

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

Thankssssss

Abstract

Contents

1	Summary	7
2	Project Specification	8
2.1	Project Description	8
2.1.1	Technical Design Specs	8
2.1.2	Features	8
2.1.3	Technologies to research	9
2.2	Development Guidelines	9
3	Technical Study	10
3.1	Integrated Development Environment	10
3.2	Ray Wenderlich Introductory Courses	10
3.3	General Data Protection Regulation	10
3.4	Android Architecture	10
3.4.1	Testing	10
3.4.2	Separation of concern: Dependency Injection	12
3.4.3	Lifecycle Events	13
3.4.4	Offline storage and persisting data	14
3.4.5	Repository Design Pattern	14
3.4.6	Model - View - ViewModel Architecture	14
4	Proof of Concept	17
4.1	User Interface	17
4.2	UI Patterns	19
4.2.1	Displaying Lists: RecyclerView	19
4.3	Dependency Injection using Dagger2	19
4.4	Database Communication	19
4.4.1	Implementing Repository Pattern	19
4.5	Testing Application Programming Interface	20
4.5.1	IBM BlueMix API	22
4.6	MapWize	22
4.6.1	MapWize SDK	22
4.7	Cisco Connected Mobile Experiences integration	23
4.7.1	Function of Cisco CMX	23
4.8	IndoorLocation Framework	23
4.9	Theoretical Features	23

4.9.1	Mobility Indicator	23
4.9.2	Safety Cultivation	23
4.9.3	Efficient Path Mapping	23
5	Conclusion	24
A	Development Guidelines	25

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 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. 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 [17] and the IndoorLocation framework [13], 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 [12]:

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 IDE supported by Google and based on the IntelliJ IDE of JetBrains, a company that develops an IDE for the most popular general purpose programming languages [2].

3.2 Ray Wenderlich Introductory Courses

3.3 General Data Protection Regulation

The General Data Protection Regulation of the European Union (EU) is a measure against possible theft of personal data and the protection of privacy, incorporated by the EU and its member states in 2016. The general changes regarding the privacy directive of the EU - firstly declared in 1995 - are as follows [24] [7]:

- Test

The application that is developed needs to take this specific regulation into account when it handles the personal data of a patient and the employees of the hospital.

3.4 Android Architecture

3.4.1 Testing

Testing Workflow

70-20-10 rule Red-green bar mantra is used by Google as the main testing workflow. First a failing test is written, then the code to pass the test is developed. After this process needed refactors are applied to the codebase.

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)

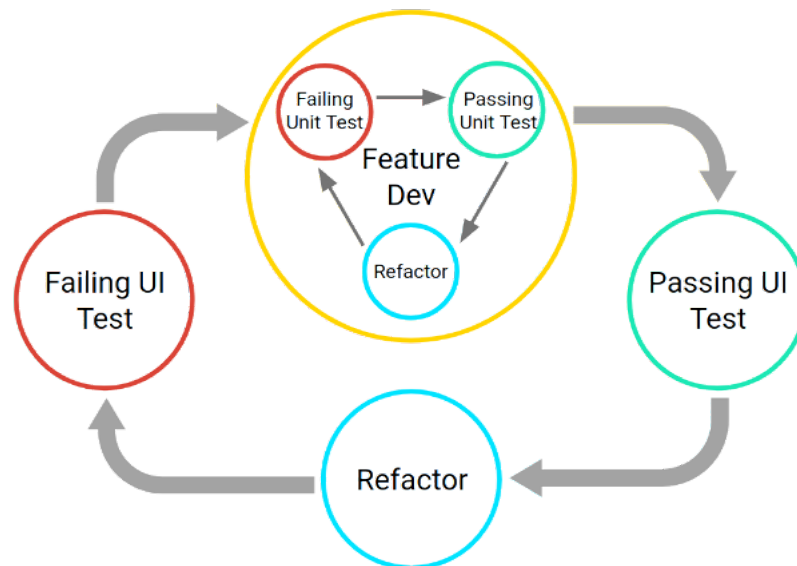


Figure 3.1: Testing workflow [10]

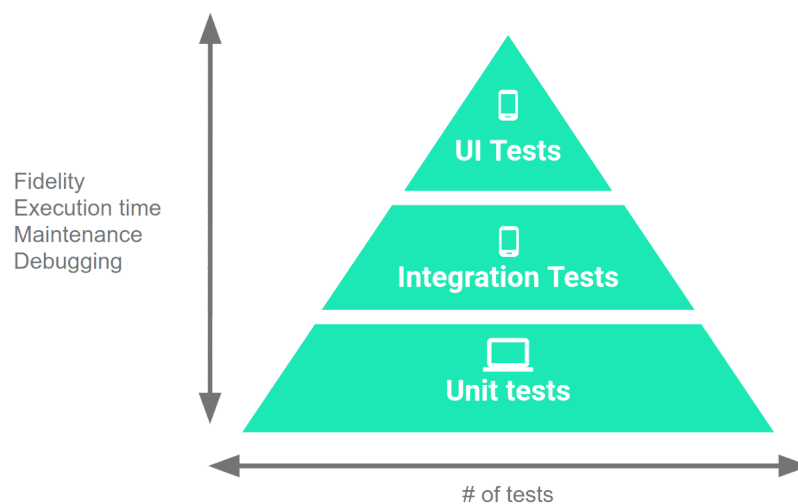


Figure 3.2: Testing pyramid [9]

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) [9]. The characteristics of a good unit test are [10]:

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 [22]. 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

3.4.2 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 [4].

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 [23]:

- 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 [25].

3.4.3 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;

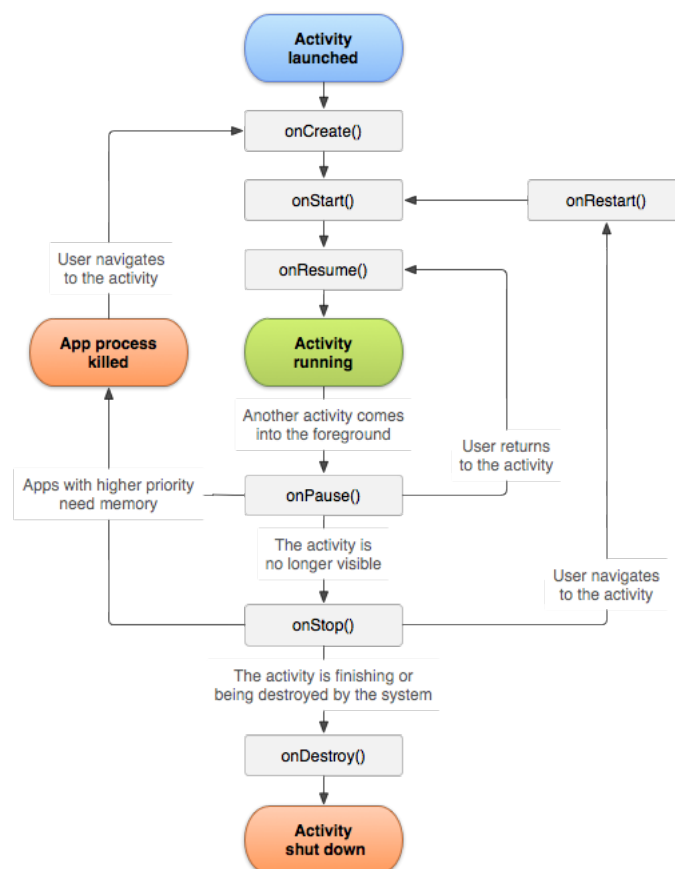


Figure 3.3: Android Activity Lifecycle Schematic [1]

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 [15].

3.4.4 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 [26].

3.4.5 Repository Design Pattern

A common pattern to use for handling database communication the repository pattern. This pattern acts as an abstraction layer on top of the data layer and the business layer and centralises the domain models. One of the common ways of integrating the pattern is creating a database access object (DAO) that interacts with the database of the application and a repository that can pass data from and to the DAO. Using the pattern as an additional abstraction layer enables easier testing and enforces the single responsibility principle [20] [19].

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:

3.4.6 Model - View - ViewModel Architecture

One of the downsides of the default architecture of an Android application is the interaction between a view, model and layout files. Normally an activity is created with corresponding models which data needs to be presented in the view. This results in , possibly, a very large activity or fragment class. A good practice is to implement a Model-View-ViewModel architecture, which focuses on separating the different components required to generate a view, thus following the separation of concern principle. This principle is one that every developer should take into account, because it is the most important one. The prime advantage of using this principle is that is easier to maintain and opts for more testable components of your application [3].

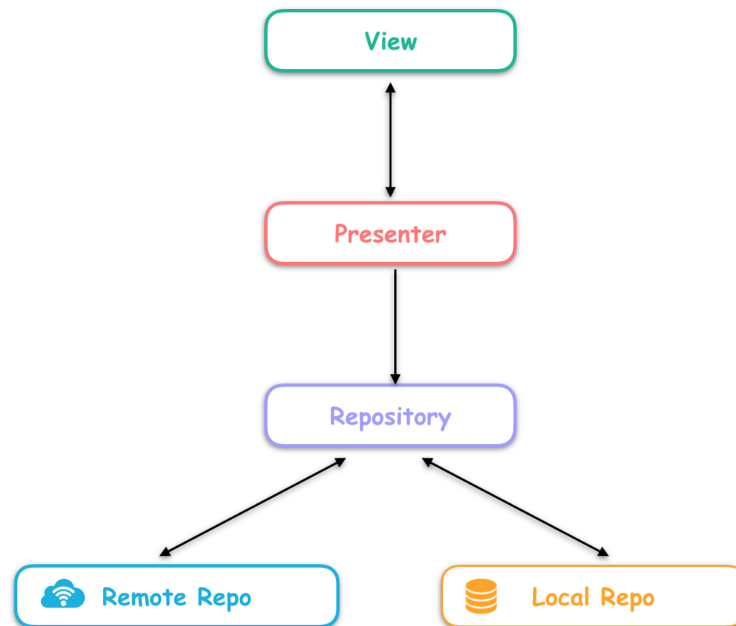


Figure 3.4: Repository Pattern inside an android application [6]

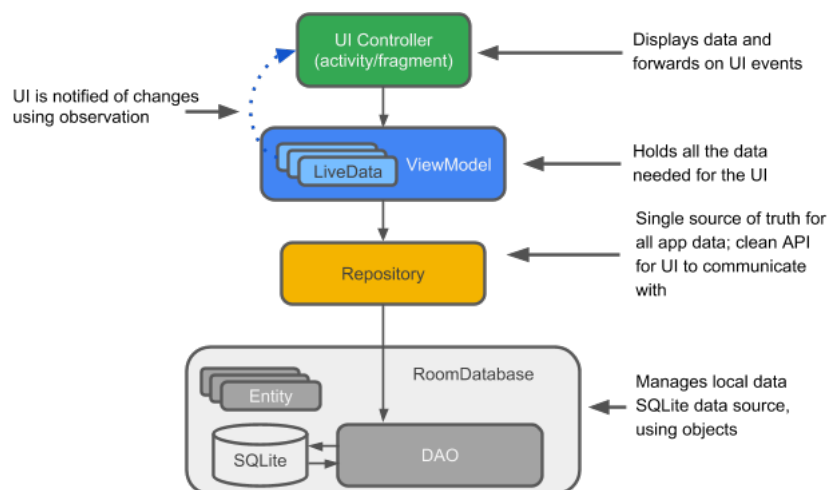


Figure 3.5: Local repository usage inside the app [11]

UI Driven

The optimal way of building an application is making the UI dependent of a model, a class holding the data for a corresponding entity, preferably persistent. A model acts independently of an activity's or fragment's lifecycle and consequently independently of the concerns regarding handling lifecycles.

Interaction between ViewModel - View

An easy way of incorporating the viewmodel is via the `ViewModelProvider` class, which returns a reference to the `ViewModel` class specified. This viewmodel contains data that will drive the UI and methods that will handle business logic. Typically the `ViewModel` has a `LiveData` list of the type of model that needs to be used and some references to a local and remote repository to fetch the data in an asynchronous fashion. The difficulty is to create a reactive UI, that handles any changes to the data in the `ViewModel` object [14].

LiveData and observables

LiveData is a datatype that is lifecycle aware, meaning it will can monitor changes and notifies the UI when that happens. The benefit of this datatype is the absence of any additional and rigid logic to handle the data binding. Methods returning LiveData can be observed using the observable pattern, the method is called from the ViewModel object and observed for changes in the LiveData that is returned by the method. This enables data-binding by only applying minor changes to the code inside the ViewModel and View [3].

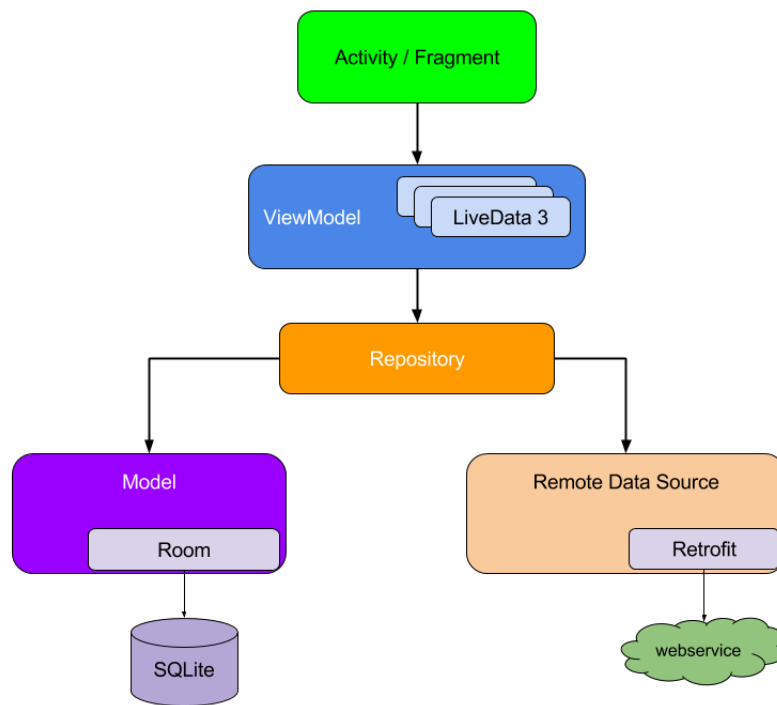


Figure 3.6: Complete architecture of application using MVVM [3]

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.

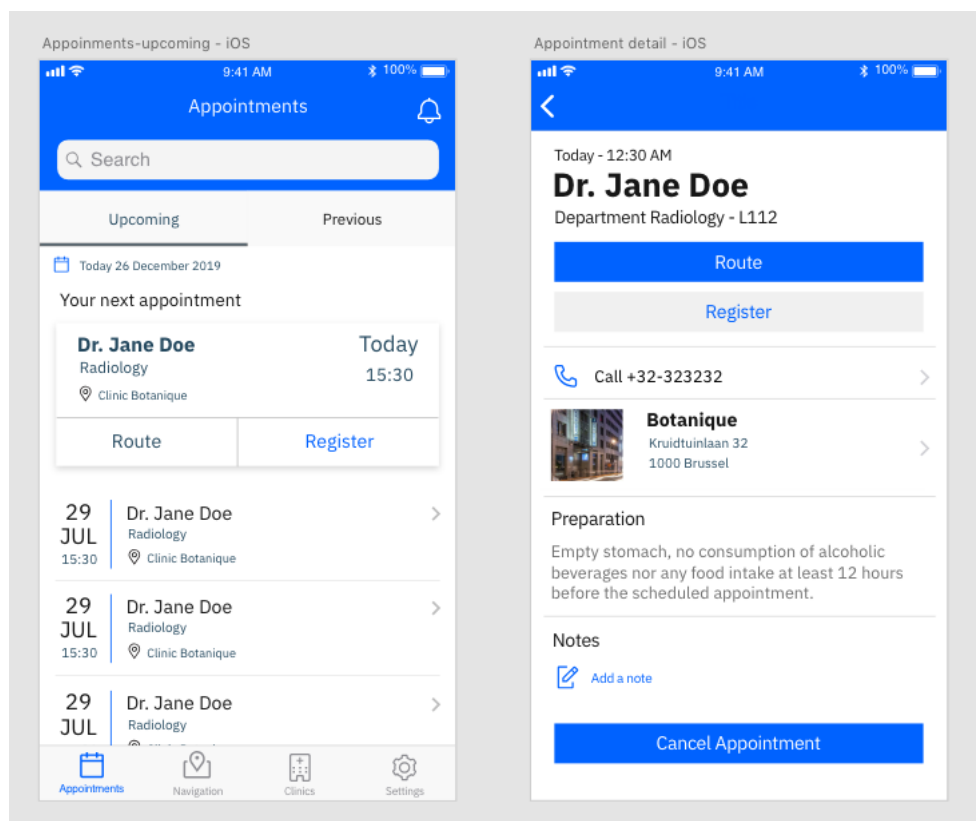


Figure 4.1: User interface of the appointments and detailed view for iOS

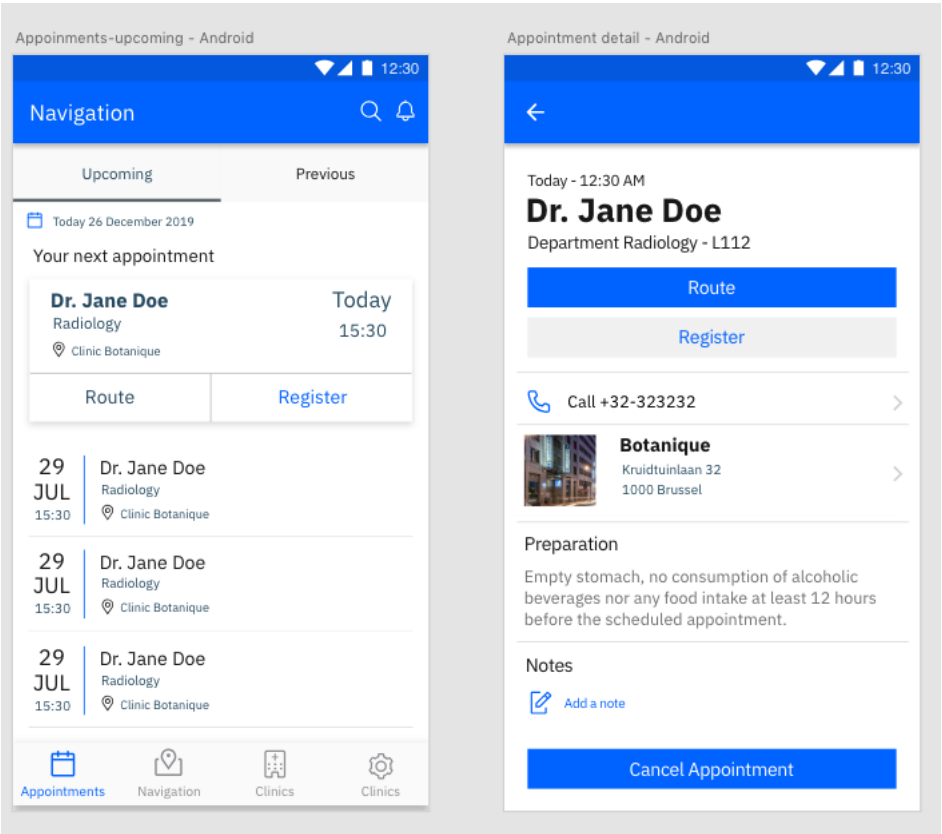


Figure 4.2: User interface of the appointments and detailed view for Android

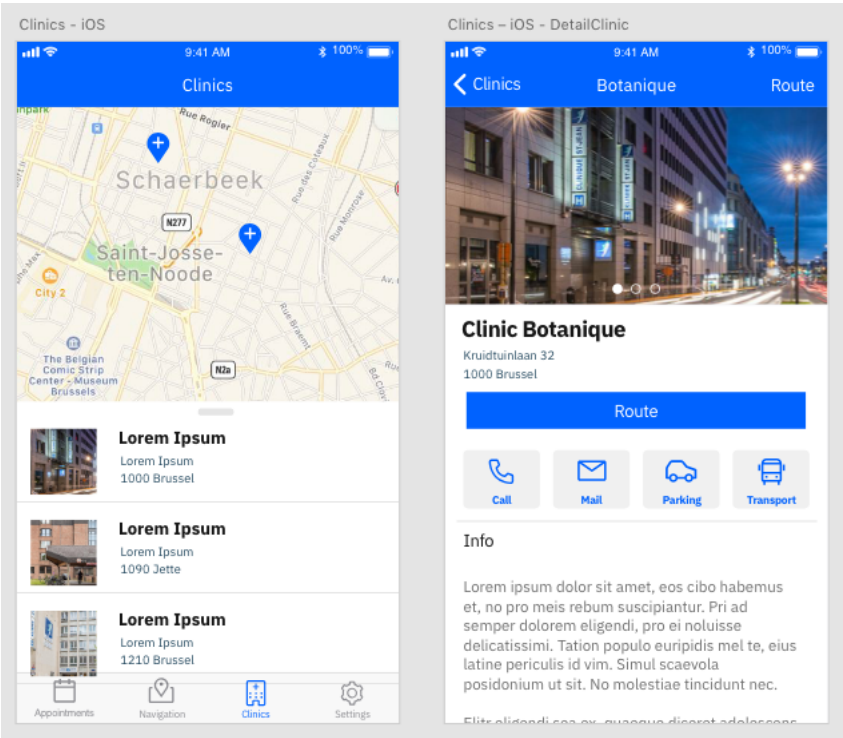


Figure 4.3: User interface of the hospital venues and detailed view for iOS

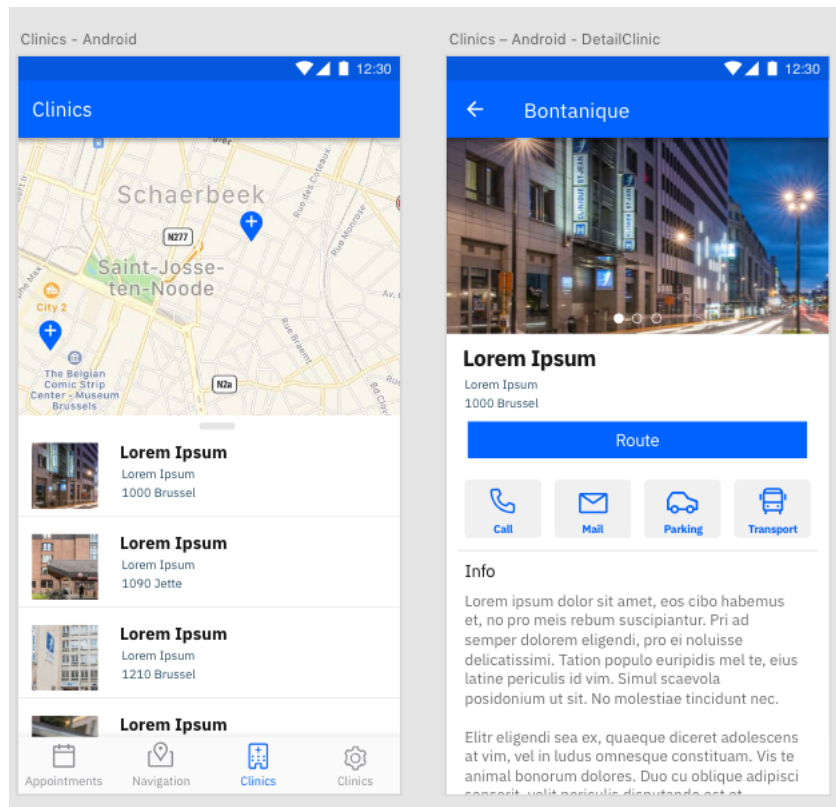


Figure 4.4: User interface of the hospital venues and detailed view for Android

4.2 UI Patterns

4.2.1 Displaying Lists: RecyclerView

[<https://developer.android.com/guide/topics/ui/layout/recyclerview>]

4.3 Dependency Injection using Dagger2

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 using the ViewModelFactory.

4.4 Database Communication

4.4.1 Implementing Repository Pattern

In practice, implementing the repository pattern in Android comes with some difficulties. Firstly dependency injection needs to be handled in a modular fashion, to attain the testable individual modules, the handling of dependency injection is covered in the next topic. Secondly, one must provide two types of data communications: online and offline.

Local Repository

The local repository contains a DAO that maps the possible database interactions. This repository fetches the Appointment objects from the local SQLite database using the RoomDB wrapper (and a static instance referring to the AppDatabase). The result is wrapped in a LiveData type to enable the functioning of observables in the activity or fragment. This local repository is also a field of the global ViewModel object, specific to the Appointment model.

Remote Repository

The remote repository handles all API communication between the application and the IBM BlueMix server. A tool used to create HTTP requests is Retrofit. When using retrofit, the only specification required is an interface containing mapping to the different possible endpoints of the requests [21]. Current WebService interface can be found below:

```
/**
 * Interface to declare endpoints of API
 */
interface WebService {
    /**
     * @GET declares an HTTP GET request
     * @Path("user") annotation on the userId parameter marks it as a
     * replacement for the {user} placeholder in the @GET path
     */
    @GET("appointments/userAppointments")
    fun getAppointments(@Header("Authorization") _authKey: String )
    : Call<List<Appointment>>
}
```

An important factor in performing this request is specifying a Base Authentication header in each request that needs to be executed.

4.5 Testing Application Programming Interface

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 [8]. 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;

An example of the JSON output can be found below:

```
[
  {
    "dateTime": "2019-03-22T22:02:37.840Z",
    "preparation": "Fuga enim non est quo",
    "doctor": {
      "firstName": "Brenda",
      "lastName": "Quitzon",
      "departments": [
        {
          "name": "Surgery",
          "venue": {
            "name": "Parisian, Parker and Waelchi",
            "additionalInformation": {
              "website": "https://benny.net",
              "phone": "1-346-029-0603",
              "parking": "https://jonas.com",
              "email": "Rossie_Paucek60@yahoo.com"
            },
            },
          "address": {
            "street": "Grimes Mission",
            "postalCode": "50003-1879",
            "city": "Stephanyton",
            "country": "Malta",
            "streetNo": 4007
          }
        }
      ]
    }
  }
]
```

Entities

The specific entities used throughout this application are:

- appointment;
- hospital;
- sites - the different locations of a hospital;
- doctor;
- patient;
- department;

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' [5].

4.5.1 IBM BlueMix 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": "0000101300000001",
    "appointmentTime": "2017-10-16T14:14:00+02:00",
    "patientNr": 2000000420,
    "patientName": "SMITH, JULIE",
    "doctorNr": 123456,
    "doctorName": "FELDUNS, JEAN",
    "departmentId": "I",
    "departmentName": "USI",
    "siteId": "B0",
    "siteName": "Botanique",
    "appointmentReason": "Consultation Orthopédique",
    "appointmentInstruction": "La pendule fait",
    "appointmentStatus": "E",
    "appointmentStatusDescription": "Evaluated"
  }
]
```

Notice that the API response is not yet normalized as it is in the testing API in Express. The testing API is only used in the first steps of the application, afterwards the data from the BlueMix server is 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 progressive web application (PWA) [18]. 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 [16], this library is out of scope for this thesis, it is sufficient to know its function inside the MapWize SDK.

4.7 Cisco Connected 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 [13].

4.9 Theoretical Features

4.9.1 Mobility Indicator

4.9.2 Safety Cultivation

4.9.3 Efficient Path Mapping

Chapter 5

Conclusion

Appendix A

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] Android Developers. *Download Android Studio and SDK tools*. 2019. URL: <https://developer.android.com/studio> (visited on 03/23/2019).
- [3] Android Developers. *Guide to app architecture — Android Developers*. 2019. URL: <https://developer.android.com/jetpack/docs/guide> (visited on 03/08/2019).
- [4] 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).
- [5] 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).
- [6] 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).
- [7] EU GDPR Portal: by Trunomi. *Key Changes with the General Data Protection Regulation – EUGDPR*. 2019. URL: <https://eugdpr.org/the-regulation/> (visited on 03/23/2019).
- [8] Express. *Express - Node.js web application framework*. 2019. URL: <https://expressjs.com/> (visited on 03/08/2019).
- [9] 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).
- [10] 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=pK7W5npkhh0%7B%5C%7Dstart=111>.
- [11] GoogleDeveloperTraining. *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).
- [12] IBM. *Med App Project Specification*. 2018.

- [13] IndoorLocation.io. *Indoor Location - The open-source framework for indoor mapping*. 2019. URL: <https://www.indoorlocation.io/> (visited on 03/09/2019).
- [14] Jaewoong Eum. *Android MVVM Architecture Components using The Movie Database API*. 2018. URL: <https://medium.com/@skydoves/android-mvvm-architecture-components-using-the-movie-database-api-8fbab128d7> (visited on 03/06/2019).
- [15] James Halpern. *Android – SavedInstanceState Bundle FAQ*. 2012. URL: <https://content.pivotal.io/blog/android-savedinstancetype-bundle-faq> (visited on 03/06/2019).
- [16] MapBox. *Mobile — Mapbox*. 2019. URL: <https://www.mapbox.com/mobile/> (visited on 03/23/2019).
- [17] MapWize.io. *Indoor mapping & Wayfinding for Smart Buildings*. 2019. URL: <https://www.mapwize.io/> (visited on 03/09/2019).
- [18] MapWize.io. *Indoor mapping for developers*. 2019. URL: <https://www.mapwize.io/developers> (visited on 03/09/2019).
- [19] Nahidul Hasan. *SOLID Principles-simple and easy explanation – Hacker Noon*. 2018. URL: <https://hackernoon.com/solid-principles-simple-and-easy-explanation-f57d86c47a7f> (visited on 03/23/2019).
- [20] Per-Erik Bergman. *Repository Design Pattern – Per-Erik Bergman – Medium*. 2017. URL: <https://medium.com/@pererikbergman/repository-design-pattern-e28c0f3e4a30> (visited on 03/23/2019).
- [21] Retrofit. *Retrofit*. URL: <https://square.github.io/retrofit/> (visited on 03/24/2019).
- [22] Roboelectric. *Roboelectric*. 2019. URL: <http://roboelectric.org/> (visited on 03/17/2019).
- [23] Mark Seemann. *Dependency Injection in .NET*. 2011, p. 16.
- [24] THE EUROPEAN PARLIAMENT AND THE COUNCIL OF and THE EUROPEAN UNION. *DIRECTIVE 95/46/EC OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL*. Tech. rep. European Parliament and of the Council, 1995. URL: <https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:31995L0046%7B%5C%7Dfrom=en>.
- [25] 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).
- [26] TutorialsPoint. *SQLite Tutorial*. 2019. URL: <https://www.tutorialspoint.com/sqlite/> (visited on 03/06/2019).