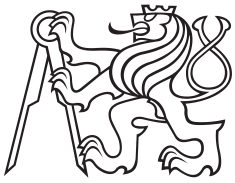


Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Mobile app for collecting data about objects placed on the city pavements

Jan Kraus

Supervisor: Ing. Ivo Malý, Ph.D.
Field of study: Software Engineering and Technology
May 2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kraus** Jméno: **Jan** Osobní číslo: **434202**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Mobilní aplikace pro sběr dat z chodníkové sítě

Název bakalářské práce anglicky:

Mobile app for collecting data about objects placed on the city pavements

Pokyny pro vypracování:

Analyzujte požadavky pro sběr a ověřování chodníkových dat, které byly definovány v rámci projektu Cityplan, tj. vyžádané ověření stavu konkrétních objektů a nahlášení problému na chodníkové síti. Dále analyzujte požadavky na aplikace sbírající data pomocí davu (crowdsourcing). Na základě analýzy vytvořte návrh struktury mobilní klientské aplikace, která umožní davový sběr dat dle definovaných scénářů. Tato klientská aplikace bude využívat také serverovou část (resp. její programové rozhraní), která je vytvářena v jiné práci. Dále vytvořte vysokoúrovňový prototyp (HiFi prototyp) mobilní aplikace na platformě Flutter. Funkčnost aplikace ověřte pomocí softwarových testů. Dále proveďte vyhodnocení aplikace pomocí kvalitativních uživatelských testů s alespoň 5 uživateli nad alespoň 3 typy chodníkových objektů.

Seznam doporučené literatury:

[1] Riganova, M., Balata, J. and Mikovec, Z., 2017, September. Crowdsourcing of Accessibility Attributes on Sidewalk-Based Geodatabase. In IFIP Conference on Human-Computer Interaction (pp. 436-440). Springer, Cham.
[2] T. Lowdermilk, User-Centered Design, O'Reilly Media, 2013.
[3] B. Fling, Mobile Design and Development, O'Reilly Media, 2009
[4] F. ZAMMETTI. Practical Flutter: Improve your Mobile Development with Google's Latest Open-Source SDK. Apress, 2019.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Ivo Malý, Ph.D., katedra počítačové grafiky a interakce FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **11.02.2020** Termín odevzdání bakalářské práce: **22.05.2020**

Platnost zadání bakalářské práce: **30.09.2021**

Ing. Ivo Malý, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Acknowledgements

I would like to express my great appreciation for Ing. Ivo Malý, Ph.D., my supervisor, for his patient guidance, encouragement and useful critique. I would also like to extend my thanks to my family and my friends, that accompanied me on my academic journey.

Declaration

I hereby declare that I have written this bachelor thesis independently and quoted all the sources of information in accordance with Methodical instructions about ethical principles for writing academic theses.

In Prague, 21. May 2020

Abstract

The motivation for this thesis was the creation of a cross-platform application to provide data for better navigation of visually and movement impaired people with the help of crowdsourcing. In the beginning, core principles and problems are defined concerning geographical data crowdsourcing, followed by a system design analysis. Later, technologies used in Flutter framework are discussed and analyzed with Redux and Firebase being the chosen approaches. The mobile application is then implemented using provided designs. In the end, the application is verified with software tests and user-testing.

Keywords: flutter, mobile development, crowdsourcing, geodata, redux, firebase

Supervisor: Ing. Ivo Malý, Ph.D.

Abstrakt

Motivace této bakalářské práce je vytvoření mobilní aplikace pro všechny platformy s cílem poskytnout lepší data pro navigaci pohybově a zrakově postižených lidí za pomoci široké veřejnosti. Nejdřív jsou definovány základní pojmy a principy týkající se sběru dat veřejností a následně jsou analyzovány požadavky pro danou aplikaci. Následně jsou analyzovány technologie pro vývoj aplikací za použití nástroje Flutter, kde Redux a Firebase jsou zvoleny hlavními nástroji. Poté je mobilní aplikace implementována na základě dostupného grafického návrhu. Nakonec byla funkčnost aplikace ověřena softwarovými a uživatelskými testy.

Klíčová slova: flutter, vývoj mobilních aplikací, crowdsourcing, geodata, redux, firebase

Contents

Figures

Tables

Chapter 1

Introduction

Navigation is an essential asset to people to reach new destinations and explore unknown places. Mapping systems have a vast amount of information about roads, but not so much about pavements, which makes the navigation car-centred. However, for people with impaired mobility and vision, the most common navigation systems fail to convey important information about pavement obstacles and accessibility attributes for existing landmarks and pedestrian segments.

To address this problem, Czech Technical University works closely with the Central European Data Agency and T-MAPY as a part of a grant provided by the Technology Agency of the Czech Republic. The project is called CityPlan and has an objective to develop a sidewalk-based Geographical Information system (GIS) with features outlining the pedestrian segments and their attributes.

Pedestrian segments are drawn into GIS by experts using utilities like satellite imagery. When they finish this step, attributes are later collected by professional on-site exploration. By using professionals, data collection is costly and time-demanding. The aim is to reduce the cost and time needed for this exploration by developing a mobile application for the collection of pavement data by non-experts, the crowd. Professionals later aggregate this data for the specialized navigation systems used by mobility and vision impaired individuals.

My thesis follows up on the work of Ms Riganová [11], that constructed the basic concept of the application, including its gamification and educational system for the users. The main focus of my thesis is to analyze and create a proof of concept for both Android and iOS using Flutter, that centres its attention mainly on the data collection part. User education and gamification aspects of the application are not a part of the implementation.

Chapter 2

Problem description

This chapter describes the subject matter around crowdsourcing as a way of collecting geographical data. It also touches on topics like motivation of people to gather data or related works done in the field. We will also define elements on the pavement networks that are cardinal for navigation of people with visual or movement impairment.

2.1 Crowdsourcing geological data

In this section we centre our attention to geological crowdsourcing as an alternative to professional data collection, examining some real-life examples, underline possible shortcomings related to amateur data collection and look for possible solutions to promote the interest of people for geological crowdsourcing.

2.1.1 Definition

Crowdsourcing provides us with a way to mitigate the time- and cost-consuming process of collecting professional data onto the crowd. Primarily geographical crowdsourcing specializes in accumulating spatial data. We swap experts for ordinary people (crowd) to provide annotations for geographical features. The original term crowdsourcing was made up by Jeff Howe's article [6] in the popular magazine Wired.

"Crowdsourcing represents the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call."

Crowdsourcing differs from its sibling commons-based peer production (which is a term coined by Yochai Benkler in [1]), where people cooperate voluntarily to reach a shared goal. There is no top-down directive of what needs to be done (like in crowdsourcing), and all the work is generated and governed by the participants, and the locus of control resides in the hands of

the community.

2.1.2 Consideration

Technology improved the possibilities for collecting and sharing geographical data that almost anybody can be a part of it. Combining technology and crowdsourcing, we can assess a significant volume of data in a short time with minimal investments (time or cost) and make modifications to them quickly. But with great power comes great responsibility, and crowdsourcing has its disadvantages. We need to consider these things:

- Amateur contributions are generally of lower quality than those of experts.
- Amateur contributors use worse equipment, which is less precise than professional equipment.
- Motivation differs between participants; some may even abuse the system and put incorrect data on purpose.

Even though these can be valid concerns, a case study by [12] showed, there is little to no difference between data collected by amateurs and those collected by experts. This study also tracked the confidence level for each identification and unsurprisingly, the more confident in identification the amateurs were, the more consistent and correct their findings were. As much as experts are invaluable in collecting data, crowdsourcing is a valid and powerful tool to collect data.

2.1.3 Types of inducement

Crowdsourcing systems highly depend on sustained interest and participation of individuals, which relies on their motives. These may vary on situational context or participant. Based on this paper [7] inducement of users can be divided into several categories (shown in Figure 2.1) with real-life examples and applications.

- Reputation systems
  - platform's users rate each other based on their conduct
  - the system combines ratings to form an assessment of the reputation
  - reputation is measured in several ways, usually represented by a numeric value
  - blocking functionality for users with low/lousy reputation
- Social incentive mechanisms

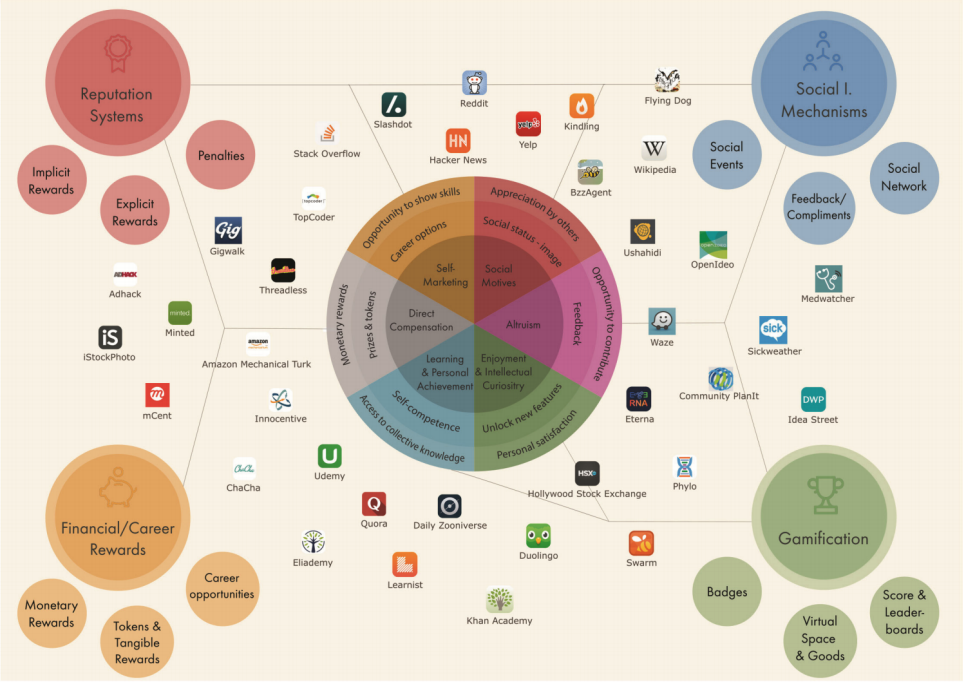


Figure 2.1: User motives, incentives and incentive mechanisms [7]

- uses the users need for a good social images and wanting to be perceived as smart or wealthy
- introduces mechanics that act as enablers of social interactions and giving the ability to showcase their skill

- Financial/Career rewards
  - compensation for the lack of enjoyable tasks or social rewards
  - the compensation varies from a chance to win a prize, monetary compensation or free service/product

- Gamification
  - using game design elements in a non-game context
  - they usually include badges, achievements, leader-boards or include virtual goods or gifts
  - users are rewarded for in-app activity, which incentives their behaviour

2.1.4 Related work

The leader in collecting geographical data by using crowdsourcing is probably OpenStreetMap[1] project, which was first introduced in 2004 with a common

[1] https://www.openstreetmap.org/

goal to generate a free and editable alternative with the help of volunteers to the traditional GIS systems and data providers. Even though it surpassed the traditional suppliers in having more detailed, accurate and up-to-date data, it still does not take special needs of people with disabilities into account.

There are successful projects on the market that either use the dataset that OpenStreetMap provides or are build on other available data like Google Street view. We can take a quick look at two of them and see, how they handle the data collection.

- WheelMap[2]
  - OpenStreetMap based solution for finding wheelchair accessible places
  - Anyone can mark public places based on their wheelchair accesibility
  - They use simple traffic light based system
    - Green - fully accessible by wheelchair
    - Yellow - partly accessible by wheelchair
    - Red - not wheelchair accessible
  - It has a web app and mobile applications for both iOS and Android
  - Multilanguage support
- Sidewalk[3]
  - Research project from the University of Washington
  - Using Google Street View[4] as a tool for sourcing data from users
  - Provides interactive guide for first-time users
  - User can either gather new data or approve already collected data
  - Training a neural network for future automation of the labeling task (users will then be used just for checking the validity)
  - Is only available in few big cities in the USA

2.2 Geographical data definition

This section lays out the types of geographical objects and data needed for a navigation system of disabled people. Based on on project documentation [2], we could classify them into three separate categories, that help us understand the different properties we have to store and each come with their needs.

[2] https://wheelmap.org/
[3] https://sidewalk-sea.cs.washington.edu/
[4] https://www.google.com/streetview/

2.2.1 Obstacles

Obstacles are one of the most common entities for which information needs to be collected. Without spatial information about impediments, it is really hard for navigation systems to redirect users through another route. This can lead to aggravating situations where a normal walk to work is made into an obstacle course for impaired people. To avoid this, we will invite users to collect the following data for obstacle objects.

- Geographical location
- Type - bin, lamp post, tree, road work, etc...
- Position on the sidewalk - to the left, to the right, right in the middle
- Throughput of the sidewalk - how much space is there to go through
- Dimensions - length and width (in various units)
- Image - photo of the object with context to its surrounding
- Additional information - any information not covered by previous points

2.2.2 Orientation points

Another type of objects we need information about are orientation objects. These include any pavement structure that is important for safe passage around pavements and through traffic. A tutorial about its parameters is recommended to collect data about these, because a casual user, the ones that have no beforehand experience with city accessibility design, would be confused about what exactly they should be looking for. For now, we differentiate two entities - a crosswalk and a corner.

- Crosswalk
  - Marking of crosswalk side
    - Information needed for each side of crosswalk
    - Types are signal strip, warning strip or sound signal
  - Type of the surface finish
    - Information needed for each side of crosswalk
    - Types are asphalt, Prague mosaic, cobblestones, etc...
  - Number of traffic lines the crosswalk goes through
  - Series of yes or no questions
    - Is there an island?
    - Is it crossing any bike/tram lines?

- Image - visual representation of the crosswalk
- Additional information not covered by above questions

- Corner
  - Shape - sharp, round or polygonal
  - Image - visual representation of the crosswalk
  - Additional information not covered by above questions

- Sidewalk
  - Minimum width of the sidewalk
  - Type of material - asphalt, Prague mosaic, cobblestones, etc...
  - Condition - poor, average or excellent
  - Type of surroundings in a direction - buildings, greenery, misc, etc...
  - Image - visual representation of the crosswalk
  - Additional information not covered by above questions

2.2.3 Points of interest

The last group of objects are points of interest. They encompass structures that are generally helpful for impaired people to know about. It can range from public transport infrastructures like subway entries or exits, bus or tram stops, etc..., or are of general use like public toilets or benches. For now, only benches have defined data needed for collection, but more should be available in the future.

- Bench
  - Head rest presence
  - Condition - poor, average or excellent
  - Material - wooden, metal, stone, etc...
  - Photo
  - Additional information not covered by above entries

Chapter 3

System design

This chapter encompasses the process of structuring and designing the application architecture, which is simple to extend, scale and overall great to work with. It is platform/programming language independent, to give us the ability to compare technologies later and helps us make strategical decisions.

3.1 Application scope

The scope and user interface is closely based on design prototypes proposed by Ms Riganová in her master thesis [11]. She constructed the basic concept of the entire application including its gamification and educational system of the users. Both of these aspects were omitted and not part of this implementation.

The design team from the Cityplan project [2] made further improvements upon the initial design. I received access to the Figma tool the designers used to iterate over the designs and I used them as my stepping stone. I did not try to fully recreate provided design as user testing is still in progress and final design is not yet finished.

I still tried to closely recreate the user experience and the main components of the application like interactive map, step wizard for data collection or sliding panel for showing a list of objects. In the Chapter Examples of provided graphical design, you can see examples of the latest design suggested by the designers.

3.2 Application requirements

This section contains a comprehensive overview of functional and non-functional requirements to be taken into consideration for designing the application. These requirements were either defined by the design provided by the project documentation [2] or discovered during consultation with the client represented by my supervisor.

3.2.1 Functional requirements

- User management
  - The client application must allow users to create an account within the system.
  - The client application must allow users to log in, respectively log out
- Data representation
  - The client application must allow users to see objects in their immediate vicinity
  - The client application must plot the data in a map interface and show a list in a comprehensive form
- Visual recognition
  - The client application must allow users to identify different types of objects
  - The client application must differentiate the object by a name and with a visual cue
- Explore
  - The client application must allow users to explore data outside his immediate vicinity
- Report new object
  - The client application must allow users to report upon a new object with a user-friendly interface
- Survey an object
  - The client application must allow users to report upon a existing object with a user-friendly interface
  - The client application picks a scenario based on the type of the object
- Image resources
  - The client application must allow users to use their camera or a library to upload an image
- Personal dashboard
  - The client application must allow users to see an overview of their progress in the application

3.2.2 Non-functional requirements

- OS independent
  - The client application is implemented to work on both iOS and Android
- Supported OS versions
  - The client application supports Android version 4.4 or newer (96% of Android devices) and iOS version 9 and newer (more than 99% of iOS devices)
- Secure data storage
  - The client application uses a service for storing JSON-like data and multimedia data with ability to restrict access to that data
- Offline use
  - The client application needs to work temporarily without an internet connection
- Data consumption efficient
  - The client application needs to be optimize to use as little data as possible
- Testable
  - The client application needs to be easily testable

3.3 Use cases

With each project, there should be a general understanding, what actions should the user be able to take. There are many approaches of writing use-cases and how their visual representation looks like. I decided to follow the simple eight steps defined in [10] and let each use-case have clear definition of following things:

1. Clear identification - identification number and a unique name
2. Actor - who takes the actions
3. Preconditions - what state the system or the actor are in before
4. Success scenario - happy path of the use-case
5. Fail scenarios - what can fail at which step

I recognized and defined eight use-cases that are viable in the scope of this application (proof of concept). I described them in detailed using the guide above and they can be found in the Chapter Use cases. Here is a simple list with references to each use-case description.

- Use Case 1 - Login (Section A.1)
- Use Case 2 - Show objects on map (Section A.2)
- Use Case 3 - Add new report (Section A.3)
- Use Case 4 - Add report to an object (Section A.4)
- Use Case 5 - Filter objects (Section A.5)
- Use Case 6 - Survey overview (Section A.6)
- Use Case 7 - Profile overview (Section A.7)
- Use Case 8 - Logout (Section A.8)

3.4 Domain model

Before describing a domain model, we need to define it. Based on [3], domain model is "a structured visual representation of interconnected concepts or real-world objects that incorporates vocabulary, key concepts, behaviour, and relationships of all of its entities."

Domain model should closely resemble the code that is written and shares the vocabulary and should not try to capture the entire scope of the project but focus more on one part that is encapsulated by a package. This way, we can iterate the project and make it easier to modularize.

3.4.1 Enumeration package

In domain models enumerations are generally used for a property, that has a limited set of values. For example, Geometry may have a property Type that could have values Point, LineString or Polygon. One disadvantage of this approach is that enumeration should only be used for non-expending static lists.

One example from our domain is SurveyType, which does not have a final set of values but is still considered an enumeration (there will be a final set of values). Other enumeration types are described in detail in Figure 3.1.

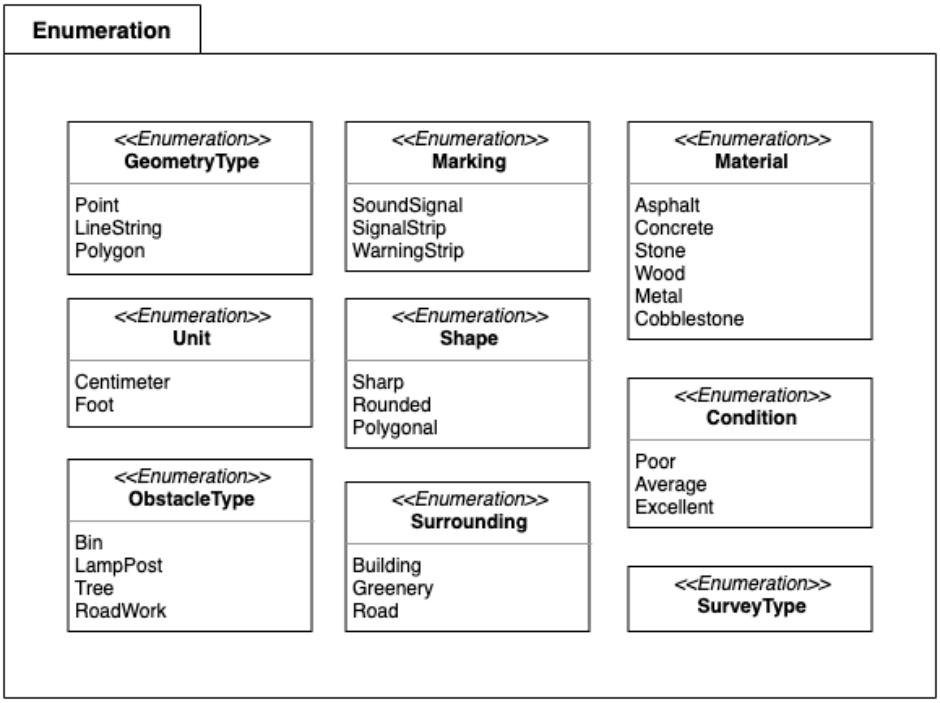


Figure 3.1: Domain model - enumeration package

3.4.2 Datatype package

A data type is an object recognizable by its value, but where we are not interested in its identity or its associations. They usually have a more complicated structure and are shared between entities. Data types used in our domain model can be seen in Figure 3.2.

3.4.3 Entity package

Entities are the first thing we need to think about when creating a domain model. We need to define what properties an entity has and how it relates to other entities. Our domain model (entity package described in Figure 3.3) has three fundamental entities:

- User - describes a person using our application, which is identified by his unique ID. User can create a Survey either based on GeoObject or a completely new one
- GeoObject - describes an object that encompasses geographical objects in the real world. Multiple Surveys can be created on one GeoObject
- Survey - it describes a report made by a User, that can belong to a GeoObject. It is extended by multiple sub-surveys (NewReport, Cor-

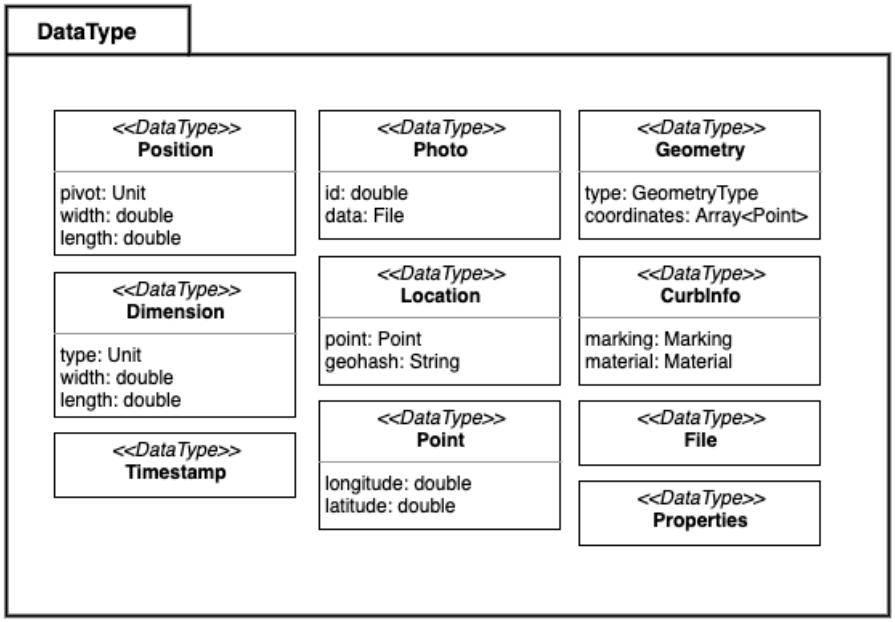


Figure 3.2: Domain model - data types package

nerReport, BenchReport and SidewalkReport) and it is probable, that more specific types can be defined later in development.

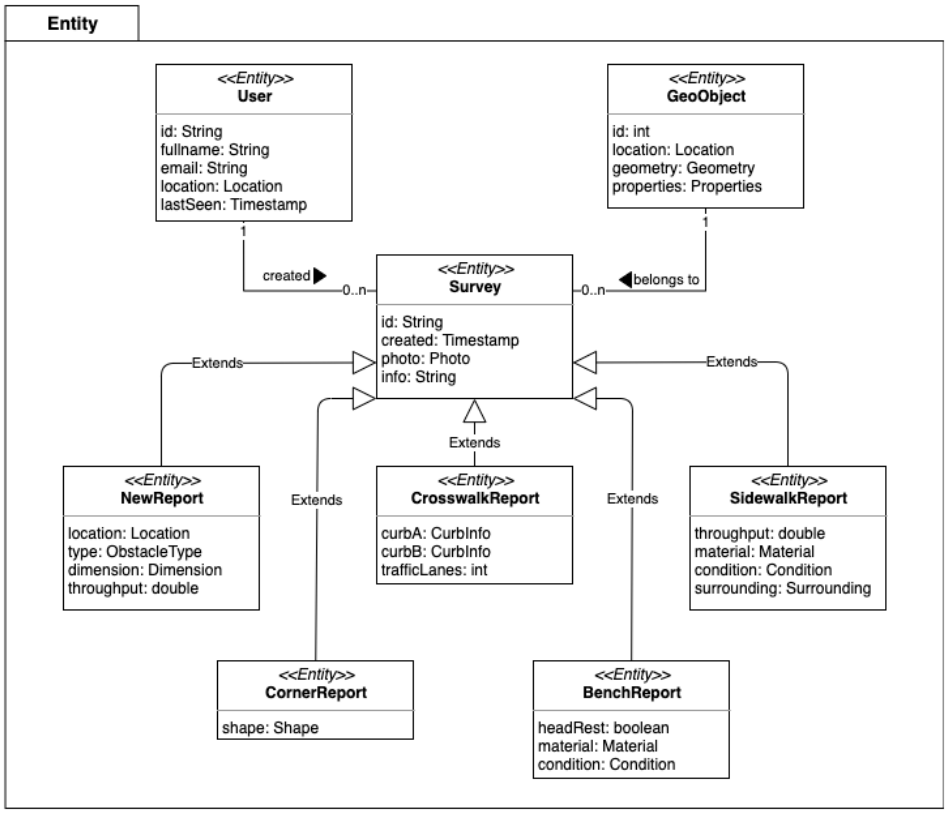


Figure 3.3: Domain model - entity package

Chapter 4

Technical analysis

At the start of a project I need to make few decisions, which steer the development in a particular direction. I introduce Flutter as our software development kit and lay out principles how I will structure the project from architecture stand point. I will also compare two state management approaches and pick one based on the analysis. Also database, storage management system and map frameworks are shortly discussed.

4.1 Software development kit

Official Flutter documentation [14] defines Flutter as a UI toolkit complete with widgets and tools for cross-platform application development. It enables users to create visually attractive, natively compiled applications with its ability to draw straight into the platforms canvas as described by Figure 4.1.

The main programming language is Dart, and it is an open-source project that was released by Google. The architecture style is based on reactive programming, following the same style as React. It is being constantly developed and new features are being introduced every minor release.

One interesting feature of Flutter is hot reload, which helps developers to easily experiment, iterate over designs or fix bugs. It works by injecting new source code into the Dart Virtual Machine, which then updates the classes with the new versions, which in turn triggers an update to the widget tree.

Another important feature of Flutter is so called tree shaking. It is a method, where function calls are presented by a tree-like structure and so functions that are never called can be eliminated. This optimizes the size of the app bundles and leads to less code the end device needs to run.

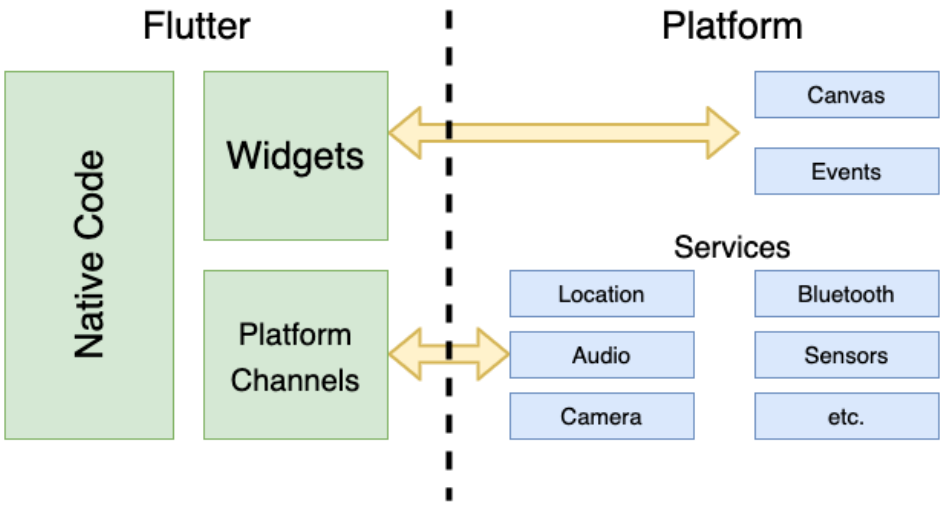


Figure 4.1: Flutter architecture

4.2 Architecture

Architecture is usually defined as an aggregation a system that is composed of components, their interaction and the principles of design and further improvement. It is crucial for setting common ground and helps make a sustainable, flexible, extensible and usable software.

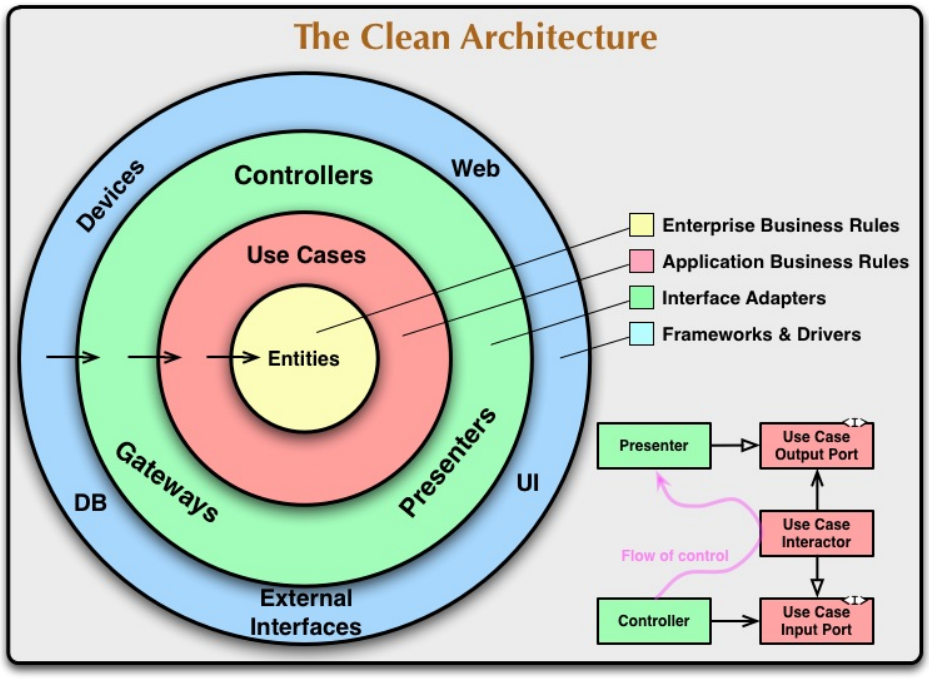


Figure 4.2: Clean Architecture by Uncle Bob (Robert C. Martin) [9]

Architectures vary in details but they all share a common objective, which is Separation of Concerns (SoC). Clean architecture tries to adhere to rules defined by [9] and Figure 4.2 is trying to integrate them into a single actionable idea.

This architecture follows the Dependency Rule, which states, that source code dependencies can only point inwards. So nothing in the inner circle knows anything about the outer layers relative to it. So, for example Entities layer shouldn't know anything about how the UI works. By definition, there are four layers:

- Entities
  - encapsulate the Enterprise business rules
  - usually an object with methods or set of data structures or functions
  - is the least likely to change when something external changes
- Use cases
  - contain application specific business rules
  - implement all use-cases of the system
  - handle changes from and to entities
- Interface adapters
  - set of adapters that convert data from the most convenient format for the use-cases and entities to external agencies like database and UI
  - convert data from an external agency like a database to a one understood by the system
- Frameworks and drivers
  - composed of frameworks and tools like the database and UI
  - glue together code that communicates inwards

In conclusion, adhering to the Dependency law and Separation of Concerns, we can save development time by having an intrinsically testable system, which is one of the requirements. As discussed in the Application scope section, design could still change, and having a robust architecture will make it easy in the future to refactor the user interface.

4.3 State management

Every app needs a way to manage and control its inner state, like user interactions, data fetching, what is currently happening etc. And when the

application grows in size and complexity, you can run into issues. Without a single source of truth, your application can easily go out of sync, and weird visual or logical bugs can occur.

There are many different solutions or architectures how to solve the issue, and the Flutter team provides an example repository [4] and overview for each one of them with description and simple implementation. Few of them were chosen for analysis, that are used by the community or are commonly used concepts in programming.

4.3.1 Framework solution

The basic state management solution that vanilla Flutter provides is called Lifting State Up and works on a simple principle. If two widgets need access to the same data, find their nearest shared parent, store the data there and pass it to the widgets mentioned above. If you need to change the state of the data from its children, you give them a callback function responsible for updating the state, and you invoke them inside the children widgets.

This approach is great for small apps and quick proof of concepts. Still, as the app grows more extensive, you get a large root widget that is hard take care of, and you need to waterfall down the callback and data through multiple layers of widgets, cluttering the overall project. It makes the code less readable and harder to debug.

4.3.2 Business Logic Components (BLoC)

In the article by Kacper Kagut [8], Business Logic Components (BLoC) is outlined as a new architecture pattern, whose essence is an event stream that handles all communication through managed Widgets. This pattern has four layers (UI, BLoC, Repository and Data Sources) as showcased in Figure 4.3.

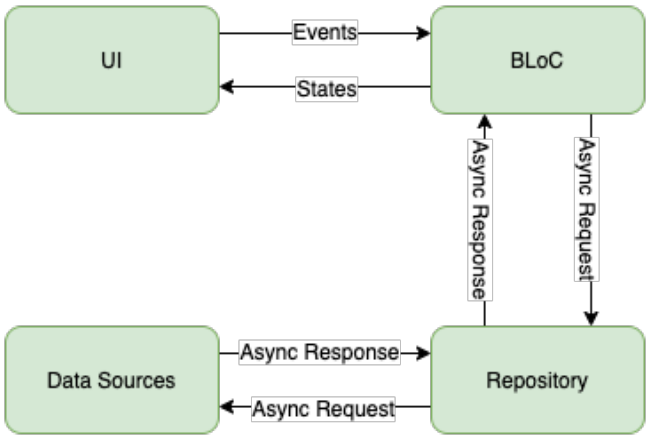


Figure 4.3: Business Logic Components architecture

Data source is the lowest layer in this architecture, which exposes simple API for making CRUD operations (`createData`, `readData`, `updateData` and `deleteData`). Its responsibility is providing raw data and should be generic and versatile, example shown in the Figure B.1. It usually provides those data from database, network calls or shared preferences.

A repository takes in raw data from data sources (there can be multiple data sources handled by one repository) and transforms them into data that can be handed over to the business logic layer, which could be implemented like in Figure B.2.

The BLoC layer has a responsibility responding to events from presentation layer with a new state. It can depend on multiple repositories to retrieve the data needed to build up the application state. Each BLoC has a state stream that other BLoCs can subscribe to, which allows them to respond to changes and can handle errors like shown in Figure B.3.

The presentation layer has a responsibility to figure out, how it should render itself based on data provided by BLoC, which is injected like in Figure B.4. It also handles user inputs, which it sends as events to the BLoC layer, which then triggers the architecture to fetch data.

4.3.3 Redux

Redux[1] is originally a Javascript library based around functional programming (taking advantage of reducing functions) and was introduced as an improvement to Facebook popular data architecture Flux[2]. It uses a standardized unidirectional data flow architecture that makes it easy to maintain and test applications.

Redux has lifecycle as outlined in Figure 4.4. Creators of Redux define five components of Redux (Store, Action, Reducer, Middleware and View), where each one has a different purpose in managing the state.

First, we need to define a Store, which is the container that holds data and functions as the single source of truth for the application and also `AppState`, which is the data structure of your app. You need to inject your app through a `StoreProvider` in the upper most parent component (usually `MaterialApp`) as you can see in Figure B.5, which allows you to access the `Store` from anywhere.

Redux uses Actions for transferring information a triggering some functionality. They can either be empty classes or can carry important data as defined in Figure B.6. Middleware gains access to Store by injecting it into the Store object. This allows the Middleware to intercept actions and do any API calls necessary as can be seen in Figure B.7. This process of calling into the outside world is also defined as a side effect. Actions

[1] https://redux.js.org/
[2] https://facebook.github.io/flux/

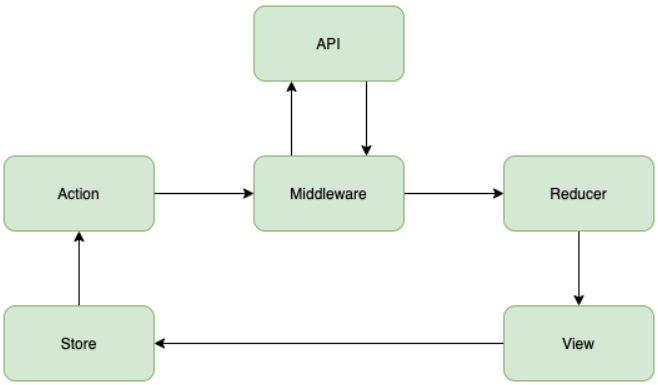


Figure 4.4: Redux architecture

are consumed by reducers that can transform and filter data to be used by the presentation layer which is a switch based on the type of the action as presented in Figure B.8.

Finally we can consume data in the presentation layer by using the `StoreBuilder` widget, which has the store injected and you have access to all the data in the store. Also with the store object, you can dispatch actions triggered by the user, which are further consumed in the Redux cycle. Example implementation can be seen in Figure B.9

4.4 Data management

The biggest part of the application is geopositional data, and the most commonly used format is GeoJSON. According to [5], GeoJSON is a geospatial data interchange format based on Javascript Object Notation (JSON). It also defines several types of JSON objects and the proper use-case in which they are combined to represent the data about features, their geometry and properties.

There are three main types of geometry, that are interesting to us and examples of their structure can be seen in Figure 4.5, Figure 4.6 and Figure 4.7

1. Point - can describe the simplest geometry, a point, used mostly for single objects

2. Line - usually used to describe streets and roads, in our application also crossings

3. Polygon - describes finite number of straight lines that form a closed segment, used for forming boundaries and describe object shape in detail

```
'type': 'Feature',
'geometry': {
    'type': 'Point',
    'coordinates': [-122.414, 37.776]
}
```

Figure 4.5: Point data using GeoJSON

```
'type': 'Feature',
'geometry': {
    'type': 'LineString',
    'coordinates': [
        [-122.48369693756104, 37.83381888486939],
        [-122.48348236083984, 37.83317489144141],
        [-122.48339653015138, 37.83270036637107],
        [-122.48356819152832, 37.832056363179625],
    ]
}
```

Figure 4.6: Line data using GeoJSON

4.4.1 Database

When working with JSON-like data, the go-to solution is NoSQL (non-relational) database that is good at storing unstructured or semi-structured data. They do not have to follow rigid schemas like relational database. There is few types of relational databases, but document store database fits our application the best.

Popular NoSQL database is Firestore, which is part of a bigger ecosystem called Firebase. As stated by [13], Firebase is a Backend-as-a-Service (BaaS) that grew up as a next-generation app-development platform on Google Cloud Platform. It contains a toolset to build, improve, and grow applications and covers most of the services that a developer would have to develop themselves. This platform includes things like analytics, authentication, databases, configuration, file storage, etc... Detailed overview is described in Figure 4.8.

The build part of the service is what is most interesting for our application in early development. The Cloud Firestore is not just a simple NoSQL database but provides a lot more functionality.

- User-based security
  - Google provides their own declarative security language
  - Restricting data access based on user identity and other patterns (one of our requirements)
  - Integrates easily with Firebase Authentication service

```
'type': 'Feature',
'geometry': {
    'type': 'Polygon',
    'coordinates': [
        [-122.48369693756104, 37.83381888486939],
        [-122.48348236083984, 37.83317489144141],
        [-122.48339653015138, 37.83270036637107],
        [-122.48369693756104, 37.83381888486939],
    ]
}
```

Figure 4.7: Polygon data using GeoJSON

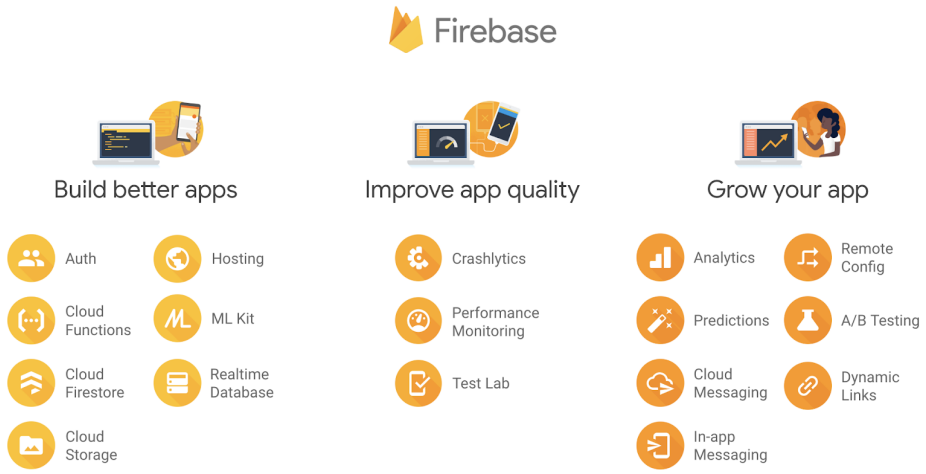


Figure 4.8: Firebase Suite - list of Firebase services [13]

- Data syncing
  - Firebase automatically synchronizes data between devices using streams and listeners
  - Users can access and make changes to their data at any time (even offline)
- Offline persistance
  - Any recently listened to data is persisted to the database
  - The data is cached and is persisted even through app restarts and device reboots

One of our requirements is for our applciation to have a user management system (allowing the creation of accounts and login/logout functionality). This is where Firebase Authentication comes in. It provides backend services, easy-to-setup SDKs and ready-made UI libraries for user authentication in the app. It supports multiple different ways of authentication using passwords, phone numbers or using identity providers like Facebook or Twitter.

Another part of Firebase Suite we will use in our application is Firebase Cloud Storage as we need a place to store multimedia. It provides massively scalable storage for files and functionality to upload and download files directly into you own bucket (which works well with Firebase Authentication to allow access to files in a way you allow) and making it more secure.

4.5 Map framework

The last thing we need to figure out is the choice of the mapping framework that comes with all the functionality we need. Two most significant packages for Flutter are Google Maps and Mapbox. Both packages come with similar usability. They allow location tracking, navigation, markers and other necessary functionality that this project will need.

On one hand Mapbox uses vectors, unlike Google Maps that use raster, for their maps system, which makes them faster to load and more performant. But where Mapbox comes short is, that it is community-driven. Mapbox themselves created it, but after the first release, it was up to the community to improve it. It is currently on version 0.0.5 and is in really early development.

4.6 Conclusion

In the technical analysis, I have talked about a few technologies and solutions we could use to implement the application. I would now like to go through them and justify why I picked them. Flutter was a requirement from the assignment. Nevertheless, I would still pick Flutter if I had the option to pick, because of the quick development cycle and performance advantage over the other cross-platform alternatives.

When analyzing the state management solution, the Lifting State Up solution that is native to Flutter wasn't an option because of its waterfall effect. Flutter developers usually recommend BLoC, but at the time of analysis felt clumsy to me. As a web developer, I felt the most comfortable with using Redux even though it was hard to setup.

There were no real contenders against Firebase. It provided us with all the different pieces of functionality (database, data storage, authentication) we needed and the libraries for Flutter were easy to use as they are all developed by Google, which is the company responsible for Flutter. Both map frameworks had the same functionality, and they were equally comfortable to use. In the end, I chose Google Maps to keep up with the theme, as there was no clear winner in this category.

Chapter 5

Implementation

In this chapter, I start by introducing the process of setting up the project from an integrated development environment to app publishing pipeline in Flutter. Then I showcase an overview, what I implemented in the application and explain interesting parts in greater detail. At the end of this chapter, I showcase unit and UI testing in Flutter and go over the results of the user-testing.

5.1 Project setup

As I am familiar with JetBrains products, I chose to develop using Android Studio[1], which they develop in cooperation with Google. I opted in for using GIT as a version control system, specifically Github, which allowed me to keep track of functionality.

Using Githubs tagging system I created an auto-publishing pipeline using CodeMagic[2], that was trigger by creating a new tag on the master branch. CodeMagic automatically build the application distribution package and published it to the Google Play Console. This made it easy to test on multiple devices by multiple users by allowing them to enter the application beta testing in Google Play store.

I divided the functionality into feature modules (login, map, survey, profile) with common functionality being stored in a core module. Adhering to the Clean Architecture principles, each module was further divided into three separate packages (data, domain and presentation). These are further divided, each having their own responsibilities.

- Data layer
  - Data sources - handles communication to Firestore

[1] https://developer.android.com/studio
[2] https://codemagic.io/start/

- Epics - synchronizes data from Firestore to Redux store
- Models - representation of data from Firestore

- Domain layer
  - Actions - contains Redux actions
  - Reducers - contains Redux reducers
  - Selectors - either simple getters from Redux store or more complex selectors with filters and conditions
  - Use-cases - each use-case can use multiple actions or selectors, are used by presentation layers

- Presentation layer
  - Pages - contains unique pages (aggregate multiple widgets)
  - Widgets - contains unique widgets to the feature

Before diving into the application, I needed a simple way to fill the database with data provided by my supervisor. I created a small client-side React application that transforms the provided data into a format expected by our database. As it was only a side project, I did not implement authentication for this service. Every time I wanted to feed new data into the database, I had to disable security rules temporarily.

One piece of data I needed for my implementation of proximity search queries, I needed for this service to calculate a geohash for every object. Geohash is an alphanumerical string that encodes geographic coordinates of a cell (small area on the map). The longer the string, the more precise location it encodes. I opted for the precision of nine, which encodes the location of a five by five meters area.

5.2 Application functionality

In this part I will go deep into interesting parts of my solution, explain which solutions failed, which succeeded and how I later build upon them. For each I reference application requirements to give better context for decisions I made during the process.

5.2.1 Application overview

At the start, I would like to overview the state of the application in the current version (v2.5.0 at the time of writing this thesis). I implemented all use-cases that are defined in this thesis. Following this paragraph is the list of use-cases with references to screenshots from application and short description.

- Use Case 1 - Login
  - Ability to login through Google (Figure C.1a)
  - Showing progress of authentication with a loader (Figure C.1b)
  - Redirect to the main Map View which triggers a request for access to device's location (Figure C.1c)
- Use Case 2 - Show objects on map
  - Ability to see objects on map with unique icons (Figure C.2a)
  - Ability to open a list with objects sorted by the closest to the furthest (Figure C.2b)
  - Start of UC3 (plus button upper left corner), UC4 (button next to each item in the list - disabled if report on that object exists by the user), UC5 (chart button upper left corner) and UC7 (profile icon bottom navigation bar) all seen on Figure C.2b
- Use Case 3 - Add new report
  - This scenario starts by clicking the plus button in upper left corner shown in this Figure C.2b
  - Scenario can be canceled at any time by clicking the cross in upper left corner or pressing the back space
  - Location can be selected by clicking the map or skipped (Figure C.3a)
  - Object type can be selected or skipped (Figure C.3b)
  - Measurement unit and values for length and width can be selected (decimal point precision) or skipped (Figure C.3c)
  - Photo can be added either from camera or from a library (Figure C.3d)
  - Once photo is selected, it can be cropped or removed, it is also shown on the screen (Figure C.3e)
  - Last selection screen from scenario has a Submit button, with which we send our report to the server
- Use Case 4 - Add report to an object (example for crosswalk)
  - This scenario starts by clicking the button next to the wanted object C.2b
  - Scenario can be canceled at any time by clicking the cross in upper left corner or pressing the back space
  - Series of selection screens (Figures C.4a, C.4b and C.4c is presented to the user for data collection with the option to skip each item
  - This scenario also has the photo selection screen
  - Can be submitted just as UC3

- Use Case 5 - Filter objects
  - Ability to filter objects based on type (Figure C.5a)
- Use Case 6 - Survey overview
  - Ability to see your previous surveys (Figure C.5c)
  - Each item has all info that was collected
  - Reports of new objects have an address created based on geolocation
- Use Case 7 - Profile overview
  - User has ability to see his info (Figure C.5b)
- Use Case 8 - Logout
  - User can logout of the application (Figure C.5b)

5.2.2 Serialization of JSON-like data

As Firestore communicates using documents, which are JSON-like structured data, I had to come up with a solution, how to turn them into our models. The conventional way of representing JSON data in Dart is by using a `Map`, where strings are the keys, and we can have anything else as a value. For our application to interpret a `Map` as a model class, we need to create two helper functions for each model we use in the database. There are two solutions, how to generate these helper functions.

The first solution is using a third-party package. I tried using `json\_annotation package` combined with `json\_serializable` which you can use to annotate classes that should be serializable. This approach generates a new file, which includes helper functions. This solution is easy and fast, but sometimes the generation of packages took up to four minutes with more complex classes.

The second solution was creating the helper functions myself, which is a time-consuming process, but it allows you more atomic control over the serialization. I also appreciated having the helper functions inside the appropriate classes without the need for the creation of new files.

In the end, I preferred the second solution, as it seemed more agile in the early development where things changed quickly, and I wanted more control over it. In retrospective, I should have chosen the automatic generation as it is a more robust solution and in newer versions provide even more control than doing it manually.

5.2.3 Services

The first service I implemented was the `AuthenticationService` (seen in Figure B.10). Thanks to Firebase Authentication, I did not need to implement any

token handling or server logic and just had to implement the client-side logic. I opted-in for signing through the Google Sign In, but any other option (like sign-in through email and password) could be added in the future. I used three separate packages for the service - `firebase_auth`, `google_sign_in` and `cloud_firestore`.

- `google_sign_in` - used for triggering the native authentication modal for Google accounts which after successful authorization returns a credential object (function `signInWithGoogle` in Figure B.11)
- `firebase_auth` - credential object is in turn used to communicate with Firebase Authentication service, which returns a Firebase User (function `authenticate` in Figure B.11)
- `cloud_firestore` - we use the `FirebaseUser` to either create a new `User` in the database or update his data (function `updateUserData` in Figure B.11)

Without authenticating in Firebase, the system denies entry into the application. Even if a malicious user would get into the application without authentication, I setup Firestore rules in a way, that disallows data requests for unauthorized users. In the future, anonymous authentication could be implemented to allow users to collect data as sign-in could be a deciding factor for some people not to use the application.

As tracking users location is one of the requirements, we needed service, that would handle it. I implemented it by using a package called `location` (as seen in Figure B.12), which offers functionality I needed for the application. One of those is requesting permission from the user and the device to use GPS tracking (done by function `requestPermission`), and once permission is granted through the popup, it creates a listener that is updated with current users location. The listener then updates location in the Redux store on every update (function `updateLocation`), keeping it in sync. It is all initialized on the first load of the Map view, which is the first screen user sees once logged in.

Once I implemented more views based on design, I needed a convenient way to switch between them. As the vanilla solution provided by Flutter wasn't sufficient and flexible, I created a service, as shown in Figure B.13. With every navigation, you either need to move forward (see function `navigateTo`) or backwards (see function `goBack`). Later in development, when implementing multiple types of surveys, I needed a way to switch to the view with a specific scenario. For that, I added the ability to pass down arguments which forward them to the view we switch to.

`NavigationService` is linked to the application through the static function `generateRoute`, which is injected into the `MaterialApp` and every-time a route changes (using `NavigationService`) `MaterialApp` triggers the function

with new arguments. Then based on current settings, we render the right view in the switch statement.

5.2.4 Data synchronization

I showed basic Redux setup in the analysis, but that just shows, how data could be stored inside the application. Usually in most applications, we need to synchronize some data from the backend. One type of data we need synchronized in the application are the objects shown on the map. I used `redux_epics` package as a middleware in the Redux implementation, which is able to listen to Redux actions and even have access to the Redux store. What I expected from the synchronization is as follows:

- Consume `StartGeoObjectsSearch` action
- Create a new `Stream` based on that action
- If the action comes when `Stream` is still active, dispose it
- Call the data source to get the updated data
- Dispatch actions to save data into Redux store
- If `EndGeoObjectsSearch` action comes, dispose the Stream

Using `rxdart` package, which adds useful `Extension` methods to `Stream`, I could implement the wanted behaviour of the epic (see Figure B.14) by using `whereType` (filters out other actions from the `Stream`), `switchMap` (converts the emitted item into a `Stream` that is collected by the epic) and `takeUntil` (disposes created `Stream` when condition met).

5.2.5 Modular scenarios

During the development of the application, a few parts of the code needed a more sophisticated solution for them to work effectively and is scalable in the future. One of the requirements for the application is the possibility to create scenarios based on the object type or situation and display it to the user.

I created a system, where it is possible to add different steps of the scenario with the preferred order as you can see in Figure 5.1. This allows the developer to create any combination of selection or question screens without touching any other part of the application.

```
List<Widget> selectWizard(SurveyType surveyType) {
  switch (surveyType) {
    case SurveyType.NEW:
      return [
        LocationSelection(),
        GeoTypeSelection(),
        MeasureSelection(),
        PhotoSelection(),
      ];
    case SurveyType.BENCH:
      return [
        HeadRestSelection(),
        ConditionSelection(),
        MaterialSelection(),
        PhotoSelection(),
      ];
    case SurveyType.CROSSWALK:
      return [
        TrafficLanesSelection(),
        CrossingSelection(),
        MarkingSelection(),
        PhotoSelection(),
      ];
    default:
      return [];
  }
}
```

Figure 5.1: Composable scenarios for data collection

Each `SelectionWidget` configuration and are rendered based on it. Perfecting this dynamic implementation would be creating the actual scenario (widget tree) by generating it with a JSON configuration, which could be served from a server and created even by a non-developer. Even though Dart support run-time reflection, it is not allowed in Flutter due to interference with tree shaking, so we would need to use some code generating library to achieve this.

```
class ConditionSelection extends StatelessWidget {
  final String question = 'In what condition is the object?';
  final Map<String, String> answers = new Map.from({
    'undefined': 'I do not know',
    'pristine': 'Pristine',
    'used': 'Used',
    'damaged': 'Damaged',
    'unusable': 'Unusable',
  });
}
```

Figure 5.2: Simplified implementation of the condition selection widget

5.2.6 Location-based query

Another requirement for the application is to query data based on either user location or camera position (when the user is scrolling through the map). This should also have a small footprint on data usage when not connected to a WiFi. I used geographical queries based on geohashes [see in Figure 5.3), which was implemented in a package called `geoflutterfire`. I had to fork the package as it was using an outdated version of `rxdart` that used deprecated class Observable and update it to the current one (the maintainer later accepted this pull request, and I could then use the updated version).

```
Stream<List<DocumentSnapshot>> getGeoObjects(LatLng cameraPosition)
↪ {
  Geoflutterfire geoFlutterFire = Geoflutterfire();
  Query collectionReference =
  ↪  Firestore.instance.collection('geoData');
  if (cameraPosition == null) {
    return Stream.empty();
  }
  GeoFirePoint location = geoFlutterFire.point(
    latitude: cameraPosition.latitude,
    longitude: cameraPosition.longitude,
  );
  double radius = QUERY_RADIUS;
  return geoFlutterFire
      .collection(collectionRef: collectionReference)
      .within(center: location, radius: radius, field: 'position');
}
```

Figure 5.3: Location based query to Firestore

By querying data based on the location, I came into a performance issue. Both the `location` package (that handles use location updates) and

`GoogleMap` (that provides camera position updates) trigger the updates too frequently, and both Redux store and Firestore were overwhelmed by them (dropping frames per second into single digits when scrolling on the map). Even though we need the current location of both camera and the user, it would be enough to get an update every one to two seconds (as the actual fetch of data is quick).

```
class Debouncer {
  static Map<String, Timer> _timers = {};

  static void debounce(String tag, Duration duration, Function
  ↪  onExecute) {
    if (duration == Duration.zero) {
      Debouncer.cancel(tag);
      onExecute();
    } else {
      _timers[tag]?.cancel();
      _timers[tag] = Timer(duration, () {
        Debouncer.cancel(tag);
        onExecute();
      });
    }
  }

  static void cancel(String tag) {
    _timers[tag].cancel();
    _timers.remove(tag);
  }
}
```

Figure 5.4: Debouncer class

This lead me to implement a simple `Debouncer` class (seen in Figure 5.4). It has a `Map` of `Timers` (each having a unique key) and a function called `debounce`. It only allows a function to be called once the given time has passed. If the function is invoked again and the `Timer` for that function already exists, it is just disposed of. Having the calls debounced improves the performance significantly.

5.3 Testing

This section showcases software and user testing. Three types of automatic testing for Flutter exist - unit (test single function, method or class), widget (test a single widget from UI perspective) and integration (test complete functionality of the app or a single flow in the app). I decided to omit integration tests, as I had trouble setting up the `flutter_driver` package to

function properly. User testing done in this project focused mostly on smoke tests as qualitative tests were not possible due to the pandemic.

5.3.1 Unit tests

Unit tests are mainly useful for verifying behaviour of a small part of the application (as small as a single function) and should be tested with all possible inputs. Flutter provides two packages - test package that provides core testing framework and `flutter_test` that provides additional utility functions. Flutter has a dedicated directory for packages, where all the tests reside. Each test filename needs to end with `_test` for Flutter to register them.

Example of a unit test is the `UserReducer_test` file, which tests inputs for the user state in the application. I mocked two unique users for testing the functionality (seen in Figure B.15) and created three unit tests. They test, if user is saved when `UpdateUser` action is called (Figure B.16), if user is updated, when `UpdateUser` action is called with different data (Figure B.17) and if we can update location by calling the `UpdateUserLocation` action (Figure B.18).

5.3.2 Widget UI tests

Widget UI test assess functionality of single widgets in a confined environment. I used package `flutter_test`, that provides multiple tools for testing widgets.

- testWidgets() - automatically creates a `WidgetTester` for each test case, which allows building and interacting with widgets in test environment
- Finder - allows searching for widgets in a widget tree from your test environment
- Matcher - used for verification of what Finder locates

For example I created a test case to test, how a `SignInButton`. As it is a `StatelessWidget`, it only reacts on passed parameters, which makes testing it quite easy. Expected behaviour of the `SignInButton` is, when it is not in loading state, it displays the Google Sign In text and icon as its contents (Figure B.20). If it is in a loading state, we should find a `CircularProgressIndicator` instead (Figure B.21). For the `WidgetTester` to be able to build our widget, I had to create a helper function (Figure B.19), that encases the tested widget with Material App widget to provide the proper context, that the tested widget is expected to have.

5.3.3 User testing

As already foreshadowed in the section description, due to the current situation around the CoVid-19, user testing had some limitations. I could not ask the testers to go outside and do the selected scenarios in the actual environment. This meant, that the user testing was actually done remotely and by specifying the exact location they should be on the map and also providing them with an image of the location.

The test scenarios are based on data provided by the project Cityplan (crosswalks around Karlovo náměstí in Prague and benches in the vicinity of Masaryk University in Brno). Scenarios are created in a way to cover the full functionality of the implementation and should lead to discovering the most amount of problems or bugs.

(a) : Photo of a crosswalk provided for first scenario

(b) : Photo of an obstacle provided for second scenario

Figure 5.5: Images provided to scenarios

Crosswalk survey

- Login into the application using you Google account
- Accept the request for location tracking
- Move to Karlovo náměstí on the map
- Open the object list
- Start Survey on crosswalk with ID 52
- Fill in the survey based on the provided photo (also add the photo to the survey) on Figure 5.5a
- Submit the report
- Try to survey the same crosswalk again (you should not be able to)
- Go to the Profile section and look at your submitted survey
- Logout

New report

- Login into the application using you Google account
- Accept the request for location tracking
- Move to the corner of street Lazarská and Spálená on the map
- Filter the objects to just show benches (no items should be shown on the map or in the list)
- Create a new report for an object and fill information based on provided photo on Figure 5.5b
- Submit the filled survey
- Logout

The test participants were not required to have any special knowledge before the testing. Users tested the latest version of the application (v2.5.0, build number 24) on their own devices. They were also asked to try to break the application. Following problems were found by the users during the testing described in Table 5.1.

Found issue	Proposed solution
User expects that swipe in profile view will switch section	Add a GestureDetection to the screen to listen to swipe event and switch section accordingly
User expects visual distinguishment of markers on the map (done vs. not done yet)	Do not show objects, that the user already collected information for
User expects select/deselect all functionality in the filter view	Add checkbox that either turns every filter on, or turns every filter off
User expects select/deselect all functionality in the filter view	Add checkbox that either turns every filter on, or turns every filter off
User expects to be able to edit/delete his surveys (eg. add photo retrospectively)	This flow is not expected to be supported
User is confused while surveying, as he has no prior knowledge	This part will be mitigated by adding the user education part, that was not part of this implementation.
User expects his surveys to be shown on the map	Create a new set of markers from users surveys and add them to the map
User expects panel with the list to open on click, not just drag	Add a button that opens/closes the panel from provided controller
User expects a message in survey list to be shown if no surveys present	Add an info text with a call to action button, if there are no surveys done yet by the user
User expects more information in the Google Play store for the application	Add proper description and information about the application in Google Play store

Table 5.1: Problems found by users and their respective proposed solutions

Chapter 6

Conclusion

The motivation for this thesis was the creation of a cross-platform application. It is part of CityPlan project, that has the goal to provide better navigation for visually and movement impaired people. The goal of the application is to provide data using crowdsourcing as the primary source of information. The data is later processed by experts who insert refined data into the navigation system.

I start the thesis by defining the core principles and problems I could face implementing the solution. I specify what is meant by crowdsourcing and what issues are faced when using this data for geographical data collection. I also go through how the application could induce users to collect data and introduce some projects that work on related topics. Lastly, I go through the object types recognized to be essential for such a project.

In the chapter System design, I define the scope of the application and which part of it I implement based on the previous design. Then I provide a comprehensive overview of functional and non-functional requirements and based on them define use-cases needed to span the application scope. In the end, I created a domain model to represent the interconnectivity of entities inside the application.

I continue with a technical analysis with which I introduce technologies I intend to use in implementation and compare options. Firstly I provide a quick overview of the cross-platform framework Flutter and describe the architecture style used. I compare three possibilities when it comes to state management, and I have chosen Redux as the best one. Finally, I choose Firebase Suite to handle things like authentication, database and multimedia storage.

In the chapter Implementation, I start by introducing the process of setting up the project from an integrated development environment to app publishing pipeline in Flutter. Then I showcase an overview, what I implemented in the application and explain a few parts in greater detail. At last, I verify the implementation with unit tests, widget tests and also by user-testing.

6.1 Future work

Even though I full-filled the required functionality, many opportunities for improving and extending the current application are still present, and I plan to continue my work beyond the scope described in the thesis.

6.1.1 Fix issues

The user-testing in Section User testing uncovered multiple problems from the user perspective. I proposed solutions to the issues, but they were not verified nor implemented. I expect to fix them during the upcoming collaboration on the project.

6.1.2 User education system

In future iterations, the application will have a system in place for educating the users. Such a system will educate users on topics like recognizing the right markers on a crosswalk. Initial design and functionality are already defined. As the application is easily extendable, this part should be easy to implement.

6.1.3 Gamification

As talked about in Chapter Problem description, we need to induce users to collect the data. The project owners decided to go with gamification as the key concept. With users already being registered, putting a scoreboard and other gamification ideas into place should be relatively easy. One pain point I can see is the scoring itself, which should probably be done by some backend service, that would aggregate the data. Doing this on the client application could become too bothersome.

Bibliography

[1] Yochai Benkler and Helen Nissenbaum. Commons-based peer production and virtue. *Journal of Political Philosophy*, 14(4):394–419, December 2006.

[2] T-MAPY CEDA Maps, České vysoké učení technické v Praze Fakulta elektrotechnická. Integrace služby hledání tras a navigačního systému pro hendikepované osoby s agendními systémy a open daty měst. `https://starfos.tacr.cz/cs/project/TH03010447`. Accessed: 2020-05-17.

[3] Oleg Chursin. A brief introduction to domain modeling. `https://medium.com/@olegchursin/a-brief-introduction-to-domain-modeling-862a30b38353`. Accessed: 2020-05-19.

[4] Brian Egan, David Marne, Pascal Welsch, et al. Flutter architecture samples. `http://fluttersamples.com/`. Accessed: 2020-05-11.

[5] Internet Engineering Task Force. The geojson format. `https://tools.ietf.org/html/rfc7946`. Accessed: 2020-05-14.

[6] Jeff Howe. The rise of crowdsourcing. `https://www.wired.com/2006/06/crowds/`. Accessed: 2020-05-01.

[7] Aikaterini Katmada, Anna Satsiou, and Ioannis Kompatsiaris. Incentive mechanisms for crowdsourcing platforms. In *Internet Science*, pages 3–18. Springer International Publishing, 2016.

[8] Kacper Kogut. Getting started with flutter bloc. `https://www.netguru.com/codestories/flutter-bloc`. Accessed: 2020-05-15.

[9] Robert C. Martin. The clean architecture. `https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html`. Accessed: 2020-05-20.

[10] Gary Randolph. Use-cases and personas: A case study in light-weight user interaction design for small development projects. *Informing Science: The International Journal of an Emerging Transdiscipline*, 7:105–116, 2004.

[11] Michaela Riganová. Motivation methods for crowdsourcing accessibility attributes. Master's thesis, Czech Technical University, 1 2020.

[12] Linda See, Alexis Comber, Carl Salk, Steffen Fritz, Marijn van der Velde, Christoph Perger, Christian Schill, Ian McCallum, Florian Kraxner, and Michael Obersteiner. Comparing the quality of crowdsourced data contributed by expert and non-experts. *PLoS ONE*, 8(7):e69958, July 2013.

[13] Doug Stevenson. What is firebase? the complete story, abridged. `https://medium.com/firebase-developers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0`. Accessed: 2020-05-18.

[14] Google Flutter Development Team. Flutter documentation. `https://flutter.dev/docs`. Accessed: 2020-05-12.

Appendix A

Use cases

A.1 Use case 1 - Login

Use Case 1	**Login**
Actor:	User
Preconditions:	No preconditions

Success scenario:

1. System requests that the user authenticates
2. The end-user tries to authenticate
3. System validates user's authentication attempt
4. System logs in the user

Fail scenarios:

3.a Failed authentication:

1. System shows failure message
2. User returns to step 1

A.2 Use case 2 - Show objects on map

Use Case 2	**Show objects on map**
Actor:	User
Preconditions:	■ User is logged in ■ System has data about objects

Success scenario:

1. User requests to see objects on a map
2. System requests access to user's location
3. User allows access to his location
4. System presents data based on user's location

Fail scenarios:

3.a User denies access:

1. System does not present data

A.3 Use case 3 - Add new report

Use Case 3	**Add new report**
Actor:	User
Preconditions:	User is logged in

Success scenario:

1. User requests to add a new report
2. System presents a new report scenario
3. User fills in the scenario
4. User submits his new report
5. System saves the report

Fail scenarios:

4.a User cancels action:

1. User cancels the scenario
2. System is returned to previous state

A.4 Use case 4 - Add report to an object

Use Case 4	**Add report to an object**
Actor:	User
Preconditions:	■ User is logged in ■ User did not report on this object

Success scenario:

1. User requests to report on an object
2. System picks a scenario based on object type
3. User fills in the scenario
4. User submits his report
5. System saves the report

Fail scenarios:

4.a User cancels action:

1. User cancels the scenario
2. System is returned to previous state

A.5 Use case 5 - Filter objects

Use Case 5	**Filter objects**
Actor:	User
Preconditions:	User is logged in

Success scenario:

1. User requests to filter objects based on type
2. System presents options
3. User selects filter options
4. System presents filtered objects

Fail scenarios:

3.a User cancels action:

1. User cancels the action
2. System is returned to previous state

A.6 Use case 6 - Survey overview

Use Case 6	**Survey overview**
Actor:	User
Preconditions:	■ User is logged in ■ User created any surveys

Success scenario:

1. User requests to see surveys he created
2. System presents user's surveys

A.7 Use case 7 - Profile overview

Use Case 7	**Profile overview**
Actor:	User
Preconditions:	User is logged in

Success scenario:

1. User requests to see his profile info
2. System presents user's profile info

A.8 Use case 8 - Logout

Use Case 8	**Logout**
Actor:	User
Preconditions:	User is logged in

Success scenario:

1. User requests to be logged out
2. System logs out the user

Appendix B

Code snippets

B.1 State management

B.1.1 Business Logic Component

```
class DataSource {
  Future<RawData> readData() async {
    // Read from DB or make network request etc...
    return new RawData();
  }
}
```

Figure B.1: Data source example for BLoC

```
class Repository {
  final DataSource dataSource;

  Repository(this.dataSource);

  Future<Data> getAllData() async {
    final RawData dataSet = await dataSource.readData();
    // here you can transform, filter, etc... your data
    return filteredData;
  }
}
```

Figure B.2: Repository example for BLoC

```
class BusinessLogicComponent extends Bloc {
  final Repository repository;

  BusinessLogicComponent(this.repository);

  Stream mapEventToState(event) async* {
    if (event is AppStarted) {
      try {
        final data = await repository.getAllData();
        yield Success(data);
      } catch (error) {
        yield Failure(error);
      }
    }
  }
}
```

Figure B.3: Business Logic Component example

```
class PresentationComponent {
    final BusinessLogicComponent bloc;

    PresentationComponent(this.bloc) {
        bloc.add(AppStarted());
    }

    build() {
        // render UI based on bloc state
    }
}
```

Figure B.4: Presentation layer example for BLoC

B.1.2 Redux

```
class AppState {
  List<Data> data;
  AppState(this.data);
}

class MainApp extends StatelessWidget {
  final Store<AppState> store = new Store<AppState>(
    reducer,
    initialState: new AppState(),
    middleware: new DataFetcher(),
  )

  @override
  Widget build(BuildContext context) {
    return new StoreProvider(
      store: store,
      child: new MaterialApp(
        child: new DataComponent()
      ),
    );
  }
}
```

Figure B.5: Store definition for Redux

```
class AppStart {}

class LoadData {
  Data data;
  LoadData(this.data);
}
```

Figure B.6: Example of Actions for Redux

```
void DataFetcher(
  Stream<dynamic> actions,
  Store<AppState> store
) {
    return actions.whereType<AppStart>().flatMap((action) {
    // Read from DB or make network request etc...
      store.dispatch(new LoadData(data));
    });
}
```

Figure B.7: Example Middleware implementation for Redux

```
AppState reducer(AppState appState, dynamic action) {
  switch (action.runtimeType) {
    case LoadData:
      // here you can transform, filter, etc... your data
      return appState;
    default:
      return appState;
  }
}
```

Figure B.8: Reducer example for Redux

```
class DataComponent extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    onInit: (store) => store.dispatch(new AppStart()),
    return StoreBuilder(
        builder: (context, Store<AppState> store) {
          // Here you can consume data from the store
          return Text(store.data)
        });
  }
}
```

Figure B.9: Example implementation of View in Redux

B.2 Implementation

B.2.1 Services

```
class AuthService {
  static final AuthService _authService = AuthService._internal();
  factory AuthService() => _authService;

  final GoogleSignIn googleSignIn = GoogleSignIn();
  final FirebaseAuth firebaseAuth = FirebaseAuth.instance;
  final Firestore firestoreDB = Firestore.instance;

  Stream<FirebaseUser> user;

  AuthService._internal() {
    user = firebaseAuth.onAuthStateChanged;
    user.switchMap((FirebaseUser u) {
      if (u != null) {
        return firestoreDB
            .collection('users')
            .document(u.uid)
            .snapshots()
            .map((snap) => snap.data);
      }
      return Stream.value({});
    });
  }

  Future<bool> signInWithGoogle() async {}
  Future<bool> authenticate(AuthCredential authCredential) async {}
  void updateUserData(FirebaseUser user) async {}
}
```

Figure B.10: Authentication service

```
Future<bool> signInWithGoogle() async {
    store.dispatch(new TriggerLoading(true));
    GoogleSignInAccount googleUser = await googleSignIn.signIn();
    GoogleSignInAuthentication googleAuth = await
    ↪  googleUser.authentication;
    AuthCredential credential = GoogleAuthProvider.getCredential(
      accessToken: googleAuth.accessToken,
      idToken: googleAuth.idToken,
    );
    return authenticate(credential);
  }

  Future<bool> authenticate(AuthCredential authCredential) async {
    AuthResult authResult = await
    ↪  firebaseAuth.signInWithCredential(authCredential);
    updateUserData(authResult.user);
    store.dispatch(new TriggerLoading(false));
    return authResult.user == null ? false : true;
  }

  void updateUserData(FirebaseUser user) async {
    store.dispatch(new UpdateUser(new User(
      uid: user.uid,
      email: user.email,
      displayName: user.displayName,
      photoUrl: user.photoUrl,
      lastSeen: DateTime.now(),
    )));
  }
```

Figure B.11: Authentication service functions

```
class LocationService {
  static final LocationService _locationService =
  ↪  LocationService._internal();
  factory LocationService() => _locationService;
  Location location = Location();
  LocationService._internal();

  void initiliaze() async {
    PermissionStatus permissionStatus = await requestPermission();
    if (permissionStatus == PermissionStatus.granted) {
      updateLocation();
    }
  }

  Future<PermissionStatus> requestPermission () async {
    return location.requestPermission();
  }

  void updateLocation() {
    location.onLocationChanged.listen((LocationData locationData) {
      store.dispatch(
        new UpdateUserLocation(
          new GeoPoint(
            locationData.latitude,
            locationData.longitude,
          ),
        ),
      );
    });
  }
}
```

Figure B.12: Location service

```
class NavigationService {
  static final NavigationService _navigationService =
      NavigationService._internal();
  factory NavigationService() => _navigationService;
  NavigationService._internal();

  final GlobalKey<NavigatorState> navigatorKey =
      new GlobalKey<NavigatorState>();

  Future<dynamic> navigateTo(String routeName, {arguments}) {
    return navigatorKey.currentState.pushNamed(
      routeName,
      arguments: arguments,
    );
  }

  void goBack() {
    navigatorKey.currentState.pop();
  }

  static Route<dynamic> generateRoute(RouteSettings settings) {
    switch (settings.name) {
      case '/login':
        return MaterialPageRoute(builder: (_) => LoginView());
      case '/main':
        return MaterialPageRoute(builder: (_) => MainView());
      case '/survey':
        SurveyArguments surveyArguments = settings.arguments;
        return MaterialPageRoute(
          builder: (_) => SurveyView(surveyArguments.surveyType),
        );
      default:
        return MaterialPageRoute(builder: (_) => LoginView());
    }
  }
}
```

Figure B.13: Navigation service

B.2.2 Synchronization

```
Stream<dynamic> searchObjectsEpic(Stream<dynamic> actions,
↪  EpicStore<AppState> store) {
  return actions
      .whereType<StartGeoObjectsSearch>()
      .switchMap((StartGeoObjectsSearch requestAction) {
    LatLng cameraPosition =
    ↪  store.state.mapData.cameraPosition;
    return getGeoObjects(cameraPosition).map((documents) {
      return new LoadGeoObjects(documents
          .map((document) =>
          ↪  GeoObject.fromJSON(document.data))
          .toList());
    }).takeUntil(actions.whereType<EndGeoObjectsSearch>());
  });
}
```

Figure B.14: Redux Epic for synchronizing data with Firestore

B.2.3 Unit testing

```
final User user1 = new User (
    uid: '12345',
    displayName: 'Jan Kraus',
    email: 'email@email.cz',
    userLocation: null,
);

final User user2 = new User (
    uid: '12345',
    displayName: 'Jan Michal Kraus',
    email: 'email@email.cz',
    userLocation: new GeoPoint(10, 10),
);
```

Figure B.15: User mocks

```
test('should add user to store in response to UpdateUser
↪  action', () {
  final store = Store<AppState>(
    reducer,
    initialState: AppState(
      user: new User(),
    ),
  );

  expect(selectAuthenticatedUser(store).uid, isNull);
  store.dispatch(new UpdateUser(user1));
  expect(selectAuthenticatedUser(store).uid, user1.uid);
});
```

Figure B.16: User is saved - unit test

```
test('should update user info from new user object', () {
  final store = Store<AppState>(
    reducer,
    initialState: AppState(
      user: User.from(user1),
    ),
  );

  expect(selectAuthenticatedUser(store).uid, user1.uid);
  expect(selectAuthenticatedUser(store).userLocation,
  ↪  isNull);
  store.dispatch(new UpdateUser(user2));
  expect(selectAuthenticatedUser(store).displayName,
  ↪  isNot(equals(user1.displayName)));
  expect(selectAuthenticatedUser(store).userLocation,
  ↪  user2.userLocation);
});
```

Figure B.17: User is updated - unit test

```
test('should update user location in response to
↪  UpdateUserLocation action', () {
  final store = Store<AppState>(
    reducer,
    initialState: AppState(
      user: User.from(user1),
    ),
  );

  expect(selectAuthenticatedUser(store).uid, user1.uid);
  expect(selectAuthenticatedUser(store).userLocation,
  ↪  isNull);
  store.dispatch(new
  ↪  UpdateUserLocation(user2.userLocation));
  expect(selectAuthenticatedUser(store).userLocation,
  ↪  user2.userLocation);
});
```

Figure B.18: User location is updated - unit test

B.2.4 UI testing

```
Widget buildTestableWidget(Widget widget) {
  return new MaterialApp(
    home: widget,
  );
}
```

Figure B.19: Helper function for widget tests

```
testWidgets('SignInButton is not loading', (WidgetTester
↪  tester) async {
  await tester.pumpWidget(buildTestableWidget(SignInButton(
    false,
    () {},
    GoogleSignInButton(),
  )));
  expect(find.byType(GoogleSignInButton), findsOneWidget);
});
```

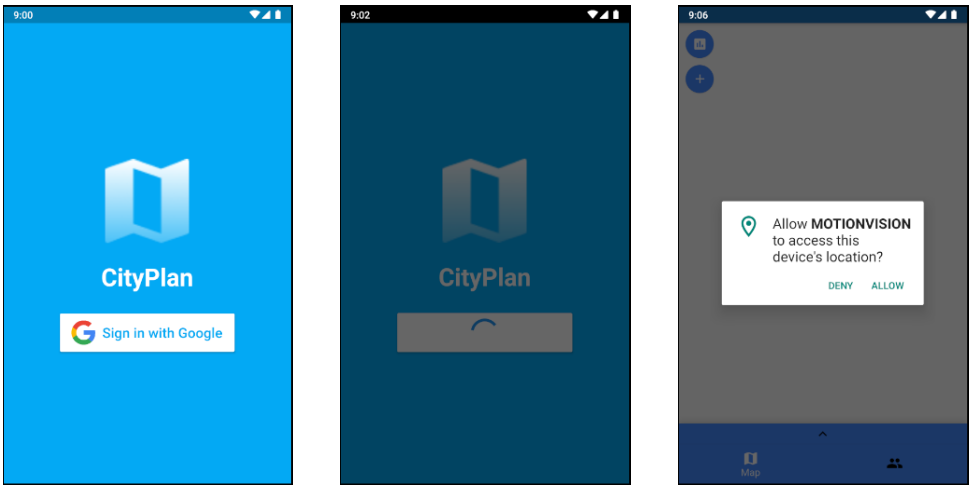
Figure B.20: SignInButton in non-loading state - widget test

```
testWidgets('SignInButton is loading', (WidgetTester
↪  tester) async {
  await tester.pumpWidget(buildTestableWidget(SignInButton(
    true,
    () {},
    GoogleSignInButton(),
  )));
  expect(find.byType(CircularProgressIndicator),
  ↪  findsOneWidget);
});
```

Figure B.21: SignInButton in loading state - widget test

Appendix C

Application screenshots

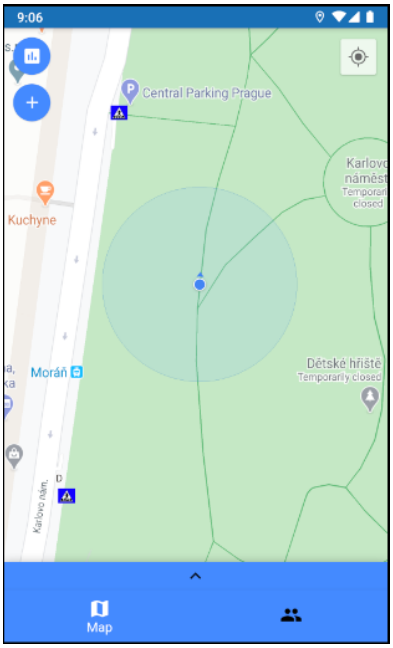


(a) : Login view

(b) : Login view loading

(c) : Allow access to location modal

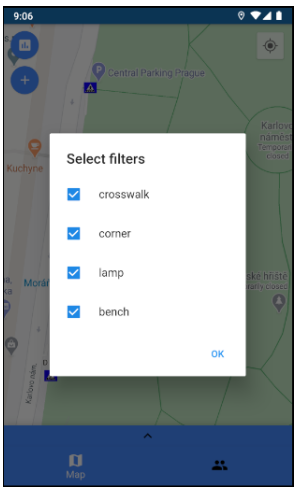
Figure C.1: Login flow



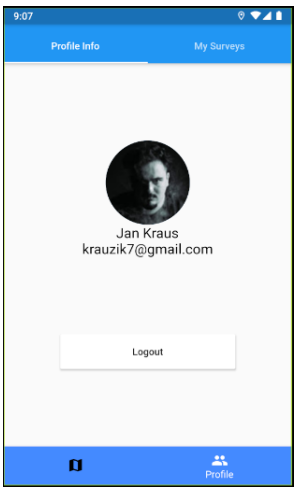
(a) : Map view

(b) : Map view with object list

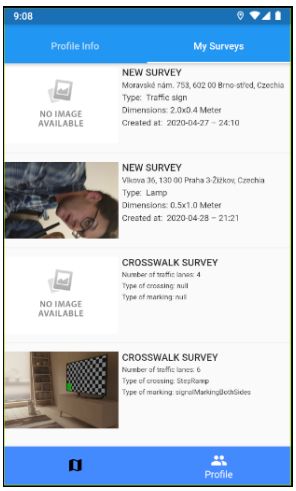
Figure C.2: Map view



(a) : Filter modal

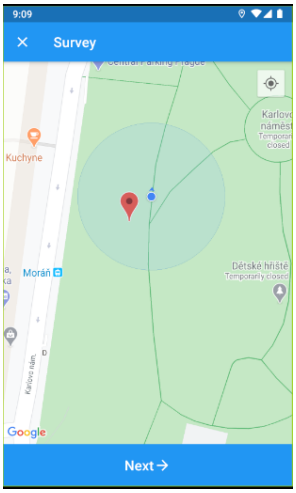


(b) : Profile overview

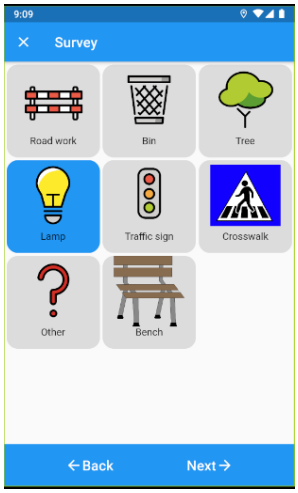


(c) : Survey overview

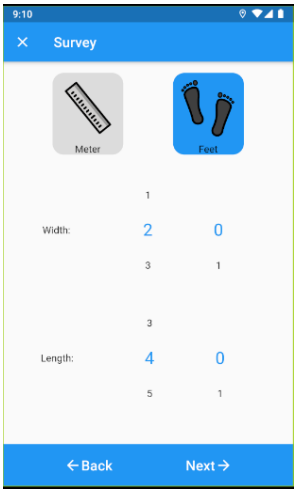
Figure C.5: Profile and survey overview views and filter modal



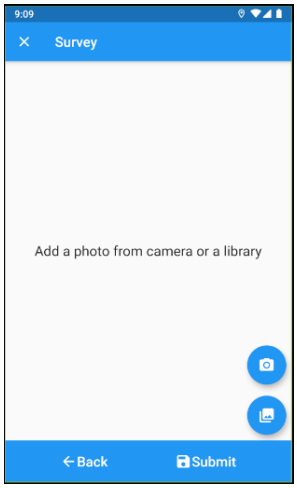
(a) : Location selection



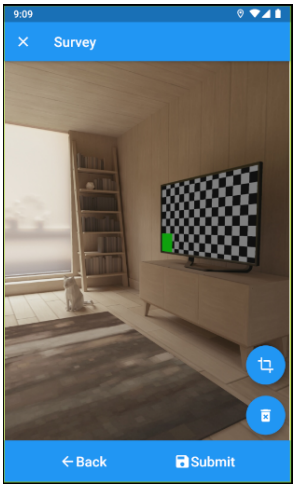
(b) : Object type selection



(c) : Measurement selection

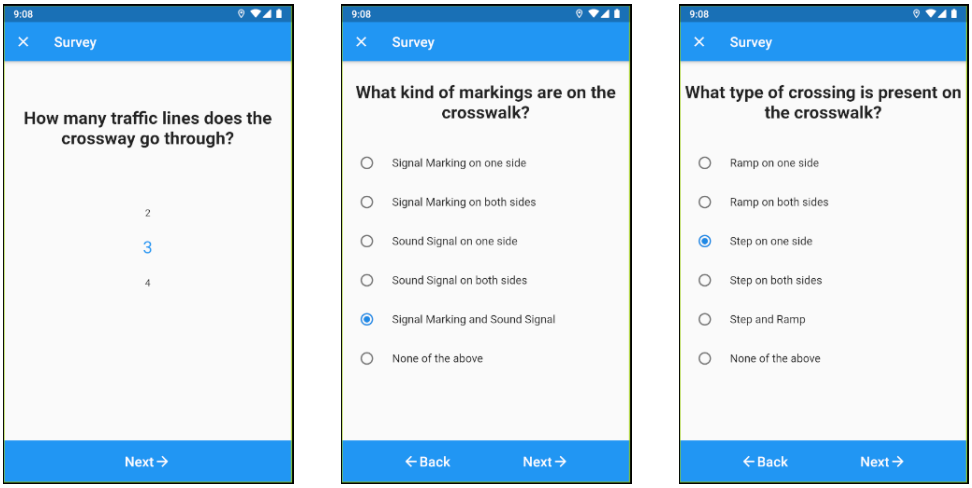


(d) : Photo selection



(e) : Photo selection with photo

Figure C.3: Scenario for new reports



(a) : Traffic lines selection

(b) : Markings selection

(c) : Crossing type selection

Figure C.4: Scenario for crosswalks

Appendix D

Examples of provided graphical design

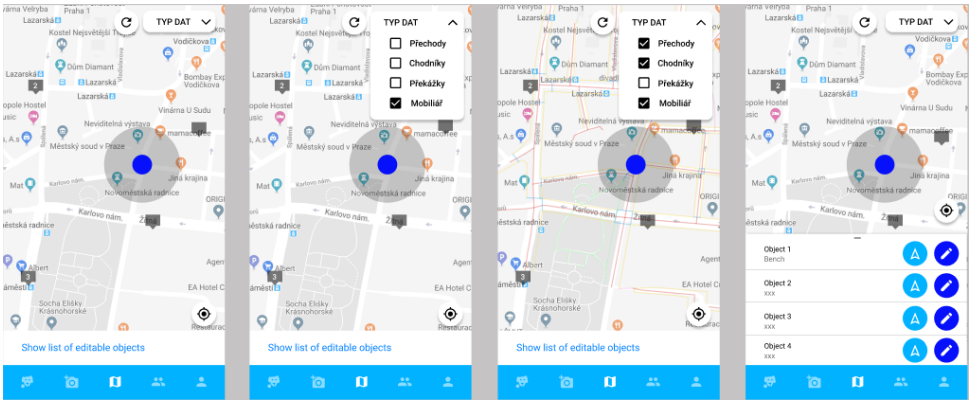


Figure D.1: Provided design of the map interface

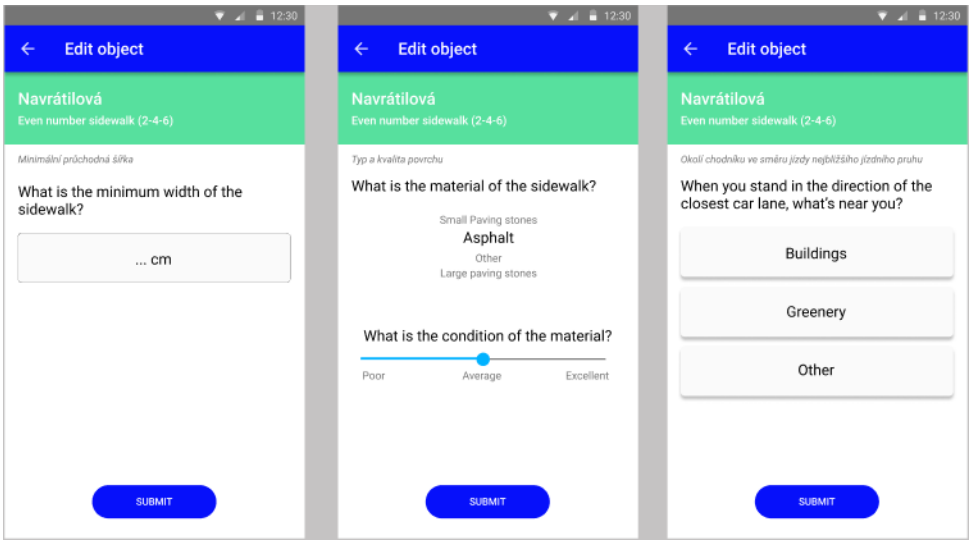


Figure D.2: Provided design of the sidewalk scenario

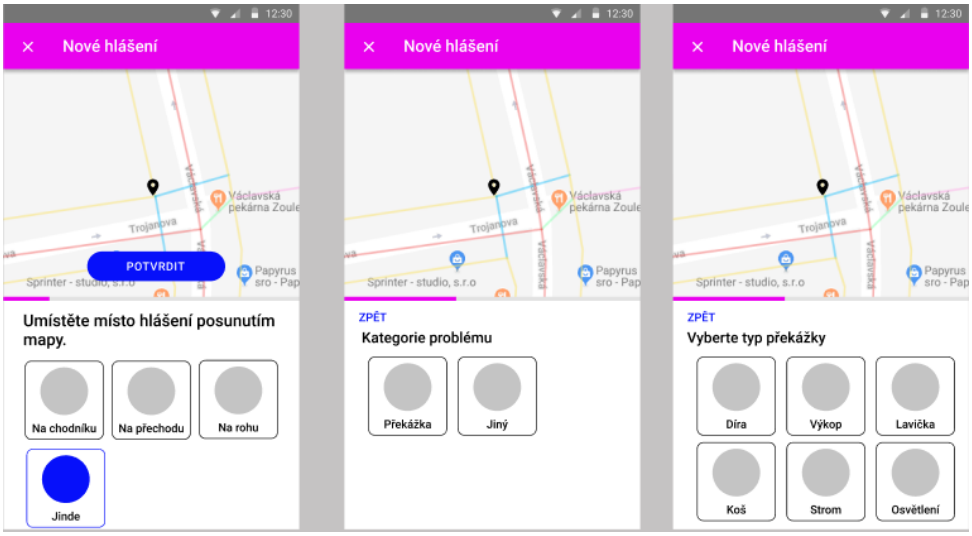


Figure D.3: Provided design of the New Report scenario - part one

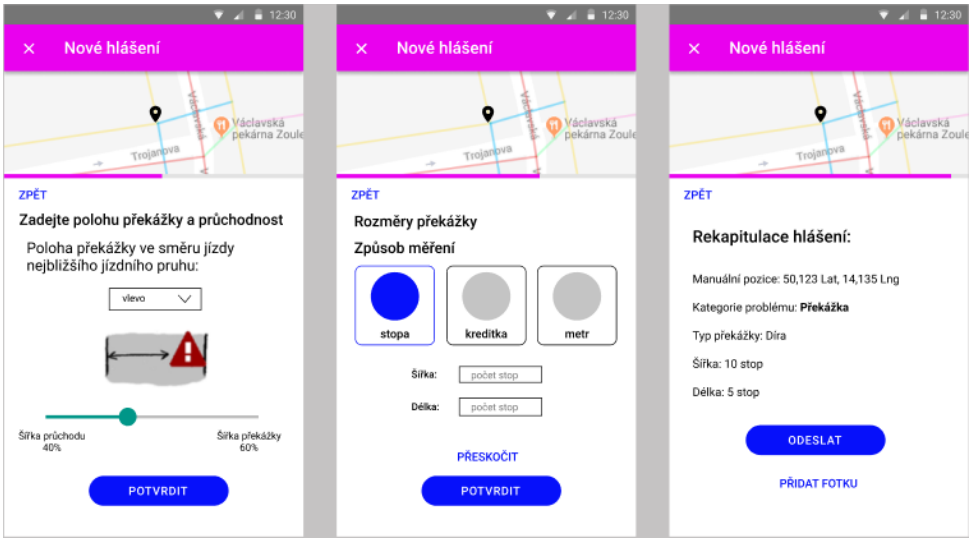


Figure D.4: Provided design of the New Report scenario - part two

Appendix E

Contents of the attached CD

- `source_code` ..... directory with source code
  - `motion_vision` ..... Flutter mobile application
  - `data_loader` ..... utility web application
- `source_latex` ..... directory with LaTex source code
- `user_testing` ..... directory with user testing scenarios
- `thesis.pdf` ..... exported PDF of the thesis