based on the IntelliJ of JetBrains, a company that develops an for the most popular general purpose programming languages.

3.2 Architecture

The emphasis of the PoC is on developing it in such a way that it should be easy to re-implement the application elsewhere. The PoC is developed in the two current formats for mobile development: iOS and Android. This bachelor's thesis will cover the implementation of the Android architecture.

3.2.1 Existing API

The existing API developed by IBM provides the application with a list of appointments (testing mode). An example of the JSON output can be found below:


```
[
    {
        "appointmentId": "000010130000001",
        "appointmentTime": "2017-10-16T14:14:00+02:00",
        "patientNr": 2000000420,
        "patientName": "SMITH, JULIE",
        "doctorNr": 123456,
        "doctorName": "FELDUNS, JEAN",
        "departmentId": "I",
        "departmentName": "USI",
        "siteId": "BO",
        "siteName": "Botanique",
        "appointmentReason": "Consultation Orthopédique",
        "appointmentInstruction": "La pendule fait",
        "appointmentStatus": "E",
        "appointmentStatusDescription": "Evaluated"
    }
```

]

Entities

The specific entities used throughout this application are:

- appointment;
- hospital;
- venues - the different locations of a hospital;
- address;
- additionalinformation - meta for a hospital and venues;
- department;

UML Diagram

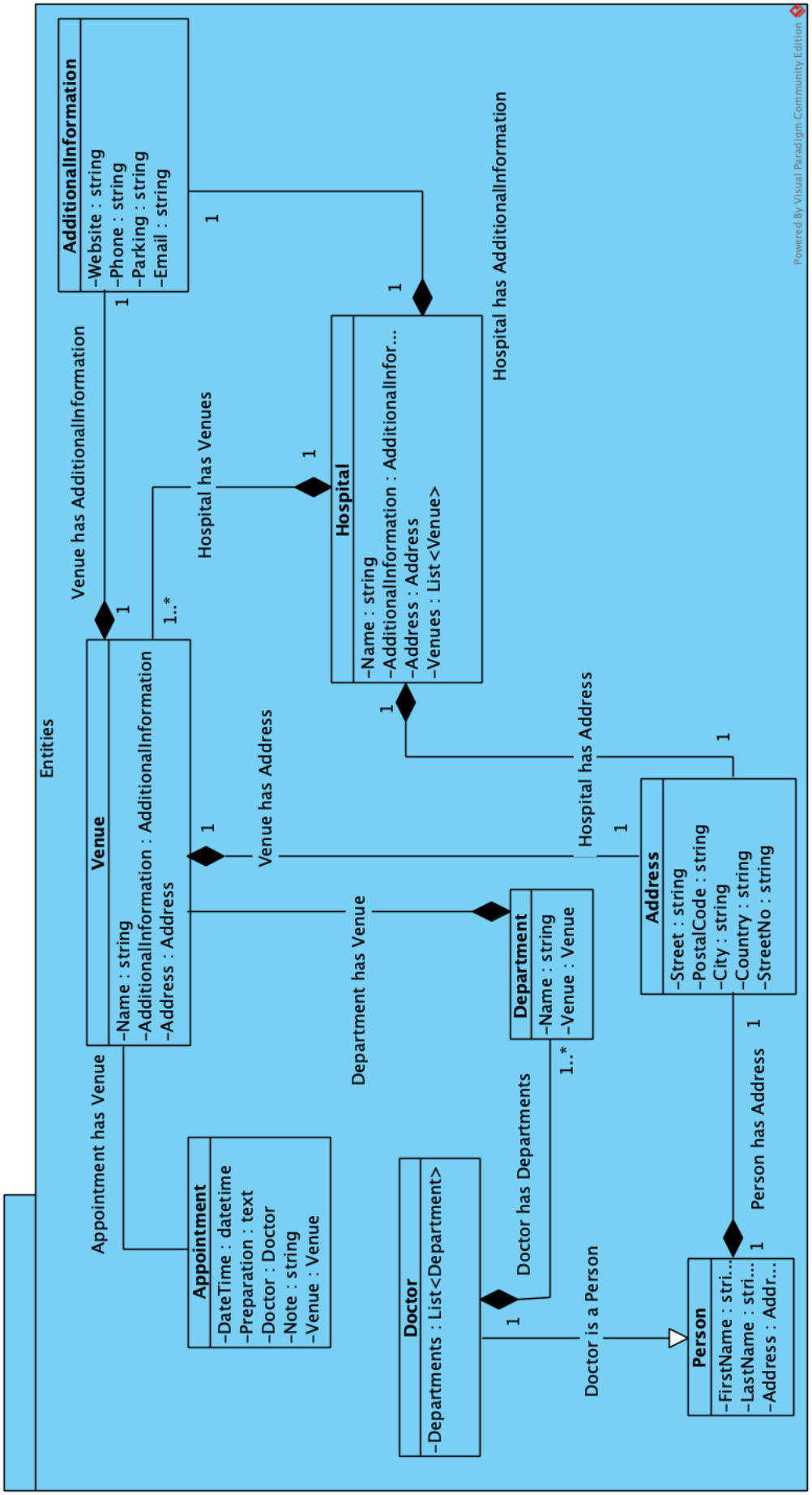

Figure 3.1: Unified Modelling Language diagram

3.3 Android Architecture

3.3.1 Testing

[7]

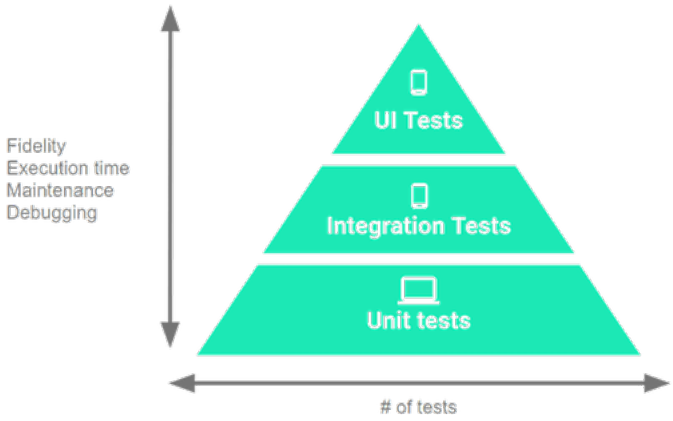

Figure 3.2: Testing pyramid [6]

Unit Testing

These tests are responsible for the smaller parts of the application (units) and use mocked or stubbed properties. This means that the properties and methods do not interfere with code written inside the main application. These tests are the fastest in runtime (compared to the integration and UI testing) because they do not need a running device or emulator, which means they are the least expensive to execute. Some testing frameworks from Android are: JUnit4 and Mockito (Mockito is used to create mock instances of dependencies, classes, properties and methods) [6]. The characteristics of a good unit test [7]:

1. Thorough;
2. Focused;
3. Repeatable;
4. Fast;
5. Verifies behaviour;
6. Concise;

Integration Testing

These tests test the interaction between different units of the application. The tests do not cover any updates on the UI thread and test the units independently from the UI. A common framework used to write integration tests is Roboelectric [12]. When an application is in development the integration tests will check the behaviour of the different interacting units when the new feature is implemented. This way, the development team can roll-out features without breaking the current application.

User Interface Testing

3.3.2 Separation of concern: Dependency Injection

Separation of concern is a general convention amongst software developers. In practice it is harder to implement than it first seems. One of the core components of this pattern are dependencies: one class depends on the structure of another class. The dependency pattern enables developers to focus on their code without having to worry about the dependency. For a class it is enough to know how a dependency is structure, there is no need for the class to know how it is implemented. This is also the last of the SOLID principles, the principle of dependency on abstraction instead of concrete implementation [2].

Dependency Injection: Restaurant Analogy

To comprehend the concept of dependency injection let's have a look at a fairly common analogy". A man comes into a restaurant and takes a look at the menu. After having taken a close look, the man decides to order fish and chips. The waiter notifies the kitchen and tells the head chef that a customer ordered the fish and chips. Upon finishing the plating, the waiter brings the wonderful plate of fish and chips to the customer (the man). The man obtained what he wanted without knowing how his dish was prepared, the head chef knew what the customer needed and provided the meal.

Benefits of using Dependency Injection

Using dependency injection might seem somewhat bloated in practice, but there are some enormous benefits upon applying this pattern on an application. Some of these benefits are listed below [13]:

- Late binding: interchangeable services;
- Extensibility: reusable code;
- Maintainability: classes with a well-defined responsibility become easier to maintain;
- Testability: classes having a dependency can be tested separately - as a single unit;
- Enforces usage of loose coupling;

Types of Dependency Injection

Constructor injection is the type of DI (dependency injection) that uses a private field for the dependency and sets this field using a parameter inside of the constructor. Setter (property) injection uses a property of the class that requires the dependency and works via getter and setter methods. The dependency is individually set instead of passed as a parameter in the constructor. This is quite easy to understand but hard to implement in a robust way, this only works if the value passed in the setter is a good value. When dependencies are only used in specific methods, it might be easier to just pass them as parameters to that method, this is called method injection. This way of implementing DI is also simple and straightforward [14].

Dagger2 - Dependency Injection Library

To remove the abstraction of the different methods of dependency injection, the library Dagger2 (supported by Google) is used. This library sets up the required dependencies throughout the application

and registers them accordingly. Upon launching a class that requires a dependency, the Dagger2 instance will load them as specified in the AppModule. Since the architecture used in the application is mvvm, the ViewModel is injected into an activity. The following code contains an AppModule, AppComponent, ViewModelFactoryModule and a subclass of the Application class, used for casting an activity:

```
/**
 * Component to be used to handle dependency injection
 */
@Singleton
@Component(modules = [AppModule::class, ViewModelModule::class])
interface AppComponent {
    fun inject(target: AppointmentList)
}

/**
 * Module that can be used to declare dependencies
 */
@Module
class AppModule @Inject constructor(private val app: Application) {
    @Provides
    @Singleton
    fun providesApplication(): Application {
        return app
    }

    @Provides
    @Singleton
    fun providesAppointmentViewModel(app: Application):
    AppointmentViewModel {
        return AppointmentViewModel(app, providesWebService())
    }

    @Singleton
    @Provides
    fun providesWebService(): WebService {
        return Retrofit.Builder()
            .baseUrl(Constants.BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(WebService::class.java)
    }

    @Provides
    @Singleton
```

```
    fun providesAppointmentDao(db: AppDataBase): AppointmentDao {
        return db.appointmentDao()
    }

    @Provides
    @Singleton
    fun providesAppDatabase(app: Application): AppDataBase {
        return Room.databaseBuilder(
            app,
            AppDataBase::class.java, "app_database"
        ).fallbackToDestructiveMigration().allowMainThreadQueries().build()
    }
}



/**
 * Child Application Class to use for casting inside an activity
 * Create the AppComponent reference and uses this
 * for injecting inside an Activity
 */
class DependencyApplication : Application() {
    lateinit var appComponent: AppComponent

    override fun onCreate() {
        super.onCreate()
        appComponent = initDagger(this)
    }

    private fun initDagger(app: DependencyApplication): AppComponent =
    DaggerAppComponent.builder().appModule(AppModule(app)).build()

}

**
 * Factory to get the ViewModel by class for DI
 */
@Singleton
class DaggerViewModelFactory @Inject constructor(
    private val creators: Map<Class<out ViewModel>,
    @JvmSuppressWildcards Provider<ViewModel>>
) : ViewModelProvider.Factory {

    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        var creator: Provider<out ViewModel>? = creators[modelClass]
```

```
        if (creator == null) {
            for ((key, value) in creators) {
                if (modelClass.isAssignableFrom(key)) {
                    creator = value
                    break
                }
            }
        }
        if (creator == null) {
            throw IllegalArgumentException("unknown model class $modelClass")
        }
        try {
            return creator.get() as T
        } catch (e: Exception) {
            throw RuntimeException(e)
        }
    }
}

@MustBeDocumented
@Target(AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.RUNTIME)
@MapKey
annotation class ViewModelKey(val value: KClass<out ViewModel>)

/**
 * Module to inject ViewModel instances
 */
@Module
abstract class ViewModelModule {

    @Binds
    abstract fun bindViewModelFactory(factory: DaggerViewModelFactory):
    ViewModelProvider.Factory

    @Binds
    @IntoMap
    @ViewModelKey(AppointmentViewModel::class)
    abstract fun bindAppointmentsListActivity(vm: AppointmentViewModel):
    ViewModel
}
```

3.3.3 Kotlin Language

3.3.4 Lifecycle Events

For the duration of the runtime of the mobile application (from the moment the app is opened until it is closed) some events occur that are typical for an Android mobile application. A brief summary of these events is listed below (in chronological order) [1]:

- onCreate() - When the activity is launched (This can happen after the onDestroy() event);
- onStart() - When the activity is visible to the user;
- onResume() - When the user returns to the activity after an onPause() event occurs;
- onPause() - When activity is no longer visible;
- onStop() - When the activity is finished or destroyed;
- onRestart() - When the activity is restarted after a stoppage;
- onDestroy() - When the activity is shut down;

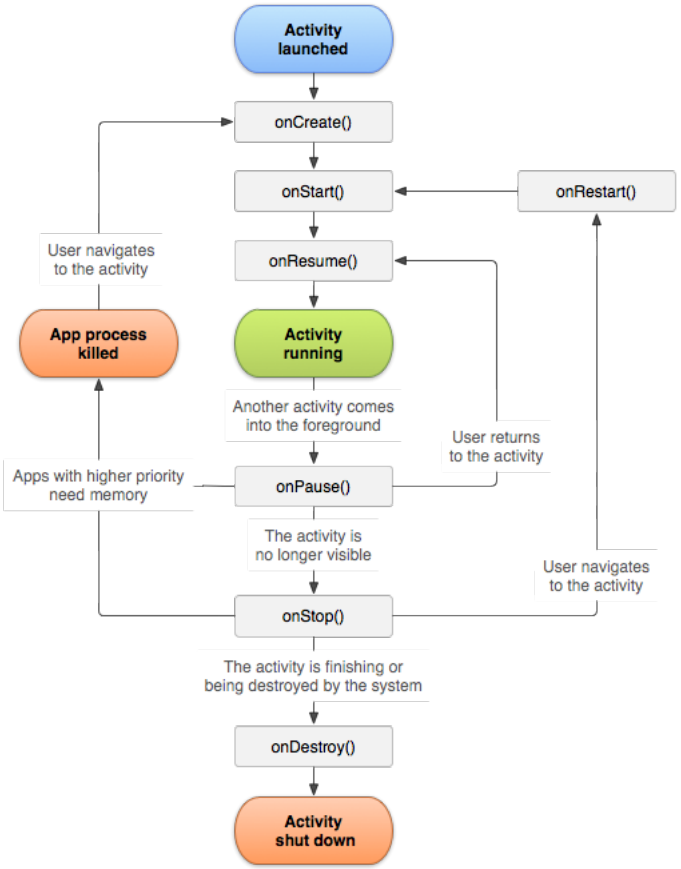

Figure 3.3: Android Activity Lifecycle Schematic [1]

The lifecycle of an activity is an important factor to take into account whilst the application is being developed. This means a certain level of persistency is required for an optimal user experience.

Bundles & Saved State

The way in which the onCreate() method is implemented allows a developer to declare a Bundle, which is an object that contains key-value pairs, that is used to restore an activity's previous state. If no such state exists then the Bundle will be equal to null. The Bundle object that is passed to an activity in the onCreate() method should only contain specific information such as user interactions: form fields, position on the screen and sometimes navigational properties. The main usage for this technology is when an activity gets paused or stopped, this means the OS (operating system) can freely destroy any activities [10].

3.3.5 Offline storage and persisting data

Another way to persist data throughout the lifecycle of an application is to use the (smart)phone's local storage. Each application can create a new local database using SQLite. SQLite is a transactional and file-based database (db), which means it is optimal for storing user-specific data. The fact that it is indeed a transactional db means that upon failure of an operation it will roll-back to the previous state and revert all existing, pending changes [15].

RoomDB

A nice feature from the Android SDK is a wrapper for SQLite inside the app: RoomDB. RoomDB is a feature set for SQLite statement and works using the repository pattern. The interaction between the application (view layer) and data layer happens using a repository which can be implemented locally (offline storage using the RoomDB wrapper) as well as remotely (remote API calls). The structure of the application is as follows: The local repository interacts with the local repository using a database

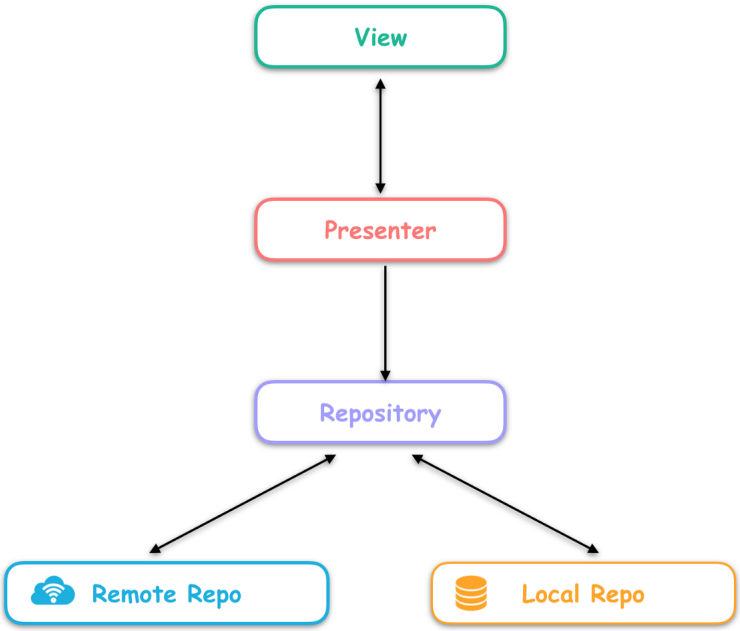

Figure 3.4: Repository Pattern inside an android application [4]

access object, which specifies the different possible statements that can be executed on the database (CRUD operations: create, read, update and delete) and maps them to functions.

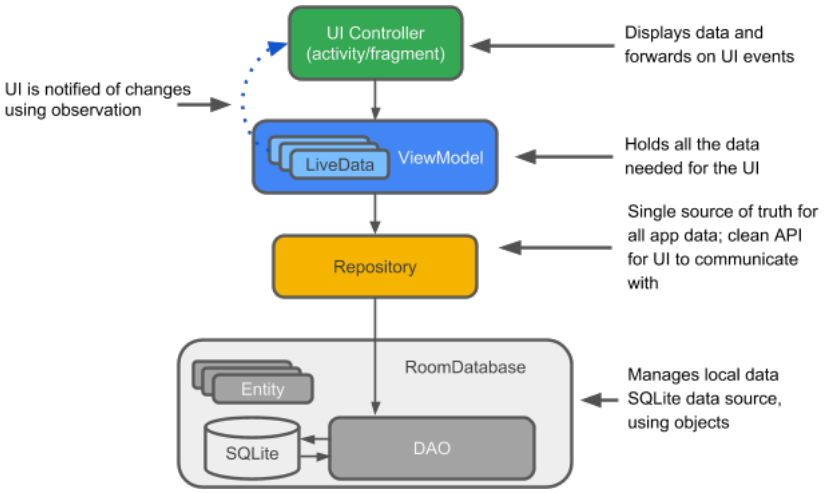

Figure 3.5: Local repository usage inside the applocal [16]

Application Cache

Strategy and abstractions

3.3.6 Model - View - Presenter Architecture

Model

View

Presenter

3.3.7 Model - View - ViewModel Architecture

ViewModel

Asynchronous Data

HTTP(S) Requests

Retrofit

LiveData Datatype

Observables

Chapter 4

Proof of Concept

4.1 User Interface

Considering the application is only a PoC there is almost no focus on the proof of concept nor on the user experience (UX). To at least give a slight indication of what information needs to be displayed where, a UI mock-up is created in Adobe Xd, Adobe Xd is a lightweight, rudimentary visual editor that enables designers to quickly develop and share interactive prototypes. A few example screens of the UI prototype can be found below.

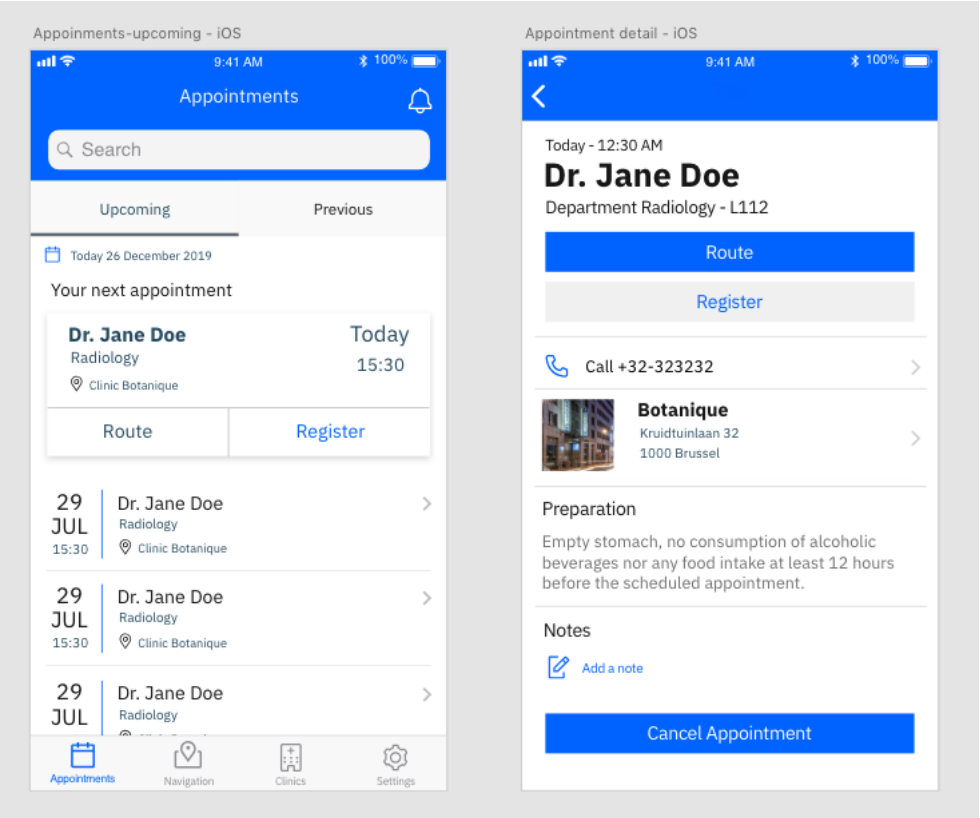

Figure 4.1: User interface of the appointments and detailed view for iOS

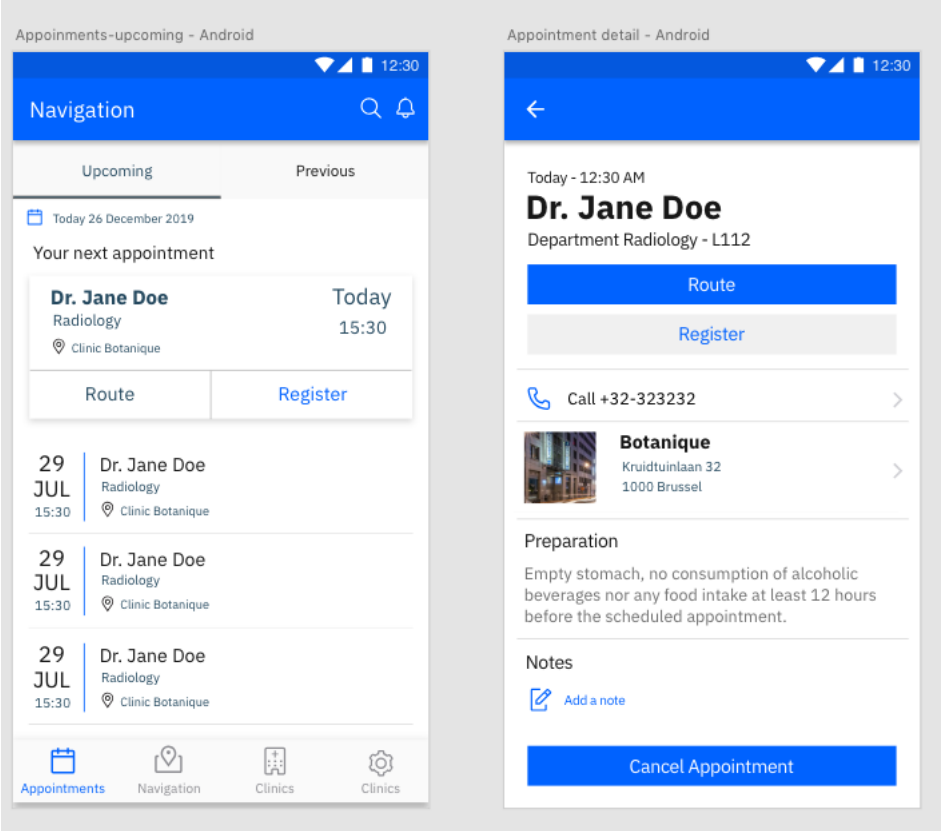

Figure 4.2: User interface of the appointments and detailed view for Android

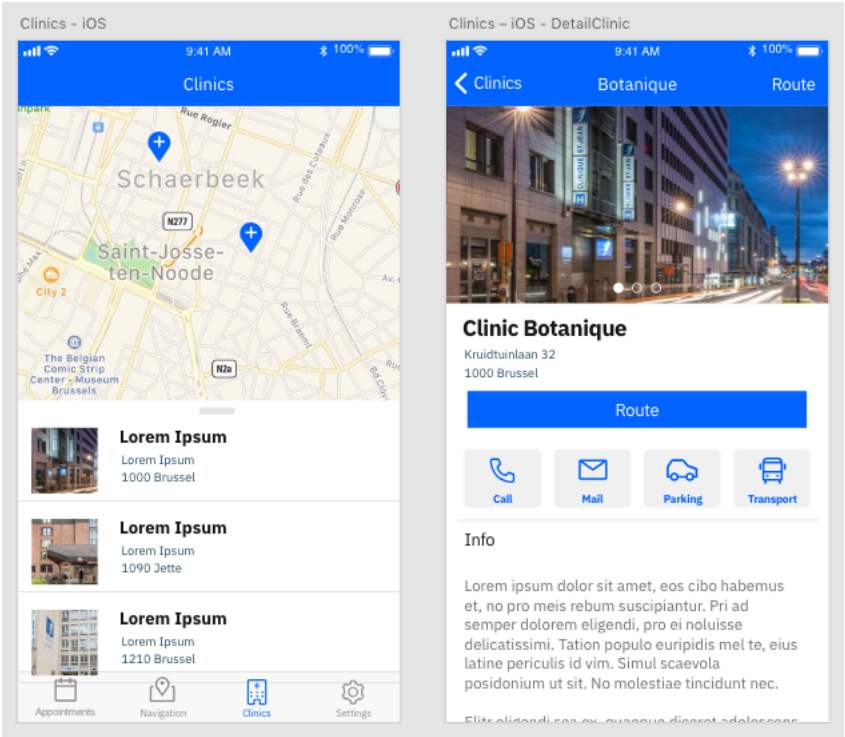

Figure 4.3: User interface of the hospital venues and detailed view for iOS

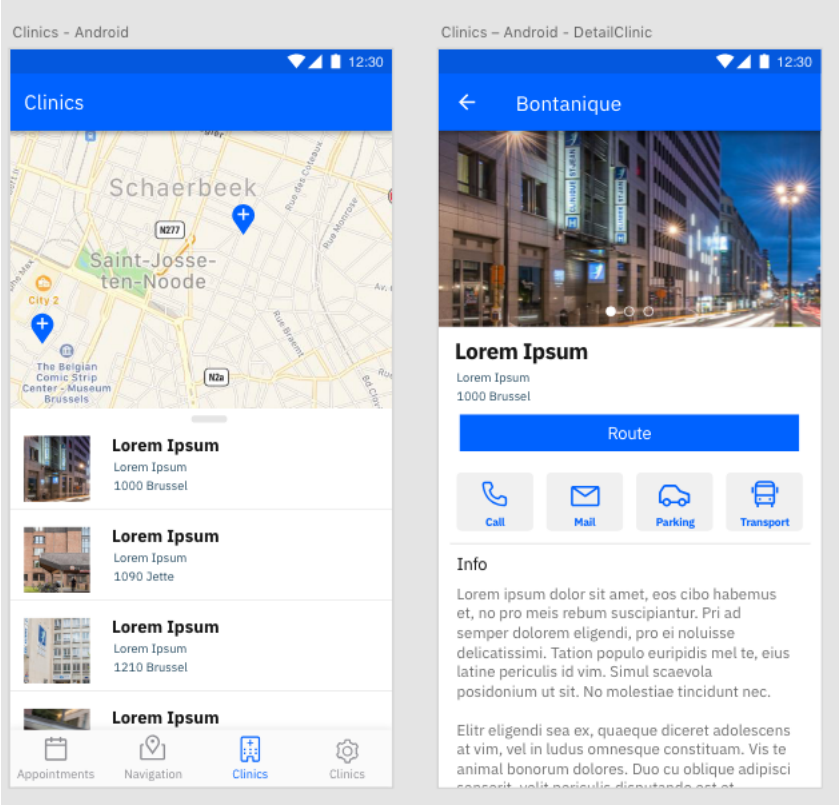

Figure 4.4: User interface of the hospital venues and detailed view for Android

4.2 General Data Protection Regulation

General Guidelines provided by the European Union (EU)

4.3 Database Communication

4.3.1 Testing API

To aid in testing the database a test API is programmed using a Node.js framework: Express. it is a very simple tool to create a web API ready for consumption [5]. The testing API is structured according to the entities specified in the UML diagram with the corresponding relations. For this specific application there are a couple of endpoints exposed for requests:

- GET /appointments - this returns a JSON array containing test appointments;
- GET /hospital - this returns information about the hospital such as address, contact details and venues;
- GET /doctors - this fetches all the doctors present in the hospital records;
- GET /departments - this returns all the departments available in the hospital and its venues;

Faker

Instead of using ad random numerical combinations or lorem ipsum texts, a library called Faker is used to generate different random values such are names, addresses, e-mail addresses and phone numbers.

Faker is available for almost every general purpose language and is easy to use. It is always easier to work with representative data than it is to work with 'lorem ipsum' or '123456789' [3].

4.3.2 IBM BlueMix API

4.4 Testing Application Programming Interface

4.5 Tools and frameworks used

4.6 MapWize

MapWize is a service that digitalizes architectural plans and makes them interactive. MapWize offers an online environment in which you can easily create floor plans that can be used by the SDK. In the online editor you can declare specific points of interest (PoIs) and routes from and to points. The creation of a digital map for testing purposes is out of scope for this thesis.

4.6.1 MapWize SDK

The team of developers at MapWize developed a completely open source SDK, targeting the following platforms: iOS, Android, JS and () [**MapWize.io2019a**]. The one for Android has three versions: ready to use UI, bare-bones and an embedded WebView component.

MapWize Barebone versus MapWize UI

The bare-bones version of the SDK comes as a plugin on top of the MapBox OpenGL library for Android. The Mapbox library handles the embedding of interactive vector assets into mobile applications [**Mabox2019**].

4.7 Cisco Connectected Mobile Experiences integration

The manner in which the position of a patient is retrieved is based on the nearest WiFi router of Cisco.

4.7.1 Function of Cisco CMX

4.8 IndoorLocation Framework

The IndoorLocation framework is one heavily used in conjunction with MapWize, it is a framework that allows developer to use geolocalization based on numerous indoor positioning technologies (IPS) such are: GPS, beacons, Wi-Fi, Li-Fi, Ultrasounds etc [9].

Chapter 5

Conclusion

Appendix A

Development Guidelines

Appendix B

Development Guidelines

Appendix C

Development Guidelines

Bibliography

[1] Android Developer. *Transformations — Android Developers*. 2019. URL: https://developer.android.com/reference/android/arch/lifecycle/Transformations (visited on 03/06/2019).

[2] Bhavya Karia. *A quick intro to Dependency Injection: what it is, and when to use it*. 2018. URL: https://medium.freecodecamp.org/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f (visited on 03/09/2019).

[3] Daniele Faraglia. *Welcome to Faker's documentation! — Faker 1.0.2 documentation*. 2014. URL: https://faker.readthedocs.io/en/master/ (visited on 03/08/2019).

[4] Eslam Hussein. *Dominate Remote/local data with (RX+Retrofit+ROOM+MVP)*. 2018. URL: https://medium.com/@eslam.hussein/dominate-remote-local-data-with-rx-retrofit-room-mvp-f2b13a0ac27b (visited on 03/17/2019).

[5] Express. *Express - Node.js web application framework*. 2019. URL: https://expressjs.com/ (visited on 03/08/2019).

[6] Fernando Sproviero. *Android Unit Testing with Mockito — raywenderlich.com*. 2018. URL: https://www.raywenderlich.com/195-android-unit-testing-with-mockito (visited on 03/17/2019).

[7] Google. *(3) Test-Driven Development on Android with the Android Testing Support Library (Google I/O '17) - YouTube*. 2017. URL: https://www.youtube.com/watch?v=pK7W5npkhho%7B%5C&%7Dstart=111.

[8] IBM. *Med App Project Specification*. 2018.

[9] IndoorLocation.io. *Indoor Location - The open-source framework for indoor mapping.* 2019. URL: https://www.indoorlocation.io/ (visited on 03/09/2019).

[10] James Halpern. *Android – SavedInstanceState Bundle FAQ*. 2012. URL: https://content.pivotal.io/blog/android-savedinstancestate-bundle-faq (visited on 03/06/2019).

[11] MapWize.io. *Indoor mapping & Wayfinding for Smart Buildings*. 2019. URL: https://www.mapwize.io/ (visited on 03/09/2019).

[12] Roboelectric. *Robolectric*. 2019. URL: http://robolectric.org/ (visited on 03/17/2019).

[13] Mark Seemann. *Dependency Injection in .NET*. 2011, p. 16.

[14] Theo Jungeblut. *Clean Code II - Dependency Injection*. 2015. URL: https://www.slideshare.net/theojungeblut/clean-code-part-ii-dependency-injection (visited on 03/17/2019).

[15] TutorialsPoint. *SQLite Tutorial*. 2019. URL: https://www.tutorialspoint.com/sqlite/ (visited on 03/06/2019).

[16] Unknown. *14.1A: Room, LiveData, ViewModel · Advanced Android Development Course- Practicals*. 2018. URL: https://google-developer-training.gitbooks.io/android-developer-advanced-course-practicals/content/unit-6-working-with-architecture-components/lesson-14-room,-livedata,-viewmodel/14-1-a-room-livedata-viewmodel/14-1-a-room-livedata-viewmodel.html (visited on 03/17/2019).