Daniel Filipe Rasteiro da Silva

Implementation of the Fireloc Application

1290 UNIVERSIDADE D COIMBRA

UNIVERSIDADE D
COIMBRA

Daniel Filipe Rasteiro da Silva

Implementation of the Fireloc Application

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering, advised by Professor Filipe João Boavida Mendonça Machado de Araújo and Joaquim António Saraiva Patriarca and presented to Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2021

Faculty of Sciences and Technology

Department of Informatics Engineering

Implementation of the FireLoc Application

Daniel Filipe Rasteiro da Silva

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering advised by Prof. Filipe João Boavida Mendonça Machado de Araújo and Joaquim António Saraiva Patriarca and presented to the
Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2021

1 2 9 0

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

Acknowledgements

First, I would like to express my profound thanks my supervisor, Professor Filipe Araújo for all the knowledge transmitted during the period of this dissertation and his attention and dedication.

To Joaquim Patriarca, for all the answers to my questions and his availability to help to integrate this work with the remaining parts of the FireLoc Project.

To Miguel, Paul and company for all the coffees and chats, serious or not, that kept me going during the pandemic.

To my family and parents, my deepest gratitude and love for always supporting me in my studies, believing, and encouraging me to achieve my goals and writing this dissertation.

This work was carried out under the FireLoc project (PCIF/MPG/0128/2017), funded by the Portuguese Science Foundation.

This page is intentionally left blank.

Abstract

Nowadays, one of the greatest problems we face recurringly is forest fires. Forest fires cause great damage to populations and their livelihoods as well as the environment when not contained early on. The Fireloc project aims at providing a crowdsourced detection system where the users will retrieve positional data and a photo of the fire to submit a report. After the data being verified, it is sent to the authorities, which may use this to quickly plan the best course of action. In this dissertation we will be developing the points of interaction of this system with the users.

This dissertation focuses on developing a mobile application to collect positional data, such as geographical position of the user, and orientation relatively to the magnetic north, and image collection. To develop the mobile application we use the Ionic framework with Angular as the underlying web framework. Using Ionic it is possible to access a smartphone's native functionalities and use them in a web application format, which will be compiled into a native application. In this dissertation, we compiled the mobile application to an android apk. We also use NgRx to manage state and component communication, which permitted simplifying the complex relationships among the various components and services present.

Also, we implemented a web portal to display fire related data using the Angular framework. In the development of the portal, we use Firebase as an authentication provider to implement several methods of authentication. We also use NgRx to handle state management and component communication similarly to the mobile application. Thanks to the shared technologies in both the web portal and mobile application, it was possible to reuse some of the infrastructure, namely services from the mobile application in the portal.

Lastly, in this dissertation we implemented an API to support the functionalities of the web portal. This API was implemented using the Django framework. Django supports GIS manipulation functionalities out of the box, which combined with its simple query API and ORM permitted us to develop complex queries using the geographical data collected in the mobile application.

Keywords

Forest Fires, Mobile Application, NgRx, Ionic, Angular, Web Development

This page is intentionally left blank.

Resumo

Atualmente, um dos problemas que enfrentamos com frequência são fogos florestais. Os fogos causam enormes danos às populações e ao seu sustento, bem como ao ambiente em geral quando não são contidos com rapidez nas fases iniciais. O projeto Fireloc procura providenciar um sistema de deteção por crowdsourcing, onde os utilizadores recolhem dados positionais e uma foto para reportar um incêndio. Após a verificação dos dados, estes são enviados ás autoridades, permitindo que estes possam planear com rapidez a melhor estratégia. Nesta dissertação iremos desenvolver os pontos de interação entre o sistema e os utilizadores.

Esta dissertação foca-se no desenvolvimento de uma aplicação móvel que recolhe dados posicionais, como a localização geográfica do utilizador, a sua orientação em relação ao norte magnético e recolhe imagens. Para desenvolver a aplicação móvel usamos a framework Ionic com Angular a servir de base. Utilizando Ionic é possivel aceder às funcionalidades nativas do smartphone e as usar num formato de aplicação web, que será depois compilada numa aplicação nativa. Nesta dissertação, compilámos a aplicação móvel para um android apk. Também utilizámos NgRx para gerir estado e a comunicação entre componentes, permitindo simplificar as relações complexas existentes entre componentes e serviços.

Também desenvolvemos uma portal web para mostrar os dados relacionados com os fogos utilizando a framework Angular. No desenvolvimento do portal, utilizámos Firebase como provider de autenticação para implementar vários métodos de autenticação. Também utilizámos NgRx para gerir a comunicação entre componentes e gerir o estado do portal à semelhança do que foi feito na aplicação móvel. Graças às tecnologias partilhadas entre o portal e a aplicação móvel, foi possivel reutilizar alguma da infrastrutura, nomeadamente serviçoes da aplicação no portal.

Por último, nesta dissertação implementámos uma API para suportar as funcionalidades do portal web. Esta API foi implementada utilizando a framework Django. Django suporta funcionalidades de manipulaçao de dados GIS, o que, combinado com a sua simples API de queries e ORM, permitiu desenvolver queries complexas com os dados geográficos recolhidos na aplicação móvel.

Palavras-Chave

Fogos Florestais, Aplicação Móvel, NgRx, Ionic, Angular, Desenvolvimento Web

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

AU Authenticated User. 97, 99, 103, 105, 119

BPMN Business Process Model and Notation. 49

CRUD Create, Read, Update, and Delete. 13, 52

FIRMS Fire Information for Resource Management System. 20

GDPR General Data Protection Regulation. 63, 78

GIS Geographic Information System. v, vii, 14, 92

IPMA Instituto Português do Mar e da Atmosfera. 18, 19

ML Machine Learning. 2

MTV Model-Template-View. 14

ORM Object-Relational Mapping. v, vii, 14

OS Operating System. 14

PV Professionals and Volunteers. 97, 99, 101, 103, 105–107, 109, 111, 112, 114, 116–119

SSD System Sequence Diagrams. 35, 40, 41, 45

UI User Interface. 34, 78, 88

UU Unauthenticated User. 97, 99, 103, 105

UX User Experience. 88

XSRF cross-site request forgery. 68

XSSI cross-site script inclusion. 68

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

This chapter presents the context, motivation and objectives for the work developed in this dissertation. We also present the contributions made to the Fireloc project and the structure of the document.

1.1 Context and Motivation

One of the greatest challenges we increasingly face are forest fires. As we may remember almost everyday on news outlets, forest fires of large dimensions spread in Australia (2020), California (2021), Turkey (2021) and also Portugal (2017), threatening large communities and ravaging the land. These fires cause great harm and suffering in neighboring communities, financial losses to the small local agrarian businesses and harm the local flora and fauna. Portugal is a country that suffers from forest fires with an average of 18277 fires and 136502 hectares of burned area by year [4], where many occur in zones of difficult access.

Considering the far-reaching reprecursions of a forest fire, its early detection is of utmost importance to effectively containing it and preventing loss of life and material. The detection strategies used are varied. These include the surveillance towers, patrols by local authorities and even more modern methods, such as the use of drones. While some of these methods can be effective, they are nevertheless costly, both in time and money spent. Another method relies on reports from the population of surrounding areas, but the question arises of the veracity of their reports, since a false report can result in firefighting means being dispatched, depriving a possible fire of fighting resources.

Figure 1.1: Figure obtained from [10]

The Fireloc project, a three year project financed by FCT, aims at providing an inovative solution. The project aims to develop a crowdsourced system for users, through the utilizations of their smartphones, to report fires by collecting some positional data and a photo of the fire as depicted in Figure 1.1. The data will be processed, and through the use of ML algorithms, Fireloc will evaluate the veracity of a report and make the data available to the authorities. Figure 1.2 depicts the flow of the proposed system. This data allows the authorities to quickly identify the location and dimensions of the fire so as to better combat it.

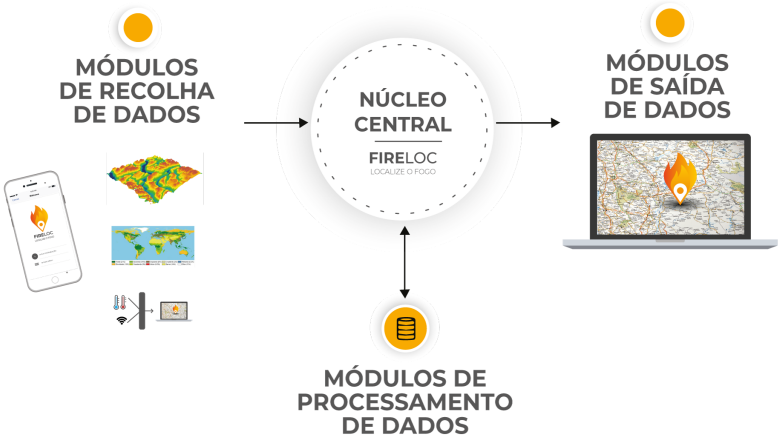

Figure 1.2: Figure obtained from [10]

The work reported in this dissertation was developed in the context of the Fireloc project, aiming at the development of the interfaces for the users to report the fires and to view the impact of their contributions.

1.2 Objectives

The Fireloc project encompasses several important tasks, the sum of which will produce the final system to be released to the general public. Among these tasks, this dissertation focuses on the key tasks of collecting the necessary data for the system and providing the visual output to the users. To accomplish this, this dissertation has three goals. The creation of a mobile application to collect data, the creation of a web portal to visualize data and a support API for the web portal.

The mobile application will provide the user with the means for collecting data to report a fire, such as the positional data and the photographical evidence of the fire. This application is to be integrated with an API that will support the necessary server operations. This API is to be developed separately by other members of the Fireloc team.

The web portal, available to browsers, will provide the user with the means to observe the collected geographic data. The data, treated beforehand, will display the extent of the fires, their locations and the contributions that led to it. The portal's functionalities are to be supported by a dedicated support API.

The support API will provide the web portal with functionalities to prepare the data to

be visualized. The API is to be incorporated in the main server of the Fireloc project.

1.3 Contributions

In the context of the Fireloc project, some of the developed systems can be considered original contributions, namely:

- The successful development of an android mobile application that collects positional data and images to report a fire. This application can collect the user's position, his orientation relative to the magnetic north, and photos of the fire. It can also allow the user to visualize his previous contributions. This application is also ready for iOS deployment.
- The development of a web portal capable of displaying data related to a fire and search of fire events based on multiple criteria.
- An array of services and infrastructure that can be shared between the mobile application and the web portal.

1.4 Document Structure

The remaining dissertation is organized in seven chapters. Chapter 2 contains the relevant technologies for the development of this dissertation. Chapter 3 presents available products with similar characteristics. In chapter 4, the requirements for the mobile app, the web portal and the support API are discussed. Chapter 5 presents the architecture defined to implement the identified requirements. Chapter 6 discusses the implementation of the mobile app, the web portal and support API, focusing on the development of the main components. In chapter 7, the tests performed on the developed products are discussed, including tests to the software and usage in real life. Lastly, chapter 8 presents the main conclusions, as well as the future work.

Chapter 2

Technologies

In this chapter we discuss the relevant technologies for the development of the mobile app, the web portal as well as the API server.

After some preliminary research we identified the most important ones as the framework we will use to develop both the mobile app and the portal, along with the package(s) we will use to render the maps as this is one of the critical components of our design.

The framework, since it will provide important scaffolding to the system and, if chosen correctly will enable us to share blocks of code between the mobile app and the web portal. The map rendering packages since the main operations of the web portal are all based upon the displaying of data whose correct interpretation requires it geographical visualization. We discuss the various choices of technologies for each of the previous points focusing on their analysis following several relevant constraints and criteria we defined to make our decision.

Lastly we give an overview of the technologies chosen to handle application state and to develop the API server.

2.1 Constraints/Criteria

We now specify the constraints for our choice of development framework and map rendering package(s). The choice of the constraints was made based upon factors discussed during preliminary meetings inside the Fireloc Project, such as the need for the quick development of several systems mainly the mobile app and its cost. Therefore, the platform support, the supported languages, the learning curve and the pricing all influence the quick development of our system.

2.1.1 Platform Support

This constraint relates to whether the framework/package is supported by both iOS, Android and Web deployment with similar performance in all platforms.

While we want an application which runs on both iOS and Android, we desire to minimize the workload by having a framework which will allow not only development of code for the mobile platforms, but can also use the same codebase for developing the web application with minimal adjustments.

For the map package, we want one which can be easily integrated with the chosen framework and therefore is also supported by these platforms.

2.1.2 Supported Languages

Which programming languages does the framework/package support for development. This constraint can impact the learning curve of the chosen technologies.

2.1.3 Pricing

Whether the framework/package(s) is free or not. This includes the framework itself and related services if there are any available.

2.2 Frameworks

We now discuss the various pre-selected frameworks present in the market and compare them and choose the best according to the constraints presented in the previous section.

2.2.1 Pre-Selection Process

Given the number of frameworks in the market its volatility regarding new technology, we decided to pre-select five frameworks based on the number of times they appear in selection/ranking articles online and how old/recent those articles are, as well as the platforms those articles are published on. Therefore, we want recent articles (2 years at most, this being from late 2017 onward, this can give us an idea of the consistency of the framework and its popularity) and published on well-respected platforms usually dedicated to the development community.

These criteria serve to filter out less known and undesirable frameworks, since we require a good, maintainable, and well regarded framework.

Table 2.1 contains the results of the pre-selection[1]:

Framework	**Forks**	**Stars**	**Watches**	**Mentions (x/5)**	**Selected (Yes/No)**
Appcelerator Titanium [17]	1.2k	2.5k	194	1	No
Corona SDK	N/A	N/A	N/A	1	No
Flutter [18]	10.1k	79.4k	2.5k	4	Yes
Framework7 [19]	3.1k	14.9k	726	2	No
Ionic [20]	13.1k	39.6k	1.8k	5	Yes
jQuery Mobile [21]	2.6k	9.9k	575	2	No
NativeScript [24]	1.3k	17.8k	713	2	No
PhoneGap [26]	2.2k	1.6k	85	3	Yes
React Native [27]	18.5k	82.7k	3.7k	5	Yes
Xamarin [28]	1.6k	3.9k	435	3	Yes

Table 2.1: Framework Pre-Selection Table

[1]References for the websites visited: [41] [33] [52] [31] [42]

2.2.2 Pre-Selected Frameworks

Flutter

Flutter is Google's open-source solution to mobile, web and desktop hybrid development[2].

- Platform Support: iOS, Android, Web, Windows, Mac, Linux
- Supported Languages: Dart (has similarities with Java and JavaScript)
- Pricing: Free

Ionic

Ionic is an open-source framework developed by Drifty Co.

- Platform Support: iOS, Android, Web, Desktop
- Supported Languages:
  - Languages: HTML, CSS, JavaScript, TypeScript
  - Frameworks: Angular, React and Vue (in Beta)
- Pricing:
  - Community Edition: Free
  - Enterprise Edition: Custom Pricing

PhoneGap

PhoneGap is an open-source distribution of Cordova from Apache Cordova's team.

- Platform Support: iOS, Android
- Supported Languages: HTML, CSS, JavaScript
- Pricing: Free

React Native

React Native is a framework released and maintained by Facebook for iOS and Android development.

- Platform Support: iOS, Android, Web (through 3rd party packages)
- Supported Languages: JavaScript
- Pricing: Free

[2]References: [11] [12] [13]

Xamarin

Xamarin is Microsoft's open-source solution for hybrid development, which extends the .NET framework.

- Platform Support: Supports iOS, Android and Windows (.NET)
- Supported Languages:
  - Mainly Used: C#/XAML
  - As an added Feature: Objective-C, Java, C/C++ Interop
- Pricing: Framework is free. IDE for enterprises 6000$ on the first year and 2600$ for the following years.

2.2.3 Evaluation

For the following discussion we shall assume the criteria to be the constraints presented in section 2.2.1.

Platform Support

Regarding platform support, we are hoping not only for iOS and Android support but also Web, since we require an application which can also run in the browser.

Both Flutter and Ionic fulfil this criteria since they support all three desired platforms. Yet, according to various comparisons, the performance on Web is substantially better using Ionic, while performance in iOS and Android is only slightly better in favor of Flutter. (Supporting links in References)

A React Native app while using essentially the same codebase as a normal React app, still requires a 3rd party package to be able to be deployed as a web application, therefore increasing the reliance on community software and subsequently the risks in using this technology.

Xamarin has no support for web deployment, since it's inserted in the .NET environment, which has a web solution in the form of ASP.NET. To use Xamarin for iOS and Android development, we would need to develop the web app using ASP.NET in an attempt to maximize shared code or use another web framework, which would also increase the workload and development time.

As for PhoneGap, there were no solutions available for web deployment and, to do the web app, we would have to use another framework with the same risks as discussed previously.

Supported Languages

Regarding a framework's support for programming languages, both Ionic, PhoneGap and React Native support JavaScript. Considering the current popularity of JavaScript in the programming community, choosing a framework that supports this language or any subset of it (eg: TypeScript) seems recommendable.

As for Flutter and Xamarin, both require a unique language integrated in their own environment. Yet, it is important to consider other facts.

Flutter is developed by Google, one of the world's leading tech companies and enjoys wide support since it was released, as well as being described by users as easy to learn. The fact it was released by Google may mean it may enjoy a better integration with other Google environment products. If Fireloc is to rely on Google products in the long run, perhaps an investment on Flutter isn't a bad choice.

Xamarin enjoys a similar relationship with Microsoft and therefore the argument above also applies.

Pricing

Regarding pricing all are free except Ionic and Xamarin, yet, Ionic's Community Version is sufficient for the good development of the FireLoc application since only a couple of features are paid. These Ionic features (advanced plugins, exclusive IDE, etc.) are not necessary and will not affect the development of the application if we choose to go with the Community Edition.

For Xamarin, the platform itself is free, but since we are developing this in a university environment we cannot be sure how Xamarin will classify us (company or not). Therefore we may be subject to high costs of production just in a specific IDE which must be used for development.

Therefore all options are equal with regards to pricing, with the exception of Xamarin.

Choice

Given the constraints presented and the framework options, we decided to develop the mobile app using Ionic. The fact we have the base knowledge to operate with it as well as the low cost of developing and maintaining, make it the prime choice. Using Ionic, we will also have a code base we can readily replicate with minor changes to support the web portal.

2.3 Map Packages

We now discuss the various packages and libraries found that may be used to render maps in both our mobile app and web portal.

2.3.1 Pre-Selection Process

The pre-selection of packages for map rendering was easier. Despite the large number of packages, many focus on integrating a root package (ex: mapbox [39]) into an existing web framework such as Angular, Vue or React (ex: vue2-google-maps [50]) therefore we will focus on the main packages, which we identified in Table 2.2.

Package	Forks	Stars	Watches
Leaflet[22]	4.7k	29.6k	955
Openlayers[25]	2.4k	7.6k	424
Mapbox[23]	1.6k	6.5k	347
Google Maps	N/A	N/A	N/A

Table 2.2: Package Pre-Selection Table

2.3.2 Pre-Selected Packages

Since all packages fulfil the requirements of being supported by our selected framework, they already fulfil the Platform Support and Supported Languages criteria.

Leaflet

Leaflet[38] is an open source, lightweight library for interactive maps. Leaflet is free although we have to provide the map layers ourselves. This isn't necessarily a problem since there are several open-sourced providers which can fill this gap.

Openlayers

Openlayers[47] is an open source library used to place dynamic maps in any web page, displaying tiles and data from any source.

Mapbox

Service provided by Mapbox. It provides several map layers and modes as well as the means to render them on a web client. This is a paid service, with a free tier of 50.000 monthly loads, with pricing starting at $5.00 afterwards[40].

Google Maps

Service provided and maintained by Google[29]. It provides both map layers and the means to render them in a web client. Google Maps is a paid service for web development with pricing ranging from $2.00 to $7.00 per 1000 requests[30].

2.3.3 Evaluation

We now evaluate them over the remaining open criteria.

Pricing

Looking at the pricing of each, it is clear we need to discard both Google Maps and Mapbox.

Our objective is to cut costs on subscriptions and usage fees and since Google charges from the beginning it is not a good option.

Mapbox is a hard choice. The free tier is generous, and could cover most of usage cycle of the product. However we need to consider two points. First, the usage is for both the web portal and the mobile app. So we need to consider the usage of both products when looking at the cost. Second, the usage is expected to greatly increase in the summer months, both for the portal and the mobile app. Both points lead us to believe the free usage quota will be reached.

Lastly we have Openlayers and Leaflet. These are open source products with no costs associated. The only cost of using this option would be if we used a paid provider for the map tiles. Since there are also open-sourced tile providers, we can say these options will have no monetary costs associated.

Choice

With regards to the package we will use to render our maps, we decided to go with Leaflet.

Leaflet and Openlayers are similar in many aspects, fulfilling our criteria and being free. However, the fact that Leaflet is considered lightweight and by far the more popular (from which we can deduce there exits more online support for future problems) makes it clear it is the best choice.

2.4 State Management

Considering both the mobile app and, to a lesser extent, the web portal, it is crucial to keep track of the data collected and requested during a session. Another critical point is how we share data between application components, which depending on the method used can become tricky as the application complexity increases. Thus, it is clear we need a tool to help managing our application's state and internal communication.

2.4.1 Vanilla state management and component communication

Using only Angular and its base dependency RxJS, we will go over some possible ways of addressing state sharing and component communication.

Component communication can be handled in several ways. We may employ the usual parent-child communication based on property binding for passing data to a child. Also, we may listen to events when passing data to the parent component using the **EventEmitter** class, which is an extension of the RxJS **Subject** class. However, these are limited ways of communication, since they depend on parent-children relationships for passing data, which are prone to code bloating when sharing too much information, requiring greater attention to the component lifecycle methods to maintain the data updated.

An alternative approach commonly found is to create a service managing various observables, which our components can subscribe to read data. The problem with this approach is the tendency to group the observables into several services, so, as the complexity of the app increases and our components and services become more and more linked, we still have the same code bloating problem, with many components having as a dependency an increasing number of common services.

Adding to this increase of complexity, we have yet to solve the problem of storing the data. The easiest way is to keep the data in the responsible component or service, and pass it to

whatever component or service needs it, using one of the previously discussed methods. A problem arises from this solution when we consider the Angular lazy loading mechanism. What happens if some component needs data from some part of our application, and we have yet to visit the component handling it? To solve this situation, we might create a repository for that data, but we are adding complexity to an already complex solution, and we still have no way of ensuring simple properties, such as, data immutability, ready access, and maintainability.

2.4.2 State management flow using NgRx

In the Angular community, a common solution for the problems presented above in applications of higher complexity is the use of NgRx, which will be adopted in this dissertation. NgRx is a framework for building reactive applications that provides an implementation of the observable store pattern, which is inspired on Redux.

Figure 2.1 depicts the state management lifecycle using NgRx. Starting with the component, this will emit actions that can contain a payload, which may or not contain data. These actions will trigger reducers, which will update the store based on the action type and payload. The selector listens to the store and each time a value is changed it will emit the value to any subscribed components. Lastly, an action can also trigger a side effect. The effect will interact with the services of our application and can also dispatch actions.

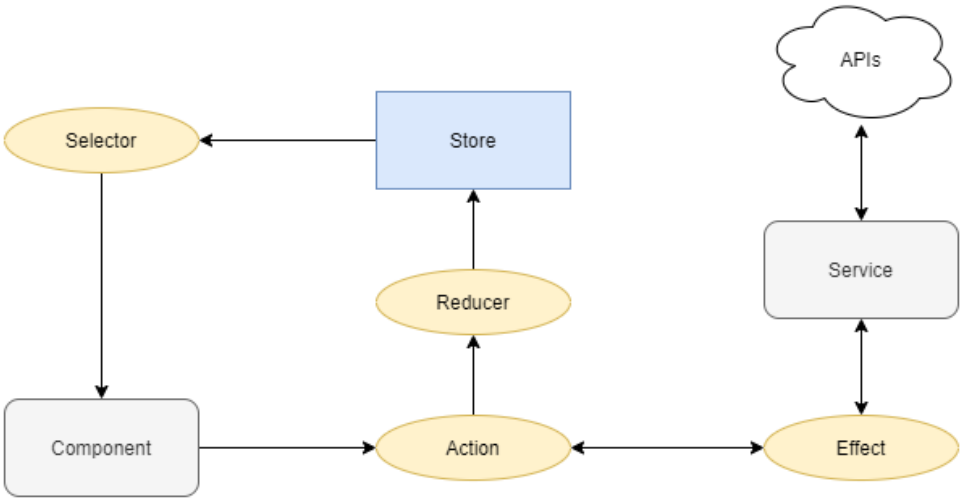

Figure 2.1: NgRx Lifecycle Diagram

2.4.3 NgRx key concepts and definitions

The following concepts are important to understand how NgRx works inside the Angular environment. These definitions were based on the official documentation [44].

Store

Store is a single, immutable, centralized, serializable and type safe data structure used to store our application data, which can also be persisted and rehydrated if necessary. It can be accessed by all components and services through the use of selectors. Also it is available throughout the entire lifecycle of the components. The store can be divided into feature

stores that provide new pieces of state — feature states. The concept of feature store we will be developed in later chapters.

When integrating NgRx store with Angular components, the only import necessary in the component is the store.

Action

Actions are unique events dispatched from components and services. They have the structure of a JSON object with the following elements:

- type: a unique string detailing the type of the action, usually following the convention "[Source] Description";
- props: an optional property in the form of a JSON object used to pass data along with the action.

Reducer

Reducers are pure functions that handle the transition between consecutive application states. This is accomplished by taking the current state and the latest emitted action, and generating a new state. The type of transition made by the reducer is based on the type of the action emmitted. When generating the new state, the reducer can generate it with data provided by the action through the props parameter or with any predefined value. A common example for the latter is the implementation of flags, where an action without props is emmitted, and a single store parameter is changed based on the action type.

To create a reducer, we use the function **createReducer()** that takes as input the initial state, the functions to handle the state changes along with the actions that trigger them.

Selector

Selectors are pure functions used to obtain slices of the state. They provide useful features such as portability, memoization, which is useful when considering performance issues, composition, which allows using various selectors to create other selectors, testability and type safety. These properties allow for safe and unique selectors, which take from the state only the data required at the time.

To create a selector we need two functions from @ngrx/store, the **createFeatureSelector** and the **createSelector** functions. The former creates a feature selector for a given feature state, and the later, which takes the new feature selector, creates the selector for a property in the feature state.

Selectors are synchronous functions by default, and will emit the latest value, when subscribed to, and will emit every new value afterwards. When integrating the NgRx store with Angular components any value to be used from the store will come from a selector in the form of **Observable<T>**, where T is the generic type of the data from the store.

Effect

Effects are a side effect model for the NgRx store powered by RxJS. Effects are long running services, which use observable streams to provide new sources of actions, and listen to every action dispatched by the store. These reduce state based on external interactions such as fetching data from an API and other interactions, which the component does not need to explicitly know about. Therefore, effects allow to isolate the components and reduce the number of directly injected dependencies.

Considering what has just been said about effects, the typical use case will be the following. We create an observable stream from the actions Observable provided by effects. This is accomplished passing it RxJS and NgRx operators to build a chain of desired operations to affect a given type of data. The most common RxJS and NgRx operators found in this dissertation will be the following:

- ofType(): allows to restrict the actions the effect listens to the actions passed in argument;
- switchMap(): returns an Observable based on an operation performed over the input values, used when making API calls;
- map(): maps a source value according to a projector functionwith the resulting value being emmitted as Observable, this allows to dispatch NgRx actions as the resulting Observable;
- tap(): allows the execution of code outside the chain.

Entity

Entity is an adapter that provides an API to manipulate entity collections. It has a state with two elements:

- ids: an array with the collection ids;
- entities: A dictionary with the entities of the collection indexed by their ids.

The entity adapter also provides CRUD methods and selectors for the data. It can also be used as a feature store. This is particularly useful when considering some of the data we will be manipulating, as described in later chapters.

To use the Entity State, we need the feature state to extend **EntityState<T>**, where T is the type of the entity, and to add it to our main state. In the creation of the reducer functions, we create the adapter by calling the **createEntityAdapter<T>()** function. From the adapter we can also employ the method **adapter.getInitialState()** to obtain the initial state necessary for creating the reducer, using the **createReducer()** function.

2.5 API Server Technologies

Now, we speak about the technologies already chosen to develop the API server. These were already defined since the API server is to be integrated into the existing infrastructure of the Fireloc Project. Therefore there will not be a selection process as with the previous client-side technologies.

2.5.1 Django

To build the server infrastructure, it was decided to go with Django[7]. Django is a python framework used to build web applications, which follows the Model-Template-View (MTV) design pattern.

Django is also highly extensible, supporting packages to fulfill several roles such as API provision (perhaps the most relevant in our case). These features are easily integrated into the working app, though the settings files.

These packages also allow us to use Django past the original MTV architecture. In our case, it is of value the internal ORM and the simplicity of its use, while we provide the views and client related logic separately using our front-end solutions: the mobile application and the web portal.

For this project in specific, it is of particular value Django's support for creating GIS applications, this will help the development of the geographical server-side components.

2.5.2 GeoServer / GeoNode

GeoServer[16] is an open source server for sharing geospatial data. It implements the standards of Web Feature Service (WFS) used to offer direct fine-grained access to geographic information at the feature and property level[53]. Web Map Service (WMS), used to provide simple HTTP interfaces for requesting geo-registered map images from distributed geospatial databases, which can be displayed in browser applications[54]. And Web Coverage Service (WCS) used to offer multi-dimensional coverage data over the internet[51].

We will use GeoServer to distribute the geospatial data needed to populate the web portal maps. The data itself will be stored and managed via GeoNode[15].

GeoNode is a geospatial content management system, built with Django which allows for integrated creation of data, metadata, and map visualizations (from multi-layer interactive maps to embedded ones). Each dataset in the system can be shared publicly or restricted to allow access to only specific users. GeoNode is also easily extensible to allow modification to meet the requirements of each application[48].

2.5.3 Docker

Docker[8] is a virtualization tool, consuming less resources than a virtual machine, which delivers software packages in containers. Each container is an isolated executable unit which packages application software as well as any dependency libraries and configuration files, communicating over previously defined channels, running on any OS.

In this project we'll use Docker to separate each of the server modules. This is due to the various dependencies each module has which can sometimes enter in conflict due to different versions needed. By using Docker, we guarantee the independence of each module over the others, as well as also assuring they are able to efficiently and easily communicate among each other.

2.6 Summary

In this chapter we presented the core technologies to be used in the development of the mobile application, web portal and the Geoportal API, along with their selection process when several options were available.

As development framework for the mobile application, we chose Ionic with Angular. This choice took into consideration the supported languages, pricing and the ability to reuse infrastructure for creating the web portal. For the map package, and considering the frameworks to be used, we chose Leaflet.

By choosing Ionic with Angular, we require a state management package and for that we decided on NgRx, the more mature of all available options.

For implementing the server, the Fireloc project predefined the Django framework, to be used in conjunction with Geoserver and Docker.

Chapter 3

Similar Products

In this chapter we discuss some of the relevant mobile apps and web pages which served as base to our design decisions.

First we discuss mobile apps whose purpose, like ours, is to collect user data through user input. We will give a brief overview of the app and go over the main points we judged to be of value to our development.

Next, we go over the web portals. These will focus on displaying information to the users, specifically geographical data. We will discuss their purpose and our main takeaways.

3.1 Similar Apps

In this section we discuss the similar mobile apps found. We chose Fotoquest Go[14], due to its purpose for using geographical data we intend to collect.

3.1.1 Fotoquest Go

Fotoquest is an app designed to collect evidence of how quickly our landscape is changing. This is done through "quests" where the user goes to a certain point in the map and takes a photo of his surroundings.

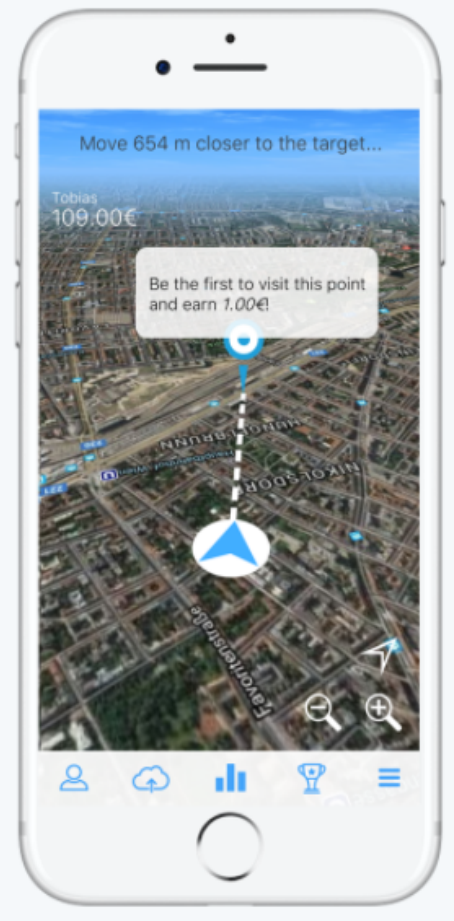

Figure 3.1: Fotoquest

As shown above, Fotoquest uses the phone's gps to provide directions to the goal. Once the user is close enough to one of the objectives, new indications are given before asking him to take a photo of his surroundings which is then uploaded to their services, as well as asking what has changed in the landscape.

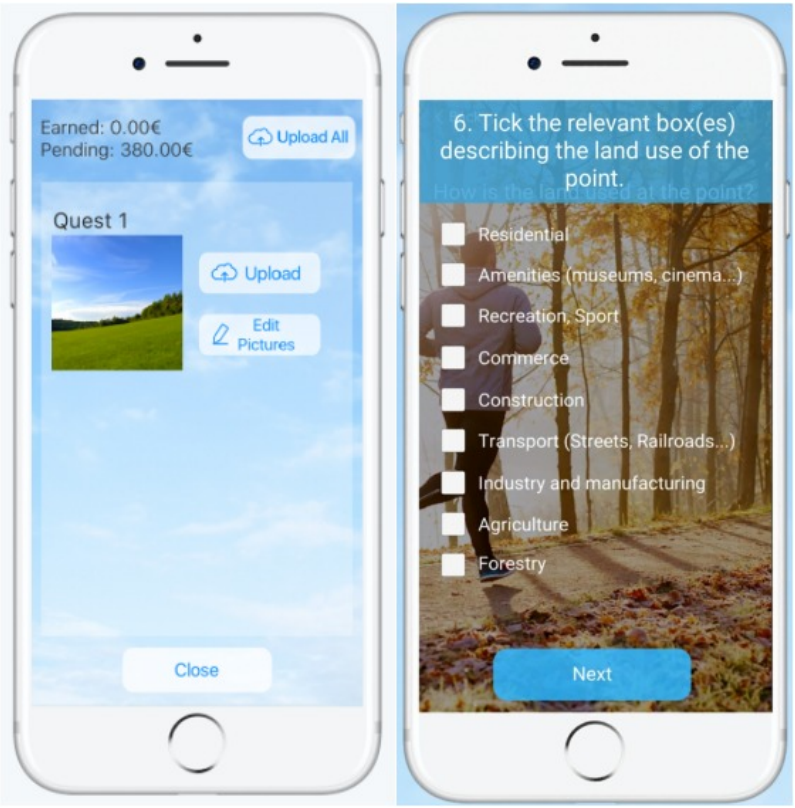

Figure 3.2: Fotoquest Steps

3.1.2 Conclusions

Fotoquest has characteristics that may help us, such as the detailed process of step-by-step instructions. However, there are also lessons about what we should not do. Namely the option to edit the photos, something our system must not allow so as to avoid adulterating possible relevant data.

3.2 Similar Portals

In this section we discuss the similar portals found. We chose the IPMA and NASA pages due to their similarity of purposes with the portal we are going to develop.

3.2.1 IPMA

IPMA is the official page for the Instituto Português do Mar e da Atmosfera. The services provided in this page range from meteorological forecasting to sea wave height to seismic activity. In this website, there are two elements of interest for the project. The seismic activity map[34] and the fire risk map[35].

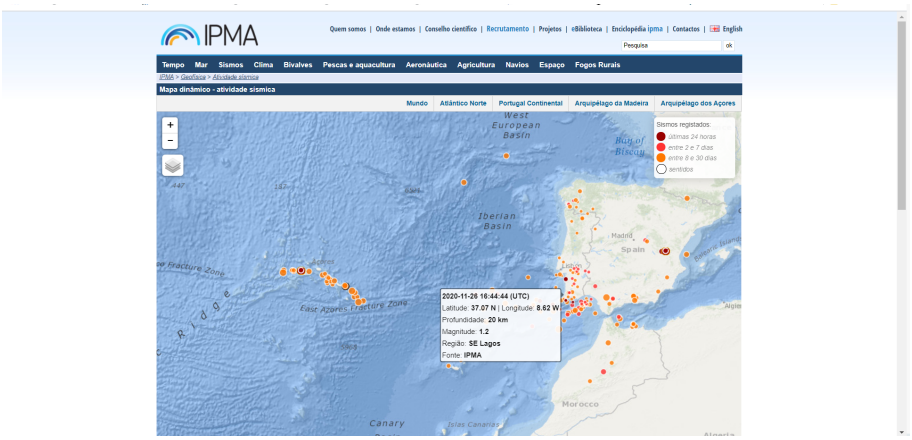

Figure 3.3: IPMA Earthquake Page

Figure 3.3 shows the seismic activity map. In it we can see the map occupies the greater part of the page and secondary information is thrust to other parts of the page.

In the map, we can also see the events marked and when we hover over them we can see relevant details.

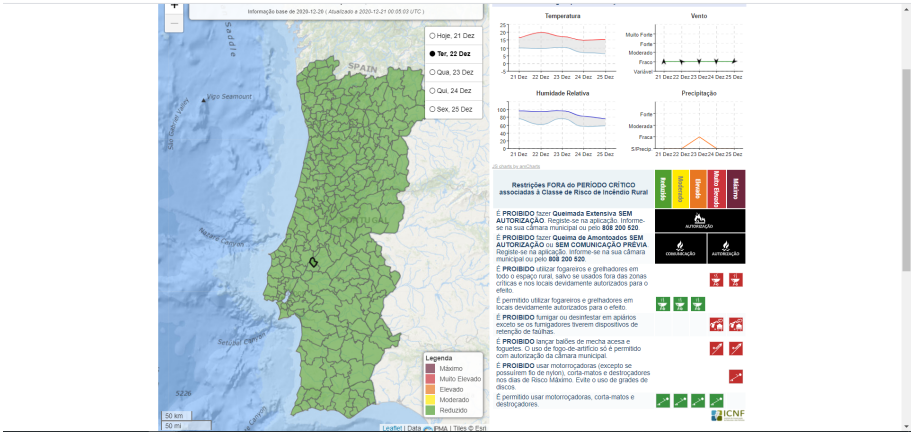

Figure 3.4: IPMA Fire Risk Page

In figure 3.4 we can observe the fire prevention page of IPMA. Here we are presented with a map of Portugal as a whole with its subdivisions clearly defined. Once we click on any of these divisions, the page zooms and centers on the chosen county, displaying relevant data in a popup as seen in Figure 3.5.

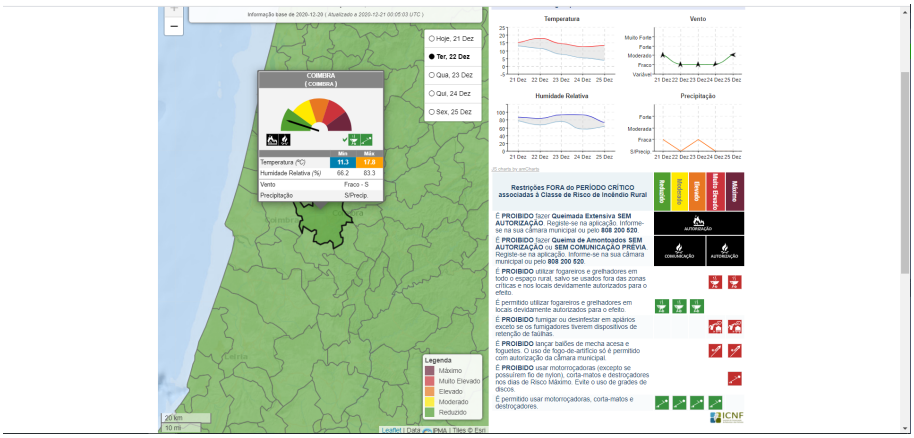

Figure 3.5: IPMA Fire Risk Selected Zone

3.2.2 NASA

NASA, National Aeronautics and Space Administration, is a north american agency responsible for space exploration and research. NASA also monitors Earth from orbiting satellites and posts such information online, among them fire related data through the FIRMS[43] portal.

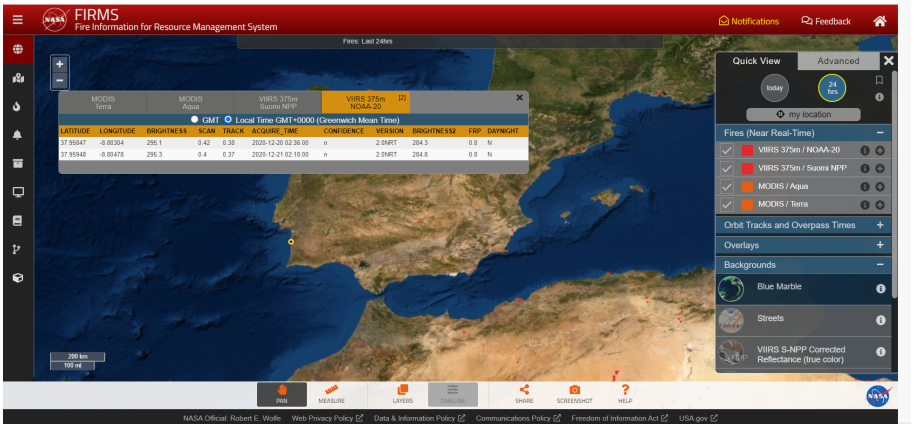

Figure 3.6: NASA FIRMS Portal

Figure 3.6 the main page of FIRMS, in it we can see a world map occupying the whole page. In the map we can see several fire events, which upon clicking display relevant information. We can also see several options of layers and tools to help visualize the data.

3.2.3 Conclusions

Given the pages shown before, we can take the following conclusions for the development of the web portal. First, the map should be the central component of the main page. The map should allow for several layers of information and the events should display some information when hovered upon as well as legends for any symbol or significant color used if there is a need to distinguish.

From IPMA we can also take the division of areas. The Fire prevention page clearly displays the boundaries and combined with other information layers should provide a good

base for our map.

Next is that any secondary task not related to the information being displayed on the map should be separated into other pages. Complementary information to the map is usually displayed on the sides.

3.3 Summary

In this chapter we presented products available similar to the applications we are developing in this dissertation. From them we concluded several aspects in the implementation of a flow of actions in the mobile application and when displaying information in the portal.

Chapter 4

Requirements

In this chapter we discuss the requirements for this dissertation. The chapter is divided between the three main objectives for the dissertation, the mobile app, the web portal and the API server.

For both the mobile app and the web portal, we will discuss the context diagrams for each identified user. From the context diagrams, we will obtain our use cases which will detail the interactions between the user and the system and how it should respond. Next, we will discuss the non-functional requirements that will support the app and the portal.

Finally, we will discuss the API server, detailing the operations it needs to support.

4.1 Fireloc App

In this section, we defined the requirements for the Mobile App.

4.1.1 Context Diagrams

For the Fireloc App, we have identified two types of user stakeholders: the unauthenticated user and the authenticated user.

In Figure 4.1, we present the context diagram for the Unauthenticated User with its respective system interactions. This user only has access to authentication interactions.

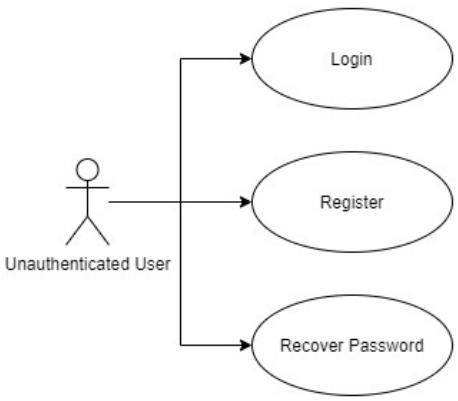

Figure 4.1: App - Unauthenticated User Diagram

The authenticated user and its interactions are represented in Figure 4.2. This user type refers to any Unauthenticated User that has either logged in or registered.

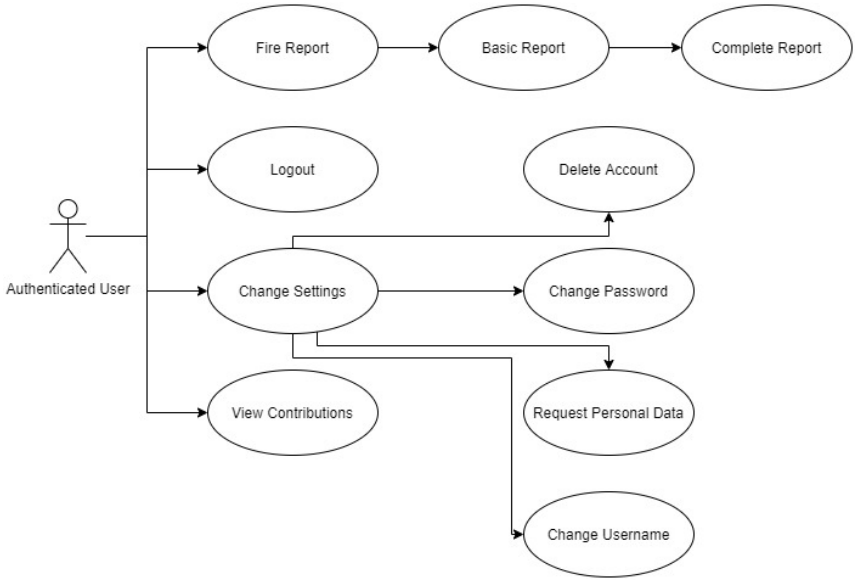

Figure 4.2: App - Authenticated User Diagram

The interactions of the authenticated user are divided in three groups: data gathering and visualization, account management, and authentication management.

Data gathering and visualization interactions encompass fire reporting and viewing the user's contributions. Fire reporting is the main interaction with the mobile app, having two operational modes, a basic report mode with the 'must-have' information and a complete report mode with a more detailed set of gathered data. Concerning visualization interactions, the app provides means for the user viewing his own contributions.

The user management interactions encompass deleting the account, changing the password, changing the username, and requesting the user's personal data.

Finally, the authentication management concerns the logout interaction. When the user logs out, he resumes his previous role as an unauthenticated user with all the associated interactions.

4.1.2 Use Cases

In this subsection we define all interactions of the user with the mobile app identified previously through use cases.

The mappings of the interactions for each stakeholder to the respective use cases are found in tables 4.1 and 4.2. The definition of each use case can be found in Appendix A.

Designation	**Use Case**	**Success Criteria**
Login	Login UC1	User can log in
Register	User Registration UC2	User can create an account
Recover Password	Password Recovery UC3	User can recover the password

Table 4.1: App Unauthenticated User Use Case Mapping

Designation	Use Case	Success Criteria
Logout	Logout UC5	User can log out
Change Username	Change Username UC11	User can change his username
Change Password	Change Password UC10	User can change his password
Delete Account	Delete Account UC4	User can delete his account
View Contributions	View Contribution UC9	The user can view his contributions
Basic Report	Report Fire UC7	User is able to submit a report
Complete Report	Complete Report Fire UC8	User is able to submit a complete report

Table 4.2: App Authenticated User Use Case Mapping

4.1.3 Non-functional Requirements

In this section we go through the non-functional requirements for the mobile app, which are usability and latency.

Usability

Given the fact that the main purpose of this app is gathering information, we must ensure it is as easy to interact for our users as possible.

Among the various functionalities, the reporting process must be of easy access once the user opens the app, as we must account for possible external factors when he makes a report. Also, we must ensure that each piece of collect information is the central point of the interaction on each page. Following a simple, intuitive design that highlights the action needed. However we must also account for human error. Thus, we need to reduce the interactions that require input of information to those absolutely necessary, and give appropriate feedback to the user, as to whether he has completed those interactions in a satisfactory way or not.

To measure the usability requirement we will use the Nielsen Usability Checklist/Euristics.

Latency

As with any mobile app, one of the key aspects users notice is how long they have to wait for content or for an operation to resolve. Thus, low latency is an essential characteristic that we need to cover. This implies we must assure low latency for the majority of our requests.

According to a study from Appdynamics [1], users do not notice values inferior to 1s but a majority will abandon the use of an app for waiting time values above 4s. So, we have set our latency target value to be inferior to 4s.

4.2 Fireloc Portal

In this section, we define the requirements for the Fireloc Portal.

4.2.1 Context Diagrams

For the Fireloc Portal, as in the case of the Fireloc Mobile App, we have two types of users, which are the unauthenticated user and the authenticated user.

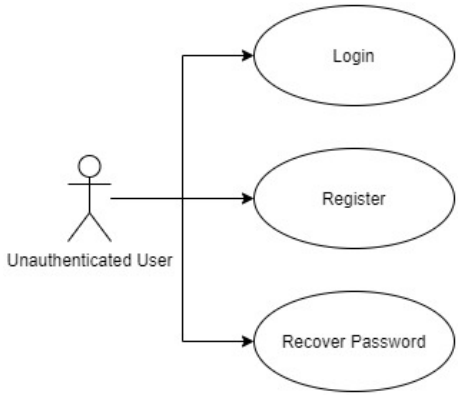

Figure 4.3: Portal - Unauthenticated User Diagram

The context diagram of the unauthenticated user is displayed on Figure 4.3. This user only has access to authentication interactions, since we aim to restrict access to advanced interactions we have with the Fireloc system to registered users.

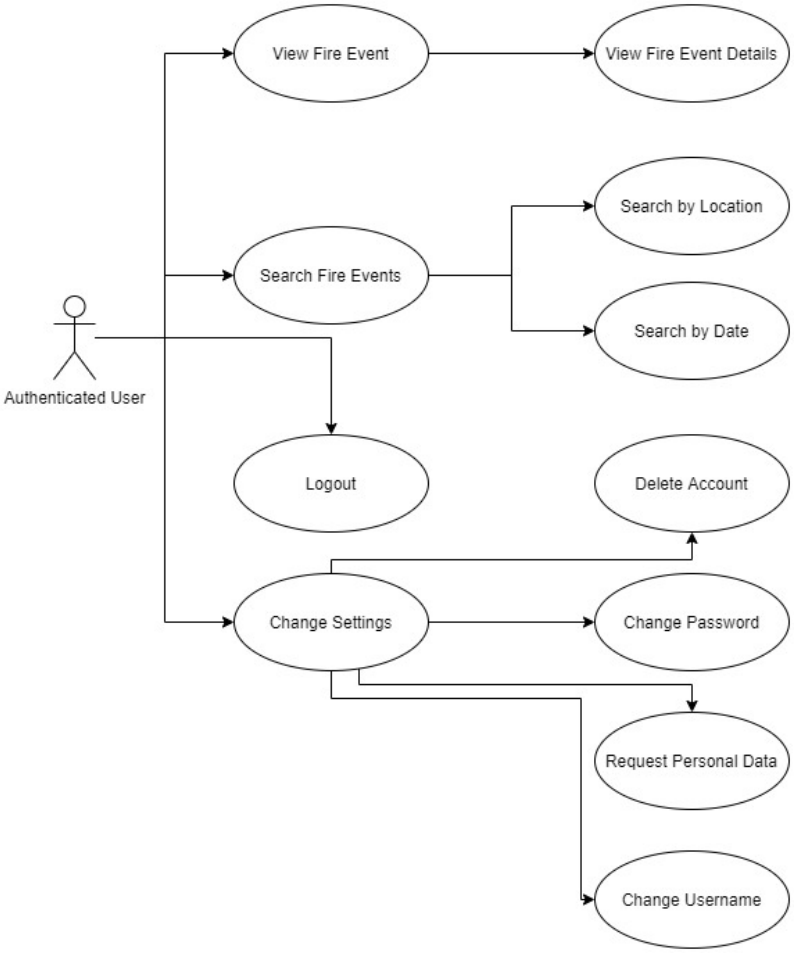

Figure 4.4: Portal - Authenticated User Diagram

The context diagram in Figure 4.4 shows the authenticated user interactions. Like the ones defined for the mobile app, they are restricted to unauthenticated users that have logged in or have registered.

These interactions are divided into four groups: searching, visualization, account management, and authentication management.

The visualization interactions comprehend viewing the results from our queries in the form of fire events and their details. The account management and authentication interactions are analogous to the ones defined for the mobile app.

The authenticated user can be subclassified into four types of users. All of them have access to the same set of operations, but the results and detailed information are filtered according to each type's privileges.

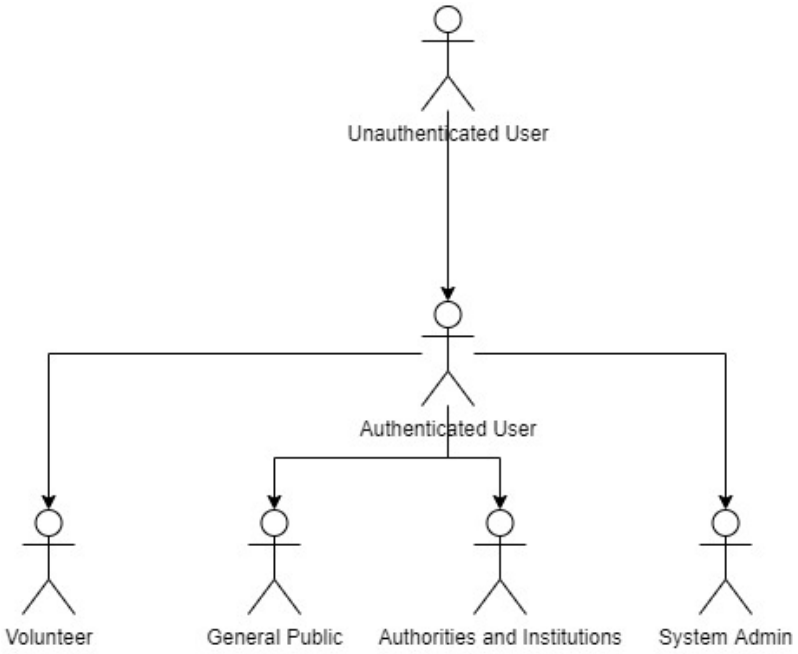

Figure 4.5: Portal - Hierarchy Diagram

The context diagram in Figure 4.5 shows these four types of authenticated users: system admins, users belonging to authorities and institutions, volunteers, and general public. Volunteers are users that have made a contribution to the system, while the general public references users that haven't made any contribution.

4.2.2 Use Cases

Now we map the interactions with their respective use cases. Those relating to the unauthenticated user are found in Table 4.3, while those relating to the authenticated user are in Table 4.4. The account management interactions are analogous to those of the mobile app, being found in Table 4.2.

Similarly to the use cases of the mobile app, these can also be found in the Appendix A.

Designation	Use Case	Success Criteria
Login	Login UC1	User can log in
Register	User Registration UC2	User can create an account
Recover Password	Password Recovery UC3	User can recover the password

Table 4.3: Unauthenticated User Portal Use Case Mapping

Designation	Use Case	Success Criteria
View Fire Event	View Occurring Fire UC12	User can view active fires
View Fire Event Details	View Occurring Fire Detail UC13	User can view active fire details
Search by Location	Search Location UC14	User can make a query by location
Search by Date	Search Timeframe UC15	User can make a query by date

Table 4.4: Authenticated User Portal Use Case Mapping

4.2.3 Non-functional Requirements

For the non-functional requirements of the fireloc web portal, we defined the usability and latency requirements.

Usability

The portal's main goal is to provide critical information to users with diverse backgrounds and motivations. So, we must ensure that the interactions are easy and intuitive for any type of user. To achieve this, we must provide the user with a familiar layout similar to the ones he might encounter in more established web pages. This will ensure that most users have some prior knowledge of how to operate the Fireloc Portal. But, we still need to reduce the complexity of the layout as much as possible. This includes reducing the number of menus that we should introduce on the portal and the depth of any possible sub-menus. Any interaction that requires user input, such as query parameters, should already have these predefined to reduce human error. In the case of any user error, we need to give enough feedback for him to be able to correct it. And this feedback should be as simple as possible.

Latency

As with the mobile app, low latency is an essential characteristic we need to cover. This requirement rises to new importance when we think about the main focus of the portal, which is the display of critical information. This implies we must assure low latency for the majority of our requests. To define this requirement, we set the latency maximum value to 4s analogous, to the mobile app.

4.2.4 Mockups

In this section we present the mockups for the portal, and as discussed previously, we follow the single page design principle commonly found in similar products.

Landing Page

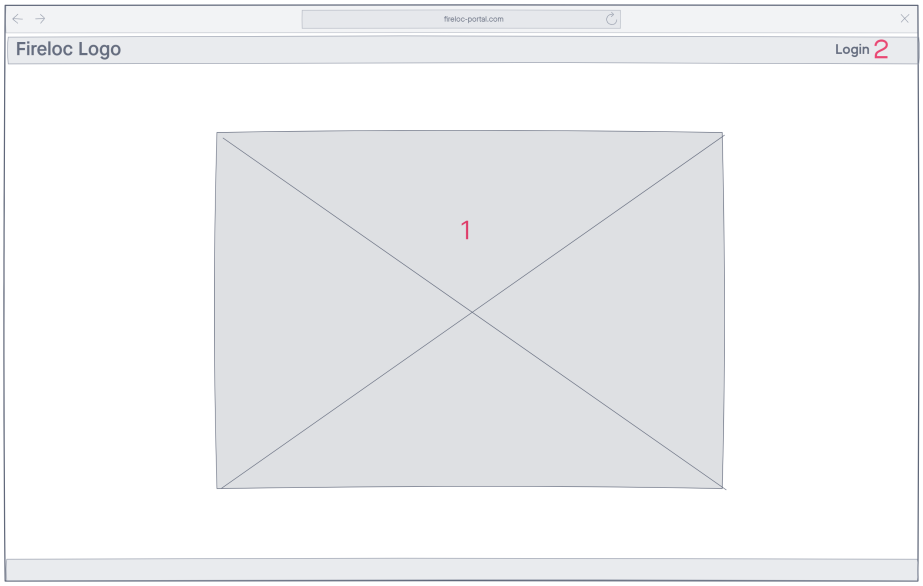

Figure 4.6: Landing Page / Unauthenticated Map Page

1. Map container: Displays the map with the active fire events
2. Login button: Clicking redirects the user to the login page

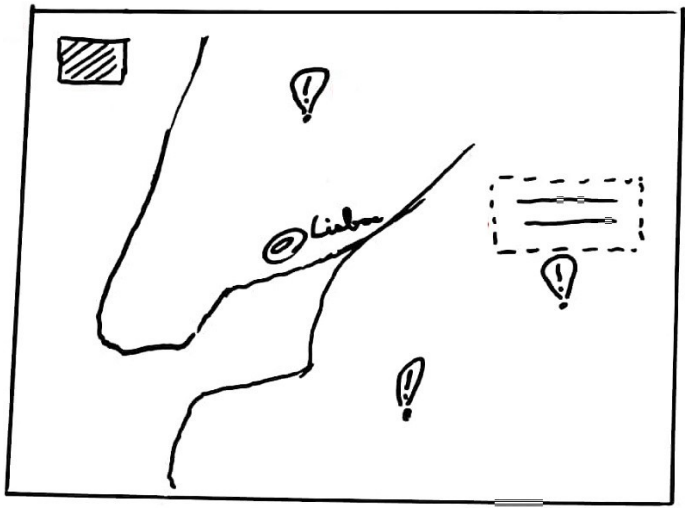

Figure 4.7: Map container

4.2.5 Main Page

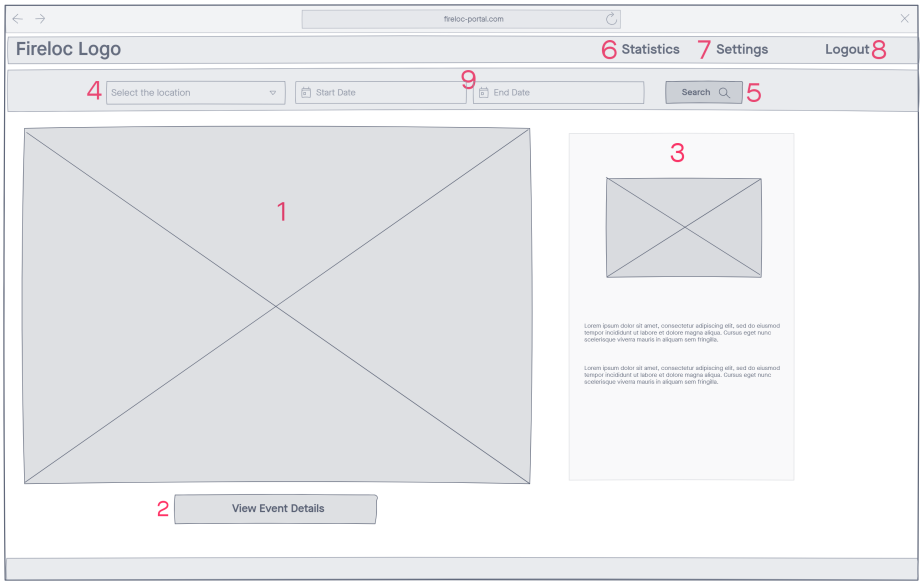

Figure 4.8: Main Page

1. Map container: Displays the map with the requested information
2. Details button: Clicking requests the information associated with the selected fire event
3. Fire Event container: Displays the information associated with a single contribution for the selected event
4. Location field: Allows the user to select a location to query about
5. Search button: Allows the user to perform queries based on the filled query fields
6. Statistics button: Redirects the user to the statistics page
7. Settings button: Redirects the user to the settings page
8. Logout button: Logs out the user
9. Date fields: Allows the user to specify a time range for the query

Login Page

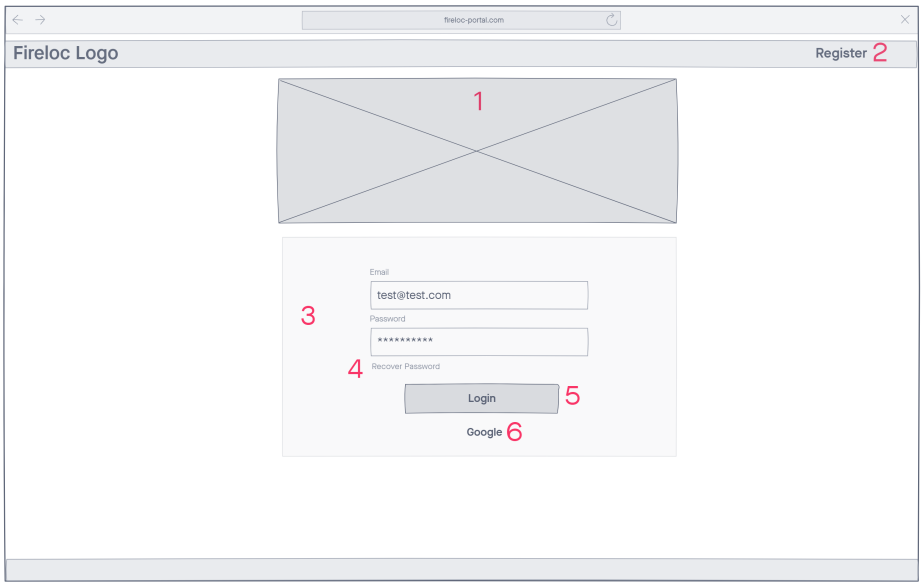

Figure 4.9: Login Page

1. Fireloc Image
2. Register button: Redirects the user to the register page
3. Login form: Contains the fields for the login process
4. Recover password button: Redirects to the recover password button
5. Login button: Submits the login form
6. Google button: Logs the user in through the google account

Register Page

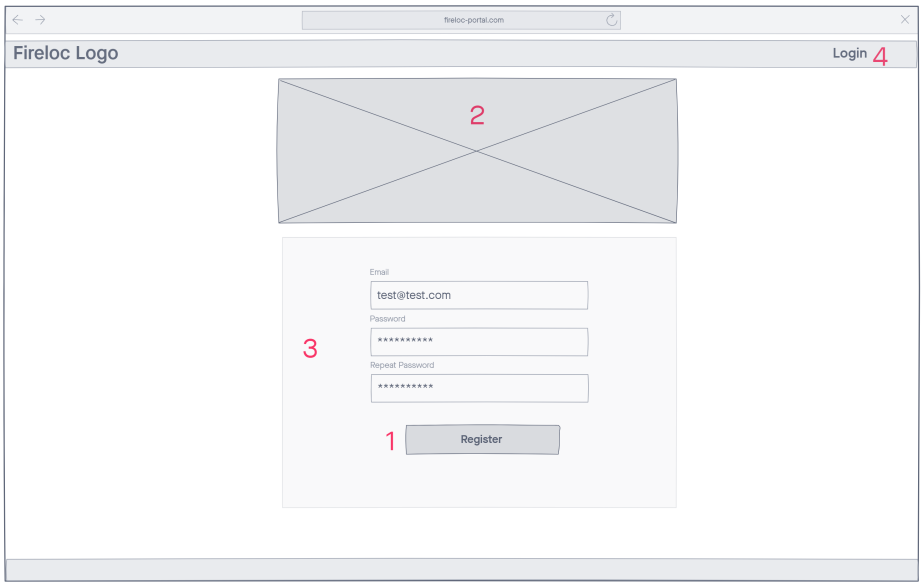

Figure 4.10: Register Page

1. Register button: Submits the register form
2. Fireloc Image
3. Login form: Contains the fields for the register process
4. Login button: Redirects to the login page

Settings Page

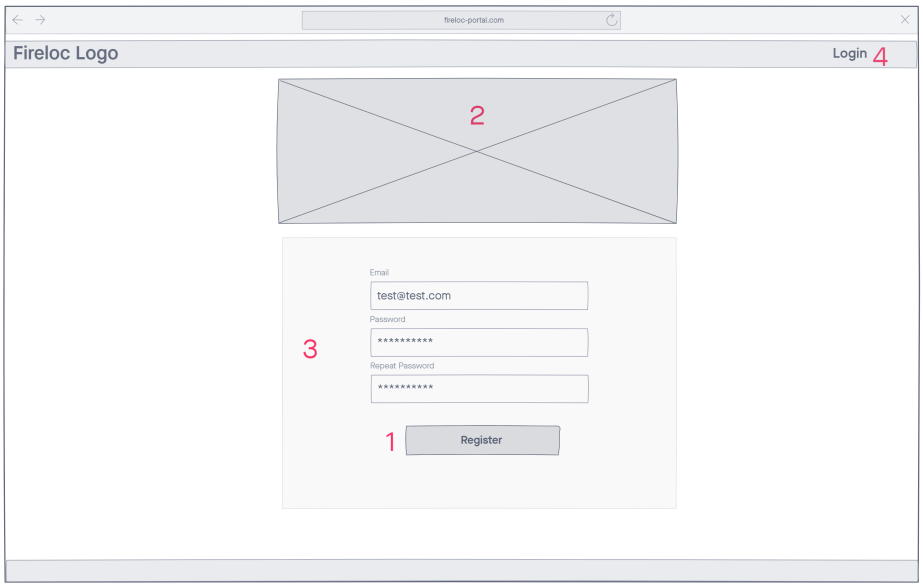

Figure 4.11: Settings Page

1. Edit profile button: Redirects the user to the edit profile page
2. Change password button: Redirects the user to the change password page
3. Change email button: Redirects the user to the change email page
4. Delete account button: Redirects the user to the delete account page
5. Statistics button: Redirects the user to the statistics page
6. Settings button: Redirects the user to the settings page
7. Logout button: Logs out the user

Statistics Page

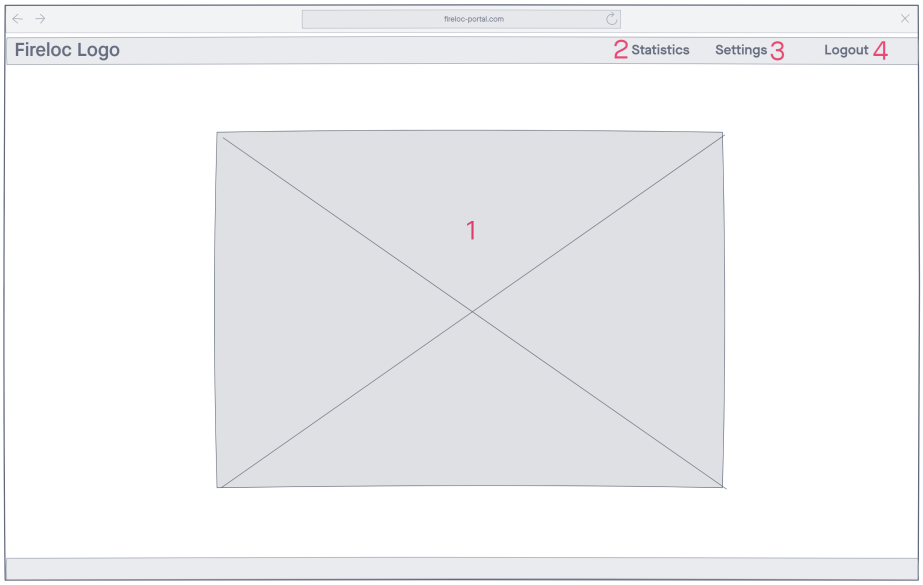

Figure 4.12: Statistics Page

1. Statistics container: Contains the graphics with the necessary information
2. Statistics button: Redirects the user to the statistics page
3. Settings button: Redirects the user to the settings page
4. Logout button: Logs out the user

4.3 Module 6 - Geoportal API Server

In this section we detail the specifications for the API Server supporting the web portal.

4.3.1 Supported Operations

The API Server must support a REST API over HTTPS that implements the following functionalities:

- fetching fire events according to a preset configuration; for instance, current time and all territory;
- searching for fire events by selected location in accordance with use case UC14;
- searching for fire events by selected date in accordance with use case UC15;
- fetching details of a single fire event in accordance with use case UC13;
- fetching User Contributions associated with a single fire event in accordance with use case UC13;

Each of the previously defined operations should return a JSON payload with the requested content to the client. The payload content must be filtered according to the data privileges of each user type. These are system admins, users attached to authorities and institutions, volunteers, and general public. The management of these privileges is not responsibility of this module.

4.3.2 Response Data

The following list breaks down the possible content for the response payload which can encompass:

- geographical position of volunteers and associated data, for instance the fire photo;
- geographical position of volunteers and associated data, for instance the fire photo, by degree of trust;
- fire event estimated position;
- fire event estimated position by degree of trust;
- probability surface for the geographical extension of the fire event;
- fire event progression projection;
- burned areas;
- risk charts with valuable infrastructure marked, such as gas posts or schools;
- statistics such as total burned area by NUT2 and NUT3, number of fires by NUT2 and NUT3;
- weight of each contribution.

4.3.3 Error Handling

Should an operation fail on the server, an error message should be sent as response with the operation and error type to the client.

4.3.4 Integration

Lastly, the server must be of easy integration with the remaining parts of the system.

4.4 Security and Privacy

In this section we defined the security requirements for both the mobile application and the web portal.

The applications must be of secure use and follow the established guidelines which apply both to Angular and Ionic applications [49].

For the mobile application and web portal this concerns the secure use of the user private data when handling authentication and, the collected data when contributing. Therefore we must ensure we keep no data beyond the duration of the authentication session and the data is sent in a secure manner to the Fireloc servers, which handle storage.

The following list contains the security requirements not included in the Angular security guidelines:

- SEC1: Data should not be persisted in memory beyond the duration of the authentication session.
- SEC2: Apart from the refresh token, no data should be persisted in the device after exiting the application.
- SEC3: Collected data should not be kept in memory beyond the duration of the contribution process.

4.5 Summary

In this chapter we presented the requirements for the mobile application, the web portal and the Geoportal API, which included the discussion of the functional and non-functional requirements.

The functional requirements presented were the use cases for the mobile app and web portal, which include the authentication, the collection of data and data visualization and their success criteria. We also presented the security requirements for the each of the these system components.

For the web portal, we also presented the mockups describing the position of the UI components.

Finally we present the non-functional requirements, which are the usability and latency and their evaluation methods and success values.

Chapter 5

Architecture

In this chapter we discuss the architecture of the Fireloc System. We start by discussing the context surrounding the Fireloc System. After, we characterize the identified high level building blocks. Then, we discuss how each high level block is constituted in its elementary components. This structure follows the first three Cs of the C4 standard model [5], whose key concepts are Context, Containers, Components; however the fourth C (Code) is defined in a later chapter.

Following the characterization of the C4 model, we present the System Sequence Diagrams (SSD) to view in more detail the operations carried by the system.

Lastly, this architecture is designed to be modular to satisfy scaling constraints.

5.1 Context

The architectural context surrounding the relevant parts of system is depicted in Figure 5.1.

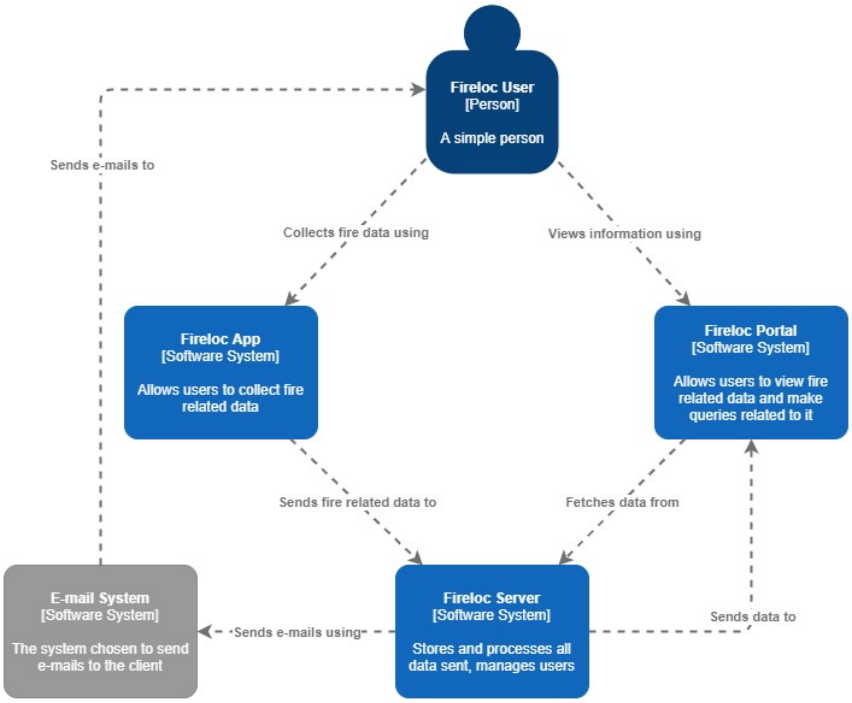

Figure 5.1: Context Diagram

As may be observed in Figure 5.1, first we have our User, who can be any registered or non-registered person who accesses our system. He has two main drivers. The first is to send data related to a fire, for that he will use the mobile app (Fireloc App) to collect and send it. The second being to view the fires in Portugal (past or present) and how his contributions may have impacted the detection of fires.

Next we have the mobile app (Fireloc App), this will interact with the User to collect the data and send it through the exposed API to the Fireloc Server.

Next we have the web portal (Fireloc Portal). This interacts with the User to provide data according to the permissions granted to the type of User he is. This data is fetched from the Fireloc Server through another API.

Finally, we have the Fireloc Server. This element is the junction of all the developed and to-be-developed functionalities of the Fireloc Project. This server will expose several APIs which will be accessed through both the Fireloc App and the Fireloc Portal as well as provide other functionalities.

5.2 Containers

In this section we describe the containers identified for the system.

The containers are the higher level building blocks of a system. We identified nine in our system. Seven of them are in the Fireloc Server. Six of them are designated as "Module X", while the other is the database. These are:

- Module 1: User Management - provides account and authentication management;
- Module 2: Event Location Assessment - verification of the user-provided position and data;

- Module 3: Geographical Data API - provides support for the mobile app;
- Module 4: Image - provides image analysis;
- Module 5: User Provided Text - provides text analysis;
- Module 6: Geoportal API - provides support for the web portal;
- Database - used to store all the system data.

The remaining two containers are:

- Fireloc App - used to collect relevant fire data;
- Fireloc Portal - used to visualize fire related data.

In the subsequent presentation we restrict our analysis to the Fireloc App, the Fireloc Portal and the Geoportal API (module 6) containers, since only these three were implemented in the scope of this dissertation. In the diagrams of Figure 5.2 we will also include the containers that interact with these three in order to identify all interactions.

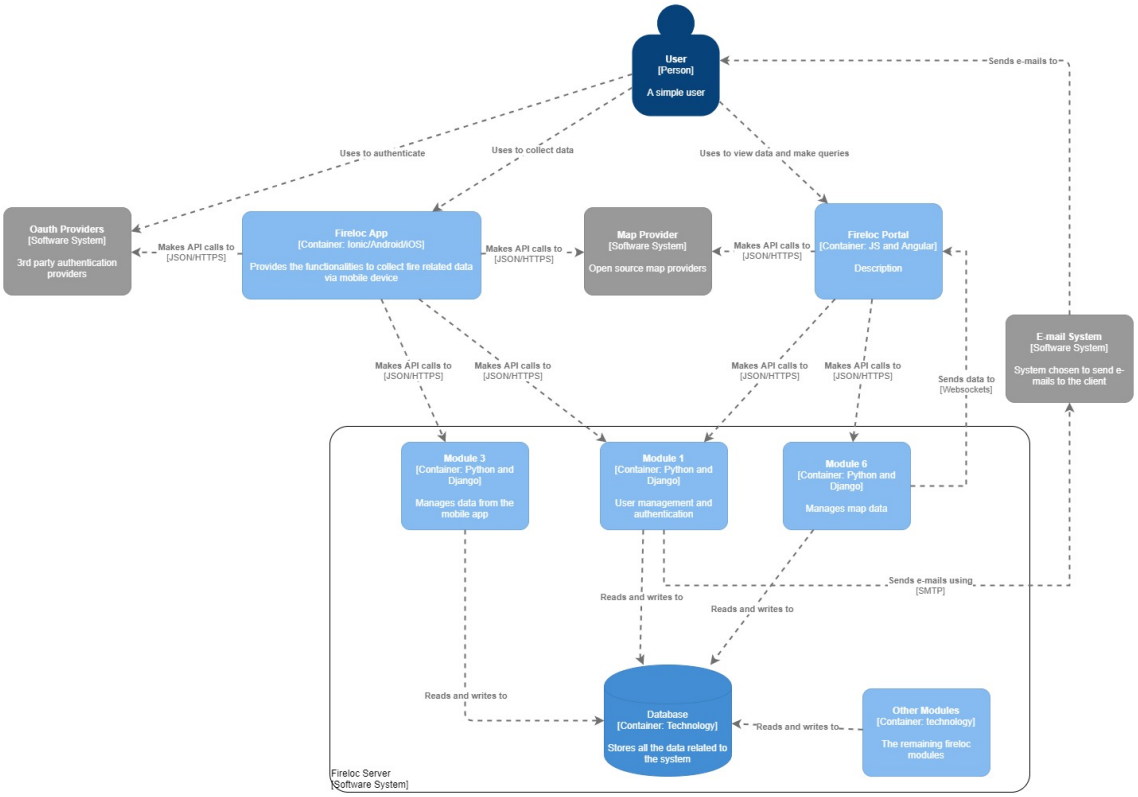

Figure 5.2: Containers Diagram

The main containers of the system are six, as depicted in Figure 5.2. They are:

- Fireloc App container. This is the mobile application. This app provides an interface that allows to collect data for reporting fires, viewing his contributions and managing his profile. It will communicate through JSON/HTTPS with the User Management container (Module 1), and the Geographical Data API container (Module 3).

- Fireloc Portal container. This is a web application. This application provides the interface for the different types of users to view fire related data as well as managing their accounts. It will communicate through JSON/HTTPS with the User Management container (Module 1) and the Geoportal API container (Module 6). The communication with Module 6 is both through REST and Websockets.
- User Management container (Module 1). This provides the functionalities for user management. This module exposes a REST API, whose endpoints the app and the portal container will connect to. It also reads and writes to a database.
- Geographical Data API container (Module 3). This manages the geospatial and image data for the fire report sent from the mobile app during the report process. This module exposes a REST API, whose endpoints the app container will connect to. It also reads and writes to a database.
- Geoportal API container (Module 6). This provides the necessary data requested by the portal. This module exposes a REST API, whose endpoints the portal container will connect to. It also reads and writes to a database.
- Database container. This stores all system data. Also, the remaining server-side containers read and write data to this container.

5.3 Components

Now we describe the components for Fireloc App, Fireloc Portal and Geoportal API containers.

5.3.1 Fireloc App

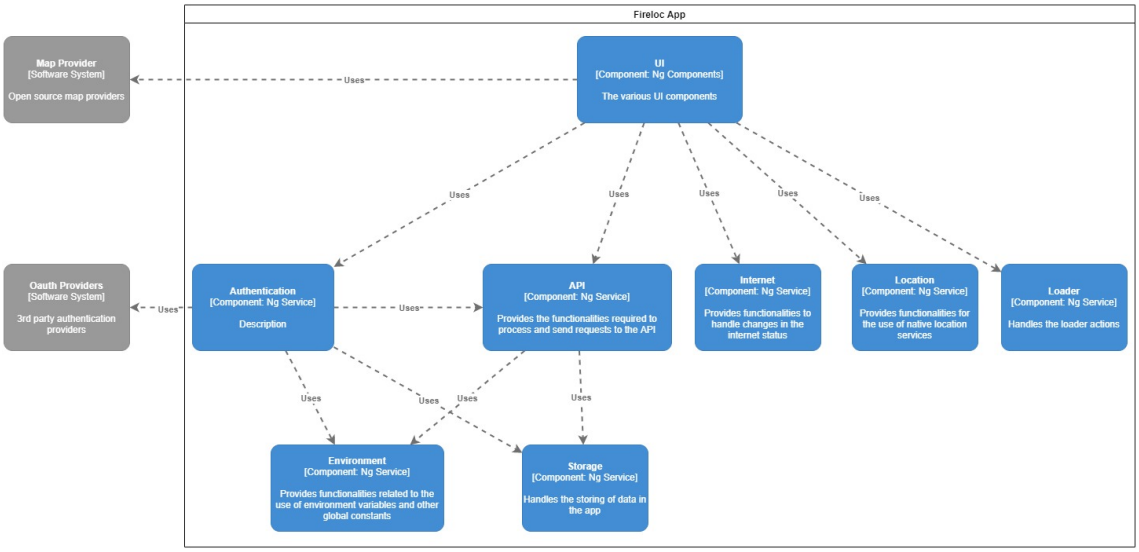

Figure 5.3: Fireloc App Components Diagram

The components of the Fireloc App container are presented in Figure 5.3. For the sake of clarity we encompassed the UI components in a single one to reduce visual cluttering.

Excluding the UI components, we have the following ones:

- Authentication: Handles the logic of the various forms of authentication defined in the in our use cases in Appendix A.
- API: Provides the functionalities required to process data and send requests to the API endpoints via JSON/HTTPS.
- Internet: Provides functionalities to handle changes in the internet status, such as alerting the user when he leaves internet cover.
- Location: Provides functionalities for the use of native geolocation services, measuring the location of the user during the contribution process.
- Loader: Handles the creation of loading UI actions.
- Storage: Handles the persistence of data during sessions.
- Environment: Provides functionalities related to management and use of environment variables and other global constants and variables that might be required by the various services and components.

5.3.2 Fireloc Portal

Comparing the services from the mobile application with the web portal, we verify that several of them are common. So, since we selected the Ionic framework these can be shared even though they are compiled to different platforms – mobile and web.

The architecture of the Fireloc Portal is presented in Figure 5.4. It has the same underlying structure of the mobile app, except for the removal of the Location service, which is not necessary for the web portal, and the addition of the Websocket Controller to handle live updates coming from the server.

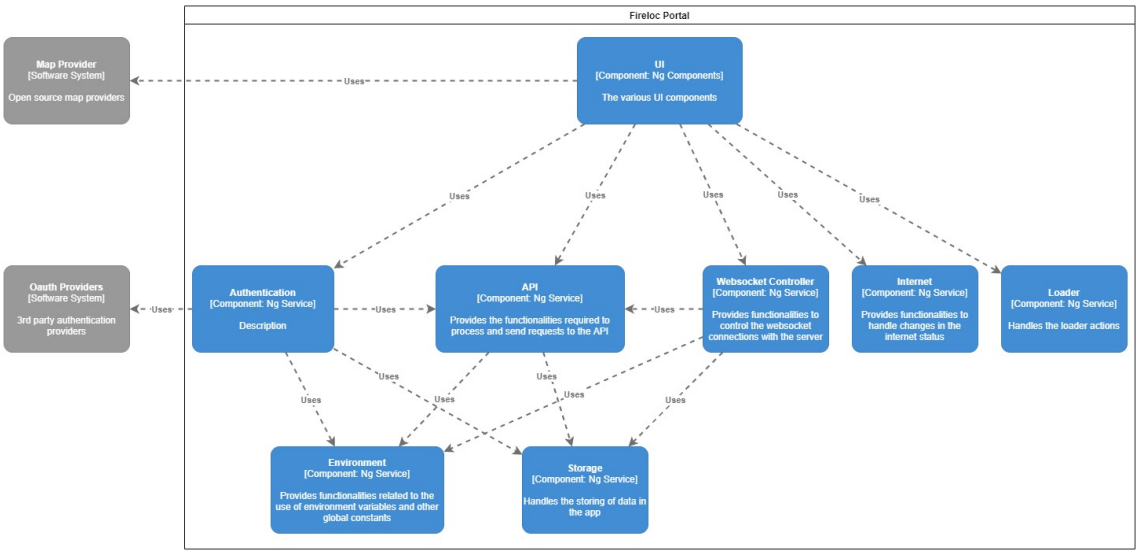

Figure 5.4: Fireloc Portal Components Diagram

5.3.3 Module 6 - Geoportal API Server

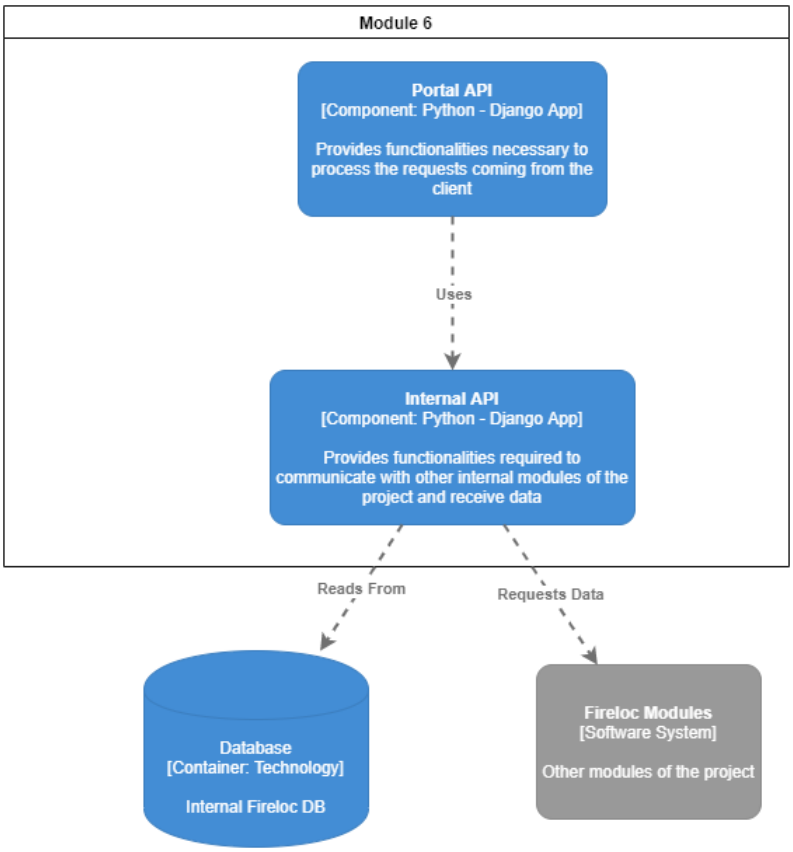

Figure 5.5: Geoportal API Server Components Diagram

The diagram depicted in Figure 5.5 represents the components of our last container, the Geoportal API Server.

In the upper level, we have the Portal API component. This is responsible for the logic necessary to handle incoming requests from authenticated clients. Below it, we have the Internal API component, which is a communication layer to isolate the Portal API from the remaining modules in the system. This API is responsible for requesting the data needed by the Portal API to the database or the different modules of the system.

The decision to divide this module into three components aims to comply with three guiding aspects. First, an external API to serve web clients (Portal API), isolating the other modules of the system. Secondly, to comply with the modular restrictions of our requirements, this API should not interact directly with the remaining internal modules, thus the need to define an internal API housed in its respective component. Lastly, the RESTful architectural logic doesn't directly apply to pushing data from a server to a client, thus we separate those functionalities and place them in the Websocket Handler component.

5.4 Sequence Diagrams

In this section we analyse the main interactions of the users with the system and the associated events through System Sequence Diagrams. While ideally each use case should

have one SSD for the main success scenario, our system has many similar sequence flows. This allows using a single diagram to cover multiple operations. This applies both for mobile app and web portal operations.

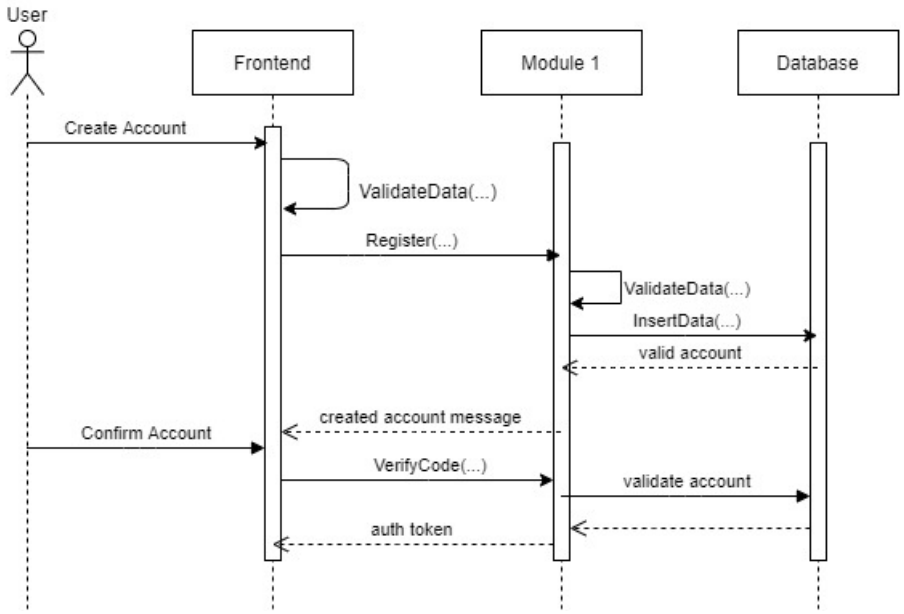

Figure 5.6: Create Account Success Sequence Diagram

The sequence diagram depicted in Figure 5.6 shows the interaction sequence for creating an account. This applies to a registration through both the mobile app and the web portal platforms. In this process, the user inserts the data for his account through our frontend that is sent to the User Management module (Module 1). This module parses and validates the data. Once the data has been validated, it is matched against the existing users in our database. If none exists, the user is inserted into the database. A message is then sent to the frontend, allowing the user to proceed. Then, the user must confirm the account creation, resulting in a code being sent to Module 1. If the code is valid, the account is validated and an authentication token is sent to the client application.

When there is a failure in the process of creating an account, this failure occurs only when verifying the data against the database, since the data is only submitted to the backend server if it passes the validation in the frontend. Therefore, if an error occurs in the database, an error message is sent to the client as shown in Figure 5.7.

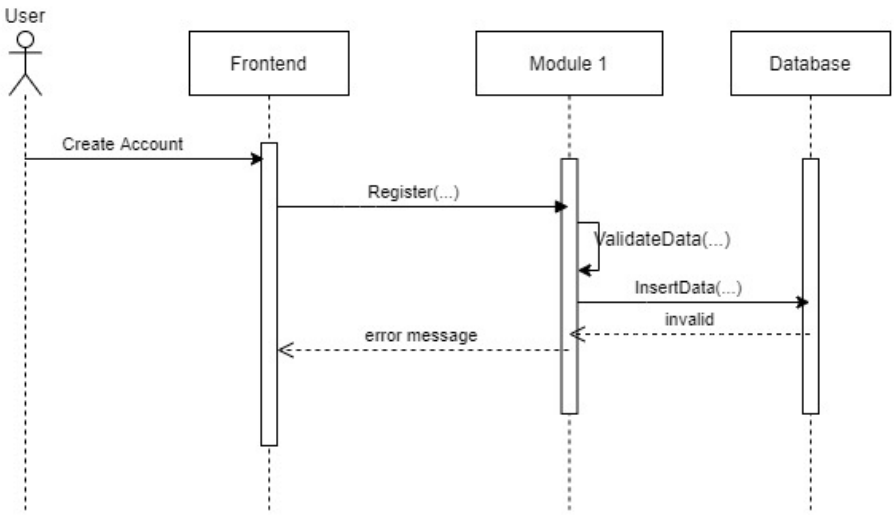

Figure 5.7: Create Account Failure Sequence Diagram

The sequence diagram of Figure 5.8 shows the login sequence when the client authenticates through our authentication service. First, the user sends his credentials through the client application. Next, the credentials are parsed and validated against the database. Once the credentials have been corroborated, an authentication token is generated and sent to establish a session on the client.

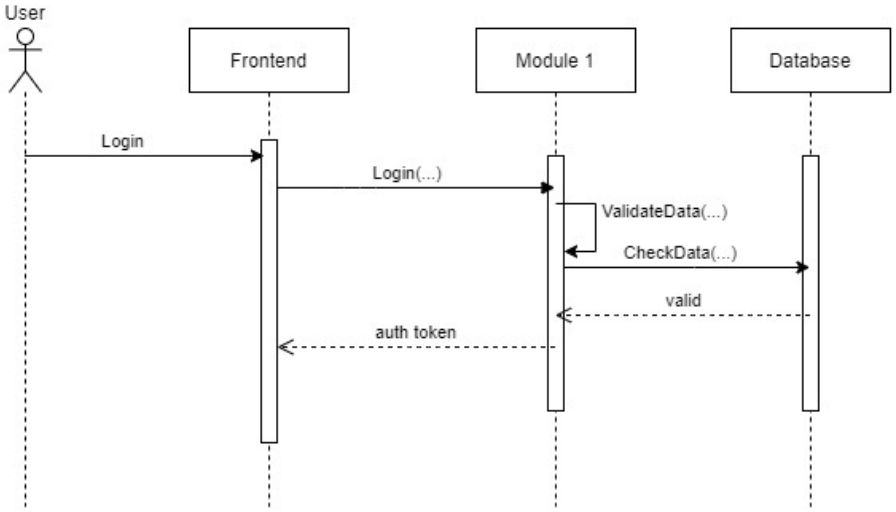

Figure 5.8: Login Sequence Diagram

Figure 5.9 depicts the case of a failure in authenticating the credentials, where an error message is sent to the client.

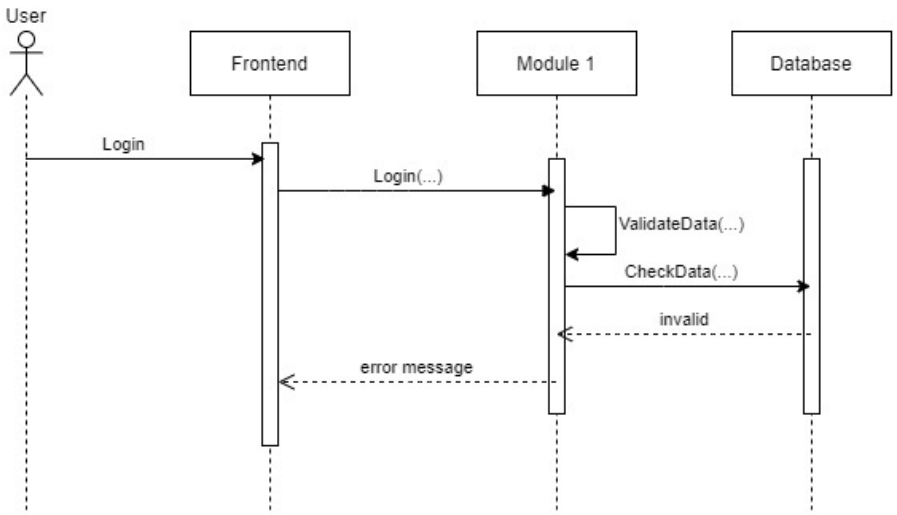

Figure 5.9: Login Failure Sequence Diagram

In Figure 5.10, we have the sequence diagram for authentication through a 3rd party provider[1]. In this case, the user's credentials will be provided through a trusted service in which the user already has an account. The service will then send a token to our frontend, which, in turn, will be sent to Module 1. This module will validate the provided token and initiate a session for the client. In Figure 5.11, we present an example of this process for the Firebase provider.

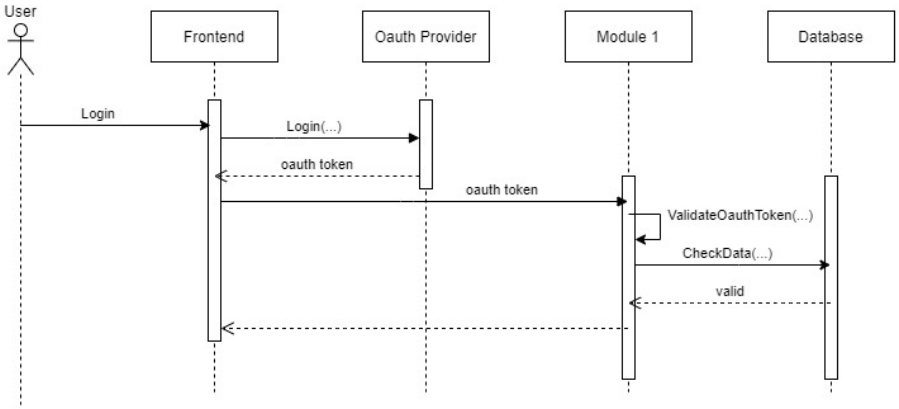

Figure 5.10: 3rd Party Login Sequence Diagram

[1]Extension 1.a of use case UC1

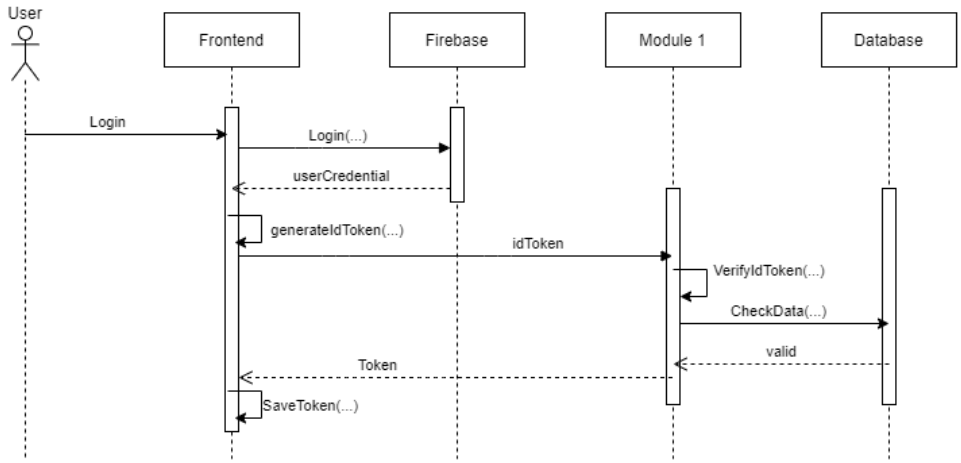

Figure 5.11: Firebase authentication sequence

The sequence diagram of Figure 5.12 shows the process of submitting a contribution. Here, the data is sent from our frontend application (mobile app) to the Geographical Data API (Module 3), which will parse and validate the data sent. Once this is completed, the data is added to our database for posterior analysis. This sequence also applies for changes in settings such as passwords, or e-mails. The only change is the data from the frontend is being sent to Module 1 instead of Module 3.

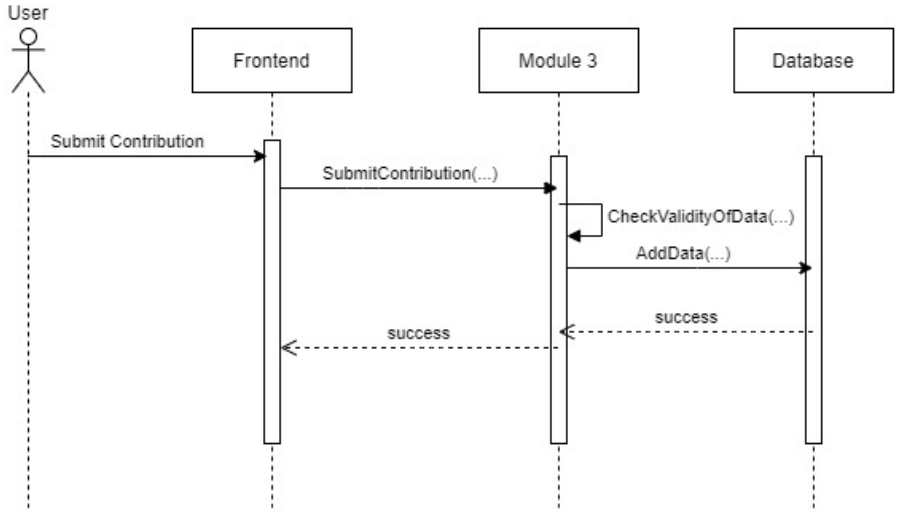

Figure 5.12: Contribution Sequence Diagram

In Figure 5.13, we show the sequence for viewing a contribution made on the mobile app. First, the client requests the contributions associated with a determined user from Module 3, which in turn fetches the data from the database, sending it to the client in response. The user will then select one of the contributions from the list sent, and the client will request the details of that contribution from Module 3. Module 3 will fetch the details from the database, and send it to the client in response.

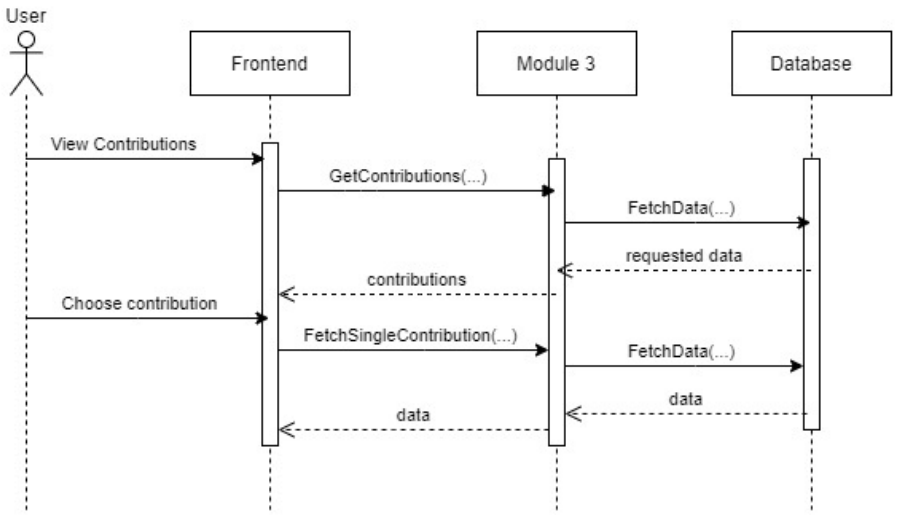

Figure 5.13: View Contribution Sequence Diagram

The sequence diagram of Figure 5.14 depicts a request from the portal. This sequence diagram is applicable to all queries made by the portal, since they all follow the same pattern. In it, the user uses our frontend portal to make a query, such as, search by date, place or path, with the parameters sent to the Portal API. The request parameters are then parsed by the Portal API, which is responsible for handling the communications with the portal. Once the query parameters have been parsed, the data is requested through the Internal API. This API will determine whether to fetch the data from our database or request it from the remaining Fireloc modules. Once the Internal API has the data, it is returned to the Portal API, which in turn sends it in response to the client.

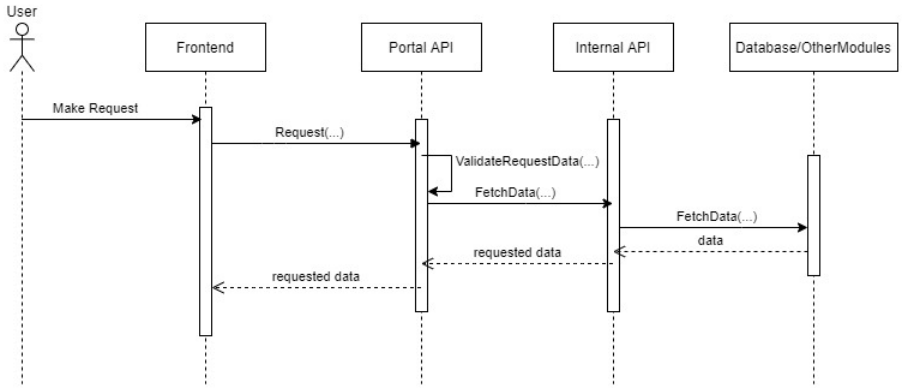

Figure 5.14: Portal Data Request Diagram

5.5 Summary

In this chapter we presented the architecture of the system. The main components of the system defined were the mobile application and the web portal, which operate in a client-server relationship with the Fireloc server, where the Geoportal API will be hosted.

The architecture of the mobile application and the web portal have many elements in common, due to the underlying Angular framework.

Lastly, we described the SSDs that define the flow of data of the various system operations,

including the fetching and sending data to the Fireloc server, and the authentication flows to be implemented.

Chapter 6

Implementation

In this chapter we present how the proposed components — the mobile app, the web portal and the API — were implemented.

For the mobile app and the web portal, we discuss the implementation of their inner mechanics, namely, state management, component comunication, authentication, security, and user management. The adoption of a similar structure for both components was possible by the choice of the Ionic framework, which allows the use of the same technology, Angular, in both. This has many advantages, such as, common services, or the testing framework and methodology, which allows reducing the development time as well as increasing the robustness of our implementation.

Lastly we discuss the API endpoints created to suport the web portal.

Before discussing the implementation, it's important to clarify certain terms used in this chapter. First, if no context is provided, the term 'event' refers to a fire event a user is witnessing or that has already occurred and wants to obtain details about it. Second, when presenting the integration with NgRx, state can be used to reference either the entirety of our application state, or just slices of it. When we reference state, if not explicitly defined, the reader is to assume we are referencing the slice associated with the context of the section. For example, when describing authentication, state references the authentication state, or when referencing settings, the local settings state. Another important term is 'main action'. This refers to an action that is the starting point of a chain of actions, for example, when discussing the login process, the main action is the action dispatched when we press 'login'. Lastly, the use of <T> denotes a generic type T.

6.1 Implementation of the Mobile App

In this section, we describe the implementation of the mobile app according to the architecture defined in section 5.3.1. We grouped the functionalities into several categories: authentication, contribution process, viewing contributions and user management. Each category was implemented as an Ionic page. A page in Ionic is a simple Angular component wrapped into its own Angular module with a routing module attached. However, since there is no tangible difference between a page component and a generic component, we sometimes use the term component to refer a page in the context of layout or in its inner functionality that does not affect the organization of our code.

6.1.1 Authentication

Since the fire information should be reliable, the requirements define that only authenticated users may contribute with reports. So, we now detail the authentication process and its implementation.

All functionalities related to authentication are encompassed in an Angular module.

Authentication Process

The process of authentication is as follows:

1. The user inserts his credentials,
2. The credentials are sent to the server through the Fireloc API,
3. The server validates the credentials and generates a token,
4. The server returns the token to the client,
5. The client saves the token to be sent with future requests

This process can be observed in the login sequence (Figure 5.8), which is according with the use case UC1.

State management

The first step is the definition of the authentication state, whose interface is presented in Listing 6.1. This stores all authentication data, while the application is running. The authentication state includes the authentication status, the access and refresh tokens, the expiration date of the access token, and an error flag. The refresh token is persisted in memory.

```
interface AuthState {
  isAuthenticated: boolean;
  accessToken: string;
  refreshToken: string;
  authFailure: boolean;
  expiresIn: number;
}
```

Listing 6.1: Authentication State Interface

Access to the state properties is made through selectors derived from the AuthState feature selector.

Authenticating a user

For a user to be authenticated, he must simply open the application and insert his credentials in the correct fields. Once he submits the credentials, a NgRx action is dispatched to

the store with the credentials, triggering an effect. The effect will call the authentication service, which will make a http request to the appropriate API endpoint to authenticate the user. Should the credentials be correct, we are returned a pair of tokens (access and refresh), which are placed in the store via a reducer function.

Hydration

Once we open the application, an action is dispatched to the store, triggering the hydration effect. This effect will check the device's storage for a refresh token and call the appropriate API endpoint to refresh the tokens. This allows for any user who has not logged out of the application, to maintain the session without being forced to constantly reauthenticate himself each time he reopens the app.

6.1.2 Security

Security is a critical concern to be addressed in the application. It revolves around sending, fetching and displaying information. Since we use Angular as the base framework of the Ionic app, the security concerns and methodology apply both to the mobile app and the web portal. Our implementation follows the Angular recommended security procedures. These are developed in section 6.2.2.

6.1.3 User Contribution

Contributing is the core action of the mobile app. It is a multi-step process, each with its own challenges. These can be divided in mandatory and non-mandatory steps. The mandatory steps are three. First, the collection of the user's position. Second, the device's orientation and lastly, a photo of the fire. The non-mandatory steps can vary in relation to the user's position or orientation, but the collection of the data remains the same and so does its implementation. For a full view of the contribution process and all steps, as well as the possible interactions, the reader can consult the BPMN2 diagrams of Appendix B.

Before Contributing

The contribution process as is detailed in use case UC7, requires that the user has to enable his device location services, which in turn allows us to collect the user's location. So, we must ensure the user is aware if the location service is enabled or not in the mobile settings. This is a twofold task and is spread over several parts of the execution of the app. To ensure the user is aware of the dependency on the device location service, each time the user opens the app, from a cold start, we check the system permissions and ask for the user to grant access to the location services (if it is has not yet been granted) and to turn on the location. This is done through the use of the **android-permissions** available through **ionic-native**. The snippets of Listings 6.2 and 6.3 illustrates the use of this API.

```
this.androidPermissions.checkPermission(
  this.androidPermissions.PERMISSION.ACCESS_COARSE_LOCATION
).then(
// handler function
);
```

Listing 6.2: Verify Location Access Permission

```
this.androidPermissions.requestPermission(
this.androidPermissions.PERMISSION.ACCESS_COARSE_LOCATION
).then(() => {
  this.locationAccuracy.request(
    this.locationAccuracy.REQUEST_PRIORITY_HIGH_ACCURACY
  ).then(
    // handler function
  )},
  error => {
    // error handler function
  }
```

Listing 6.3: Request Location Access Permission

But, the user may decide not to enable the location services, or he may minimize the app and disable the location services, and return to the app. These actions need to be accounted for and the behavior of the app should reflect them. The solution to these limitations was to implement an Angular guard to verify the status of the location access. An Angular guard or route guard is an Angular interface that allows controlling the access to a route based on a condition provided in its implementation. Ionic exposes all five Angular guard interfaces: **canActivate**, **canActivateChild**, **canDeactivate**, **resolve** and **canLoad**. We use the **canDeactivate** interface. This guard decides if a route can be deactived — if we can navigate out of it. In the implementation of the **canDeactivate** interface, we use the locationEnabledFlag property of the menu options feature store. This flag holds the on/off value of the device location service. The guard verifies the following two conditions: fist, if the url we are navigating to is the contribution url and the device location service is off, we remain in the same page (the menu page) and display an alert; second, if the location service is on, or if we are navigating to a different url, we allow the navigation out of the menu page.

To detect when the user changes the location to on or off, we use the **diagnostic** plugin from ionic-native to register a listener to the diagnostic state as depicted in the snippet of Listing 6.4. This step occurs whenever the app starts. As presented in Listing 6.4, when the location mode changes in the diagnostic state, we emit an NgRx action to the store, which in turn triggers the menu reducer and update the menu options feature state.

```
this.diagnostic.registerLocationStateChangeHandler(state => {
  if (state === this.diagnostic.locationMode.LOCATION_OFF) {
    this.store.dispatch(AppActions.locationIsDisabled());
  } else if (state === this.diagnostic.locationMode.HIGH_ACCURACY) {
    this.store.dispatch(AppActions.locationIsEnabled());
  }
});
```

Listing 6.4: Register Diagnostic Listener

State management

For the contribution process, we set up a feature store divided into three parts each with its own reducer functions as shown in Listing 6.5.

```
interface ContributionState {
  userContribution: UserContributionState;
  backgroundLocation: BackgroundLocationState;
  contributionOptions: ContributionOptionsState;
}
```

Listing 6.5: Contribution State Interface

UserContributionState stores the current contribution values, which the user actively collected. **BackgroundLocationState** stores the location measurements collected in the background. Since the background locations is a collection of locations, we store it using EntityState<T>. This allows us to easily manage the measurements. Lastly, we have **ContributionOptionsState** that stores the various flags and options needed throughout the contribution process.

The store is updated throughout the contribution process and is reset once the contribution is submitted or the user cancels the contribution.

Collecting Location Data

Collecting the user's location data is the most important step in the whole process. Part of the location data is provided by the user in two distinct moments along the contribution process, and another part is collected in the background by the application.

Each time a user starts the contribution process, that is, when he enters the contribution page, an action is dispatched to the store. This action triggers an effect, which calls the location service to start measuring the location in the background. This action is dispatched through a resolver — Background Location Resolver. A resolver is an Angular service that implements the **Resolve<T>** interface by overriding the **resolve()** method. This method is invoked when the navigation to a route starts and awaits for a value before activating the route. In our case, we use this period of time to dispatch the action.

```
resolve(route: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): Observable<any> {
    this.store.dispatch(AppActions.calculateAutoInitialLocation());

      return this.store
        .pipe(
          select(selectCanAdvanceFromMandatoryPositionResolver),
          filter((hasValue: boolean) => hasValue),
          tap(() => {
            this.store.dispatch(
            ContributeActions.advancedFromMandatoryPositionResolver()
            )
          }),
          first()
        );
}
```

Listing 6.6: Background Location Resolver

In Listing 6.6, we present the Background Location Resolver. The **resolve()** function is used to dispatch the action to populate the store with the first set of user coordinates, which is utilized to instantiate the map component. Meanwhile, the resolver is blocked by the **filter()** operator, which checks the value of a flag in the store. When the store has the required coordinates, it sets a flag indicating to advance the process. Once this flag is set, the resolver is unblocked in the **filter()** operator. Then the resolver dispatches a new action that will call a service method that starts the background location collection. After, it emits the first value of the stream, finalizing the observable and exiting the resolver.

Once the service method is called, as long as the user is contributing, we dispatch to the store a new position consisting of the longitude and latitude values of the device at that exact moment, at fixed intervals. This position is then added to **BackgroundLocationState** through **adapter.addOne()**, which is one of the CRUD operators from NgRx Entity.

Collecting the user data is done on the verify-location component. This component displays a map instantiated with the latest location fetched before activating the component.

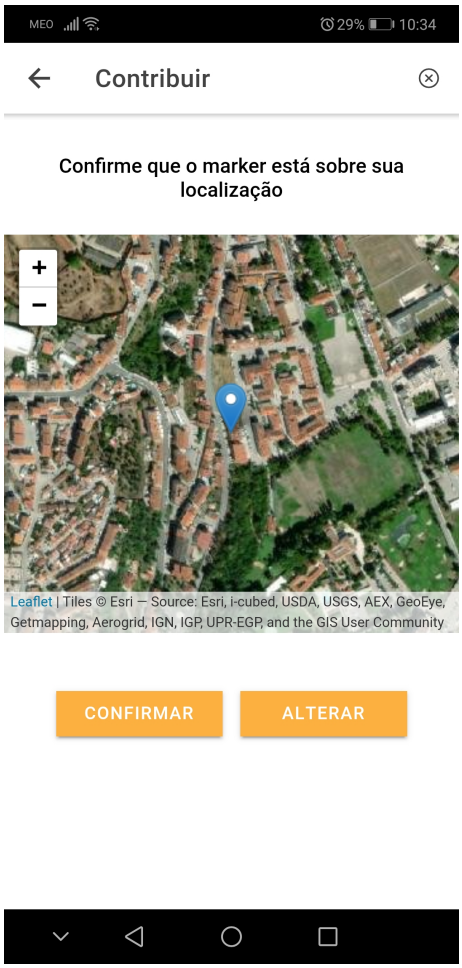

Figure 6.1: Verify Position Component

In Figure 6.1 we show the verify-location component. Here, the user verifies if the location being displayed matches his real location and confirms it. The base map is provided by ArcGIS service [3] and shows real satellite data. The user can explore the surrounding area by zooming in or out and moving around the map to verify his position.

The user may realize the position is incorrect, which could be caused by the presence of bodies of water or a change in the GPS satellites. In this situation, he can change the location by pressing 'Alterar'. Once he has clicked 'Alterar', any change in position of the map will move the marker to the center of the map, displaying the new position of the user. By clicking 'Reset' the map will automatically center on the initial position. If the position is correct, the user presses 'Confirmar', dispatching a new action to the store with the position of the user.

Collecting Orientation Data

After collecting the user's position the system has to acquire his orientation in relation to the magnetic north, which is accomplished by collecting the device's orientation.

Each time the user is performing a contribution, he has to calibrate the device. We indicate this condition when the user enters the component through an alert message asking the user to move the device around, allowing the device to self-calibrate.

Once the compass is calibrated, the application can collect the orientation measurement. This requires that the device be in the horizontal, pointing towards the fire. Then the user can press 'Confirmar Orientação' to submit the measurement. To help in this submission, the application display several values on the screen to inform the user. First is the orientation value in relation to the magnetic north in degrees, and the others are 'Ângulo' and 'Inclinação', which are the rotation according to the X and Y axis of the device. This screen can be observed in Figure 6.2.

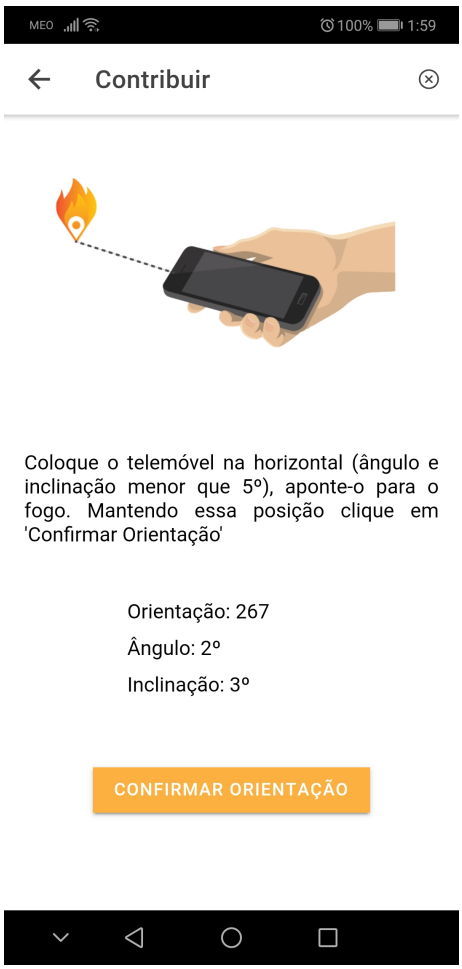

Figure 6.2: Verify Orientation Component

To obtain these values we attach a listener to the orientation event provided by the Motion capacitor plugin, which in turn returns the values of the rotation of the device according to the X, Y. Considering any rotation along these axes may have an impact on the overall orientation values, the user cannot submit the orientation unless the angles have a value inferior to a predefined threshold of 5 degrees, which will trigger a warning message. The orientation value is obtained from the DeviceOrientation plugin from @ionic-native/device-orientation/ngx.

Once all conditions are met and the user presses 'Confirmar Orientação', an action is dispatched to the store with the values of the orientation.

Collecting Photographical Data

The last step needed to submit a contribution is taking a photo of the fire the user is reporting. Taking the photo is done through the Camera capacitor plugin. The plugin calls the device's camera app where the user takes the photo of the fire. Once the user takes the photo, and confirms it is the photo he wants to submit, an action is dispatched to the store with the base64 representation of the photo. The photo is also saved on the user's gallery.

It is important to reference two important facts. First, the user cannot import a photo from his gallery, this is to force him to take a fresh photo. Second, the photo cannot be edited while in the app so as not to alter any important aspect that the backend server may need to correctly process the image once the user submits his contribution. All these aspects are done based on the settings with which we call the camera plugin as seen in the snippet of Listing 6.7.

```
takePhoto() {
  Camera.getPhoto({
    quality: 95,
    allowEditing: false,
    resultType: CameraResultType.Base64,
    saveToGallery: true,
    source: CameraSource.Camera,
  }).then(value => this.confirmPhoto(value.base64String));
}
```

Listing 6.7: Camera plugin use

Cancelling the Contribution

Cancelling the contribution is a necessary part of our process. To this effect we need to take into account the status of the contribution, mainly if the user has already done the mandatory steps or not. If a user has concluded the mandatory steps, we have sufficient data to be able to contribute.

To this effect whenever a user clicks the cancel button, a **cancel** action is dispatched to the store. This action triggers an effect that will determine if the mandatory steps are concluded or not. This is done through selecting the last mandatory parameter in the contribution, the photo. If there is a photo in the store, the application opens an alert, asking whether the user wishes to cancel or submit the data already collected (the mandatory fields and all optional data until that moment), as depicted in Figure 6.3.

If the user declines, an action is dispatched to the store to clean up the contribution store and stop the collection of the background location. If the user agrees, the action **contributeOnCancel** is dispatched to submit the data to the backend server.

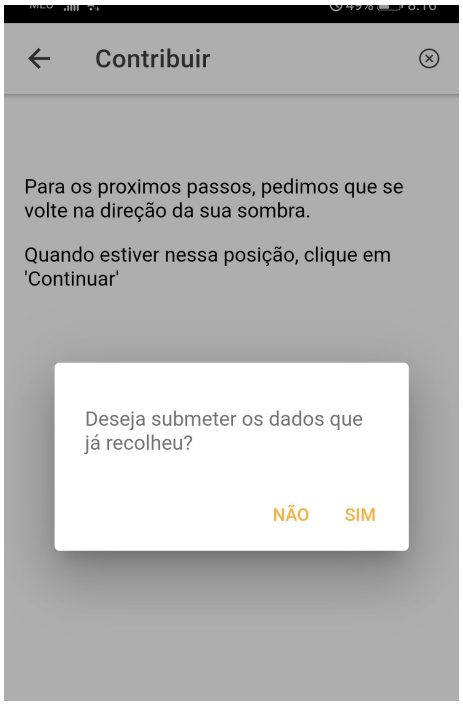

Figure 6.3: Cancel Dialogue

Submitting the Contribution

A contribution can be submitted at three points: once the mandatory fields are completed (**partialContribute** action), or if the user cancels during the collection of the optional data (**contributeOnCancel** action), or when he finishes the full contribution process (**fullContribute** action — mandatory + optional fields).

The process of submitting a contribution is divided in three effects. The first effect is to extract the data collected by the user, while the second to read the data collected in the background, and the last one submits all the data to the backend server. This division in three effects aims to reduce the complexity of the observable stream.

In Listing 6.8 we present the first effect. This effect will select the contribution data collected by the user from the store, and pass it as the props of the action (**contributeSelectResult**) that triggers the next effect. The **ofType()** operator in the effect is used to select the actions **contributeOnCancel**, **partialContribute** and **fullContribute** that signal the start of the submission process.

But how is the data selected by the effect? The effect accesses the store and passes the data to the observable stream, which is accomplished by the **withLatestFrom()** operator. This operator joins two observables: the action observable filtered by the **ofType()** operator and the observable from the store selection. So, this effect's observable stream will now contain two observables, but only the store select observable will be used. To reduce the number of observables in the stream, we dispatch the action **contributeSelectResult({data: selectResult})** with the selected data as props, allowing the next effect to start with only one observable.

```
contribute$ = createEffect(() => {
  return this.actions$
    .pipe(
      ofType(ContributeActions.contributeOnCancel,
             ContributeActions.partialContribute,
             ContributeActions.fullContribute),
      withLatestFrom(
        this.store.pipe(
          select(selectContribution)
        )
      ),
      map(([_, selectResult]) => ContributeActions.contributeSelectResult({
        data: selectResult}
      ))
    );
});
```

Listing 6.8: Selecting Contribution Data

The second effect has a structure very similar to the previous one, so we omit the discussion of its details.

Finally, the final effect is depicted in Listing 6.9. It makes a call to the Fireloc API with the contribution data as payload. As we can see in the snippet below, we are handling only one observable as described in the discussion of the first effect.

```
dispatchContribution$ = createEffect(() => {
  return this.actions$
    .pipe(
      ofType(ContributeActions.dispatchContribution),
      tap(() => this.loader.createLoader()),
      switchMap(action => this.api.sendContribution(action.data)
        .pipe(
          map(() => ContributeActions.contributeSuccess()),
          catchError(err => {
            return of(ContributeActions.contributeFailed());
          })
        )
      )
    );
});
```

Listing 6.9: Submitting Contribution Data

Once the API sends a response, the effect dispatches either a success or failure action, which will clean up the contribution data in the feature stores, resetting the state for another contribution.

Navigation

Navigating from one step to the next is important. In this particular case, navigation is done through side actions in our effects. This allows us to isolate the components in terms of dependencies and restrict access to the Angular router to the effects. Listing 6.10 shows the effect triggered when the user sets the orientation that navigates to the next page.

```
step2$ = createEffect( () => {
  return this.actions$
    .pipe(
      ofType(ContributeActions.setUserPosition),
      tap(() => {
        this.router.navigate(['contribute/verify-orientation']);
      })
    );
}, {dispatch: false});
```

Listing 6.10: Navigation in Effects

Navigating back is done through use of the back button provided by Ionic that has been placed on the navbar as seen in Figure 6.4.

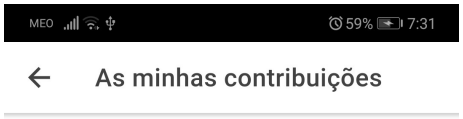

Figure 6.4: Back Button

Ionic keeps track of the components we navigate to and the order of navigation. Whenever we press the back button Ionic will pop the last item from the navigation stack and render the previous component.

6.1.4 Viewing Contributions

Another important aspect of the mobile app is providing means for the user to visualize his own contributions. This functionality has two steps, fetching a list of contributions associated with the user, and fetching a specific contribution from the returned list. This sequence is best illustrated in the sequence diagram 5.14.

Contribution List

To view the users contribution list, the user first navigates to the Contribution List page. Associated with this route is a resolver that dispatches an action to the store. The effect associated with this action calls the API service and loads the contribution list.

The data loaded from the server contains the information required to instantiate the list of contributions, namely the timestamp and the id of the contribution. With this data, we initialize our state using the Entity State, where our entity is the list item returned from the server. This can be seen in the snippet in Listing 6.11.

```
const adapter = createEntityAdapter<UserContribListItem>({
  selectId: model => model.fid,
  sortComparer: sortContributionListItems
});
```

Listing 6.11: Contribution List Entity Adapter

Additionally, we pass a sorting function (sortContributionListItems) to the adapter, so all entities will have the order we define. In this case, we require our entities sorted in a descending order by their timestamps. The list is reloaded from the server each time the user enters the component.

To display the contribution list, we pipe the **selectAll** store selector provided by our entity adapter to the observable **contributions$**. The snippet in Listing 6.12 shows the creation of the list.

```
<ion-list *ngIf="(contributions$ | async).length > 0 else noContributions">
  <ion-item *ngFor="let item of contributions$| async"
            (click)="viewContributionDetails(item.fid)">
    {{ item.timestamp * 1000 | date: 'medium' }}
  </ion-item>
</ion-list>
```

Listing 6.12: Contribution List

In Listing 6.12, we have two important parts, which are checking if the contribution list has any items and displaying those items. The checking of the contribution list is performed by the ***ngIf** directive, which verifies the length of the list obtained through the **contributions$** observable. Displaying the list of contributions is done with the ***ngFor** directive. The data displayed in the list items is the timestamp. The timestamp value is transformed to a readable format through the use of the **date** pipe. Both directives automatically subscribe to the **contributions$** observable using the **async** pipe. The **async** pipe automatically unsubscribes the observable when the component is destroyed, avoiding any memory leaks. An example of a contribution list can be seen in Figure 6.5.

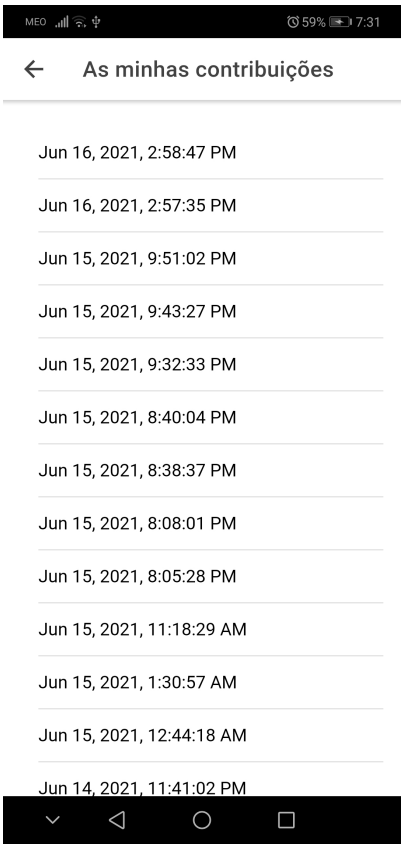

Figure 6.5: Verify Orientation Component

Contribution Details

Once the user clicks the list item corresponding to the desired contribution, the application navigates to the component where the data will be displayed. Similarly to the contribution list component route, the contribution detail component route has also a resolver. This will call the API service to fetch the data associated with the selected contribution.

The loading time of a contribution is a critical aspect of a responsive application. A direct approach would be to load all content in a single request. But we verified that sometimes it passed the threshold of the 4s as defined in chapter 4. This was due to the size of the image the user submitted of the fire. As an alternative approach, we load the image in a separate step. This allows us to initialize the map and display the data within the established time threshold, while the image is being loaded from the server.

These two steps require two store operations, one to create the entity of the contribution detail with the data and another to update it with the image. As presented in Listing 6.13, the more complex operation is perhaps the update of the entity, which takes an Update<T> object as well the current state as parameters. Update<T> requires two fields, one is the id of the object we are going to update, the other is the changes in an json object format. Since we wish to update the image field of the contribution, we require the Update<T>, where a is the NgRx action payload, and r, which is the response from the server.

```
const updated: Update<SingleContributionClean> = {
  id: a.id,
  changes: {
    image: r.data
  }
);
```

Listing 6.13: Contribution Details Update

To maximize the responsiveness of the application, we should also minimize the number of requests to the backend server. We accomplish this minimization by persisting the contribution details in the store for the duration of the session, using the Entity State. The API request to the backend is regulated by checking if the data is present in the store before doing the request. The sequence of operations is presented in Listing 6.14, where the **concatLatestFrom()** operator reads the data from the store, and the other operators are used to check if the data exists in the store and block the request to the backend server if so.

```
concatLatestFrom(action => this.store
  .pipe( // adds the selection result to the stream
    select(selectSingleId, {id: action.id})
  )
),
tap(([_, selectResult]) => {
  if (selectResult !== -1) {
    this.store.dispatch(ViewContributionActions.contributionAlreadyLoaded());
  }
}),
filter(([_, selectResult]) => selectResult === -1),
```

Listing 6.14: Contribution Details Update Effect

As for displaying the data itself, we use the **asymmetrik/ngx-leaflet** package to create the map, initializing it with the values selected from the store for the desired entity, and the same base map as in the contributing process. Once the photo is loaded, we sanitize the content using the Angular DomSanitizer and update the component to display the photo, as we can see in Figure 6.6.

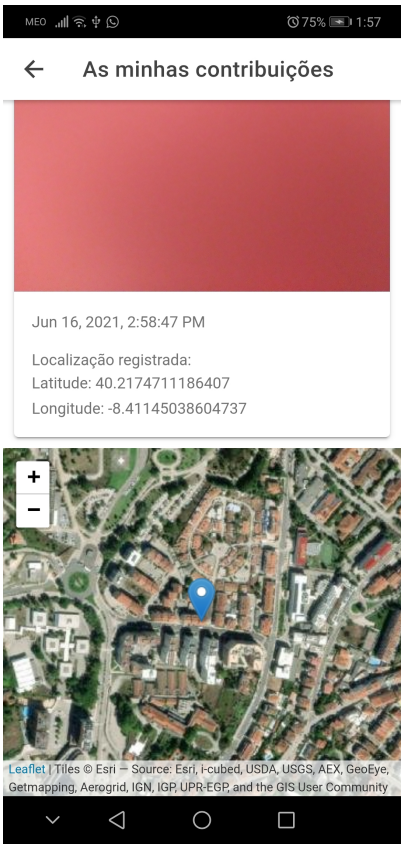

Figure 6.6: Contribution Details Component

6.1.5 User Management

The mobile app allows the user to manage his Fireloc account. This functionalities include changing the user's password, email, edit the user profile as well as deleting the account.

User State Management

User management is handled in the settings page. The state described in Listing 6.15 is used to keep the profile data loaded from our server. This loading operation is done by a resolver on the settings page route, which calls the backend server. The success action will update the store with the information. As with all other feature stores in the app, when a logout action is dispatched to the store, we clear the data from the profile.

```
interface UserState {
  email: string;
  firstName: string;
  lastName: string;
  tig: string;
  forest: string;
  fwho: string;
  instruction: string;
}
```

Listing 6.15: User State

Editing data

Changing the email and the password is a fairly simple process, in which, the user navigates to the desired page, fills the fields (according to the use cases UC10 and UC11) and submits the data. The submission of the data dispatches a NgRx action to the store with the new data. This triggers the corresponding effect that makes the appropriate API call.

Changing the profile information is a more complex process. While some of the fields are simple enough to understand, some require more careful consideration and the data may change according to the perception of the user at the moment he edits the profile. For example, he may not remember if he filled a given field when the account was created or the value he inserted. Therefore we prefill the profile with the data from **UserState**, with the user only changing the parameters that require changing. Submitting the data follows the same process of editing the other parameters of the user account.

Validation

Sensitive data like passwords and emails are always validated before submitting. The parameters of validation are the same present in the backend server, such as pattern matching for email validation, length of password, or inserting twice the password.

6.2 Implementation of the Web Portal

In this section we describe the implementation of the web portal according to the architecture defined in section 5.3.2. To implement it, we divided the functionalities into several categories: authentication, data visualization and user management. Following angular best practices, we encompassed each of these three categories in an angular module, authentication, data visualization, user management.

6.2.1 User Authentication

Authentication is an important aspect of the portal. We need to ensure each user only has access to his allowed content, and, we need to make it a secure process. We now detail the implementation of authentication, which is encapsulated in the authentication (auth) module. The auth module implements our authentication service and includes the UI components related to authenticating a user.

Firebase Service

We use Firebase to accomplish the authentication. Firebase is a mature solution that has all the authentication methods identified in the requirements and is compliant with the GDPR [9]. Its integration allowed to reduce the development time of the web portal. Among the available authentication methods in Firebase, we implemented the email/password authentication and google login.

It is important to define and clarify some terms and expressions. A Fireloc user/account is the user representation in the Fireloc server. A Firebase user is the user representation on Firebase and is associated with the Fireloc user. A google user is also a Firebase user.

The process of authentication is as follows:

1. The user inserts his credentials,
2. the credentials are validated on Firebase,
3. Firebase returns a Firebase user reference,
4. we retrieve the userId for the Firebase user,
5. the userId is sent to our server, through our google authentication endpoint,
6. the server validates the userId using the Firebase admin SDK and fetches the user from google,
7. a Fireloc account is created if none exists associated with the user, otherwise we fetch the Fireloc user,
8. the server generates a token associated with the Fireloc account and sends it to the client,
9. the client stores the token to be sent with future requests

These interactions can be observed in the diagram in Figure 6.7. In the diagram we adapt the 3rd party login sequence (5.10) to the specifics of Firebase.

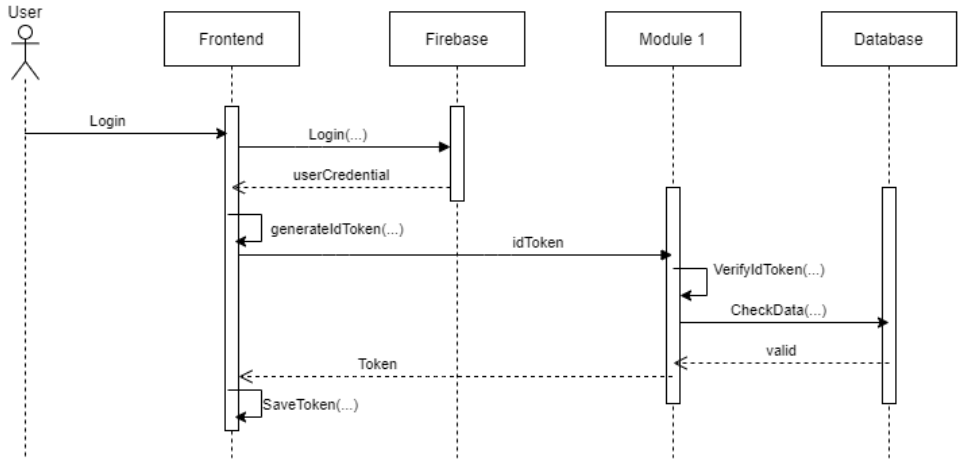

Figure 6.7: Firebase authentication sequence

To implement the authentication functionalities, we created a Firebase project. There, we generated two sets of credentials, one for the web client in the form of a JSON object, and a service account key, in the form of a JSON file stored in our server. In the snippet of Listing 6.16 we present an example of the web client credentials.

```
firebaseConfig: {
  apiKey: 'my_api_key',
  authDomain: 'authDomain',
  projectId: 'projectId',
  storageBucket: 'storageBucket',
  messagingSenderId: 'messagingSenderId',
  appId: 'appId',
  measurementId: 'measurementId'
}
```

Listing 6.16: Firebase Configuration Object

On the web client, we import the AngularFireModule and initialize it with the web client credentials in the app.module file. The AngularFireModule is part of the @angular/fire (or angularfire2) package [2], which is the reference package for using Firebase with Angular. Then, in the auth module at auth.module.ts, we import the AngularFireAuthModule. While we could have imported this module directly at app.module, since all authentication functionality will be encapsulated in our auth module, it's better to only have access to the AngularFireAuthModule from the modules that require those functionalities.

From Firebase itself we need two groups of functionalities, authentication functions and user management. The authentication functions are encapsulated in the auth service, while the user management in the api-http-settings service in the settings module. These services are in turn called by the NgRx effects each time a corresponding action is dispatched from the store.

The output of the angular/fire functions in the auth service are converted into observable streams for better integration with NgRx effect streams. This conversion is accomplished using the RxJS **from()** operator.

Authentication State Management

After setting up the auth service, the next step is to generate the AuthState feature store, which is the slice of state responsible for everything related to authentication. The auth store, shown in the snippet of Listing 6.17 is divided into three parts: **flags**, containing any flag to regulate component behavior, **errorCodes**, containing all the authentication error codes supported by the portal, and lastly, **auth**, containing authentication state flags.

```
interface AuthState {
  flags: AuthFlagsState;
  errorCodes: AuthErrorCodesState;
  auth: AuthenticationState;
}
```

Listing 6.17: Settings State Interface

The AuthState is perhaps the state where most of its properties are used separately; so we created selectors to access each single property, which were grouped using the same criteria as the properties.

The actions for authentication are more complex than those of the mobile app, since we

have here multiple errors defined associated with each function. This is exemplified in Figure 6.8, where for each action dispatched from a component, we will have a success response action and a corresponding failure response action, which in turn dispatches an error action associated with one of the predefined codes.

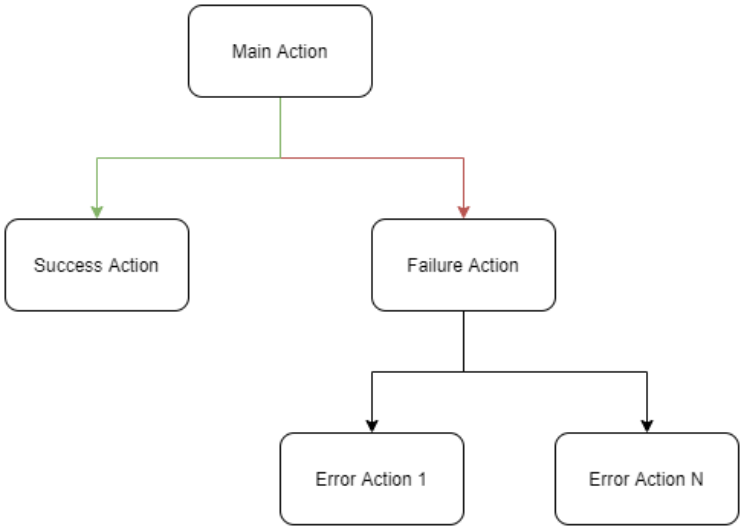

Figure 6.8: Action Hierarchy

These sequences of actions, main/success or main/failure/error actions, occur in the authentication effect. As presented in Listing 6.18 this effect starts by filtering the Actions Observable stream through the NgRx **ofType()** operator, resulting in the main action observable. This observable is processed by the RxJS **switchMap()** operator, which calls the corresponding API function. To the response observable we pass two operators: **map()**, to dispatch the success action and **catchError()**, which dispatches the failure action with the error code as a property. The handling of the error code is in a new effect that maps the error code to the respective action.

```
authenticateUser$ = createEffect(() => {
  return this.actions$
    .pipe(
      ofType(firebaseLoginUser),
      switchMap(a => this.auth.firebaseLogin(a.email, a.password)
        .pipe(
          map(() => firebaseLoginUserSuccess()),
          catchError(e => {
            return of(firebaseLoginUserFailure({errorCode: e.code}));
          })
        )
      )
    );
});
```

Listing 6.18: Firebase Login Effect

After discussing the flow of actions, it remains to present the reducers. Apart from the failure and the setCookie actions (and none of these alters the store state), no other actions

requires props, so our reducer function on each action will generate a new state object based on a preconfigured value. The auth state starts with the error codes initialized as false and upon each emitted error action, the reducer changes the value to true. In each main action, the reducer resets all properties associated with that action and possible error codes. Both cases can be seen in the snippet of Listing 6.19. Success actions, such as, login-success and logout-success continue to alter the state in their expected behavior.

```
const _reducer = createReducer(
  on(FirebaseAuthActions.firebaseLoginUser, state => {
    return {
      ...state,
      normalLoginFirebaseUserNotFoundError: false,
      normalLoginFirebaseInvalidEmailError: false,
      normalLoginFirebaseWrongPasswordError: false,
      normalLoginFirebaseNetworkErrors: false
    };
  }),
  on(FirebaseAuthActions.firebaseLoginUserFailureInvalidEmailError, state => {
    return {...state, normalLoginFirebaseInvalidEmailError: true};
  }),
)
```

Listing 6.19: Authentication Reducer Snippet

Route Guards

In the mobile app, the user had no access to the paths of the components and the app regulated the navigation through a defined sequence. But in the web portal, the users have access to the urls, being able to access content they're not authorized to navigate to. We can avoid this limitation using Angular guards.

Angular guards are classes that implement one or more of the five interfaces, which are **canActivate**, **canActivateChild**, **canDeactivate**, **resolve** and **canLoad**. These can be passed to our routes configuration and whenever a user tries to access the route, the guard mediates access to that route.

Among the possible interfaces, we chose the **canActivate** guard interface. In its implementation, we select and evaluate the isLoggedIn state property, which indicates if a user is authenticated. If the user is authenticated, we return the value 'true'. This indicates the route can successfully resolve and it allows access to the content. If the user is not authenticated, we return the value 'false'. This should have solved our problems, however, if the user was not authenticated, he would be left in a blank page, since no content was allowed to be displayed. Thus, we also indicate the router that it should navigate to the home page before resolving, allowing for a more pleasant user experience.

6.2.2 Security

One of the advantages of using Angular to build the web portal is the support for security features. So, creating a secure application passes largely by following the security patterns, as defined in the official Angular documentation [49]. Thus, we now describe the steps taken to secure our web application.

Secure Elements

To display images coming from our server, we must first make the value safe, this is done through the **bypassSecurityTrustResourceUrl** method available in the DomSanitizer class. This allows us to mark the image value as safe for Angular.

Http Requests

There are several ways to make an http request, one is through the Fetch API, the other being through the HttpClient, a service class available through the Angular common package. Http requests are usually vulnerable to attacks such as cross-site request forgery (XSRF) and cross-site script inclusion (XSSI). To mitigate these risks, it is recommended we use HttpClient [49], a recommendation we followed.

HttpClient automatically sets a X-XSRF-TOKEN http header on all mutating requests to relative urls, which can be used to identify the origin of the request, thereby protecting against XSRF attacks. For XSSI attacks it automatically recognizes the ")]}',
n" string used by servers to make JSON responses non-executable and parses the response content.

It is also important to note that all requests to the Fireloc API are made through HTTPS.

6.2.3 Data Visualization

The visualization of fire data is the main goal of the web portal. The functionalities are querying the database according to a specific set of parameters (location, start and end date), and visualizing statistical and generic data.

Authentication and permissions

These functionalities require data at several levels of permissions. The generic data, which are active fire events, does not require authentication to access it, while the remaining, such as query results and statistics, do. To achieve this separation, the statistics data has its own page, protected by the authentication guard, while the main page displays the various components based on the authentication status. If a user is authenticated, the details panel, the query bar and the details button are displayed, if he's not, only the map is displayed and the user can not perform those actions.

The amount of data displayed on the portal, depends on the type of user and is handled on the server side. The portal only displays data and does not make any kind of filtering based on permissions. This permits the portal to only fetch relevant data from the endpoints.

Queries

To make a query, the user interacts with the search-bar component. The component has four fields: District, County, Start Date and End Date, a Search button to submit the query and a Reset button to return all to the default values. All available values are preset into the options of each field, reducing the chance of error by the user.

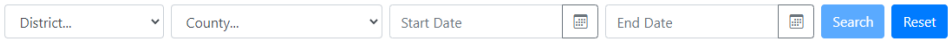

Figure 6.9: Search Bar Component

The 'District' and 'County' fields serve to specify a location we wish to query. A user can pick the county directly from a list of all available counties or, can first choose a district, and from there pick the county from those of that district. Should a user pick a district and no county to perform a query, the county field is highlighted to indicate the necessity to fill that field. It was decided to lock queries to a single county. So, a certain fire event may be present in several queries made with different locations if the fire was active in more than one county. District and Counties were taken from the Carta Administrativa Oficial de Portugal [6].

The Start and End Date fields delimit the interval of time that a fire may have existed in. Start Date delimits the earliest day for the fire to have started. End Date, the latest day for the fire to have ended. A user can omit either parameter in the query. If no start date is chosen and end date is, all fires ending at or before the end date will be displayed. If no end date is chosen, all fires starting at or after the Start Date will be displayed.

Each of these three parameters can be used alone to make a query or grouped for a more detailed result.

Once the parameters are set, pressing 'Search' dispatches a NgRx action with the values for each parameter. Following the flow of the NgRx effect, through the use of the **switchMap()** operator, the API service makes an HTTP call to the backend server. Once a response is received, if data is returned, the store is updated and the new data is displayed on the map component.

Event Details

To obtain an event's details, the user simply has to select the event marker and press 'Details'. This will dispatch a NgRx action to the store and trigger the fetchEventDetails effect, which in turn will make an HTTP GET request to the API. A successful response will contain a GeoJson payload. This payload contains the data necessary to draw the extent of the fire event upon the map, producing the result observed in Figure 6.10.

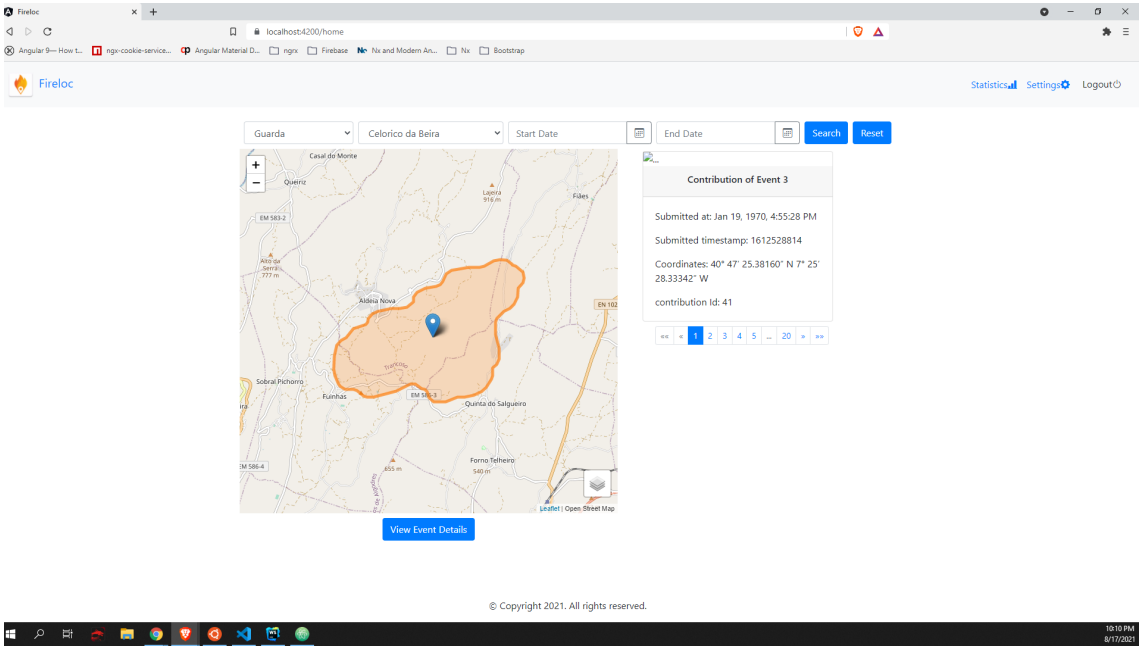

Figure 6.10: Web portal fire extent

Unlike some of the data the user fetches from the server, such as the contributions, the application does not save the result from fetching the event details to the store, since this data can change between queries as more contributions are added and the predicted area of the fire changes.

Contribution Details

The contribution details are the data available per user type from the contributions submitted and associated with a fire event.

Fetching the data is a two step process. First, we fetch a list of contribution ids, which are returned in the same response as the event details. Based on the returned ids, the NgRx entity adapter is initialized with the id field, but the other fields remain to be initialized. The interface of the entity adapter is displayed in Listing 6.20. Second, we load the remaining data of the contribution from the server. This is done by calling the endpoint '/events/event_contribution/${contributionId}/' with the id of the contribution to be fetched. This endpoint returns the remaining contribution data, which is used to update the entity collection.

```
interface ContributionDetail {
  id: number;
  location: string;
  direction: string;
  img: string;
  timestamp: number;
}
```

Listing 6.20: Contribution Detail Interface

The update of the entity collection is performed by an Update<T> object, where T is

the interface of the entity. The Update<T> object requires two fields, one is the id of the object we are going to update, and the other is the changes in JSON format. So, we update all the remaining contribution fields creating the Update<T> object as presented in Listing 6.21.

```
const updatedDetails: Update<ContributionDetail> = {
  id: data.id,
  changes: {
    id: data.id,
    location: formattedLocation,
    img: data.img,
    timestamp: data.timestamp
  };
```

Listing 6.21: Contribution Detail Update

This update object is then sent as payload on a success NgRx action, which triggers the store reducer. The reducer will call the entity object and update the state through updateOne() as presented in Listing 6.22.

```
on(MainActions.fetchSingleContributionDetailsSuccess, (state, action) => {
  return adapter.updateOne(action.update, state);
})
```

Listing 6.22: Contribution Detail update reducer

Filtering Contributions

Filtering a contribution is performed based on the confidence degree of the fire event position and on the confidence degree of the contributor position. Both, by default, have an interval value ranging from 50% to 100% of confidence, allowing the visualization of only the more trustworthy contributions. The filters are added as parameters to each query request sent to the server. This component is depicted in Figure 6.11.

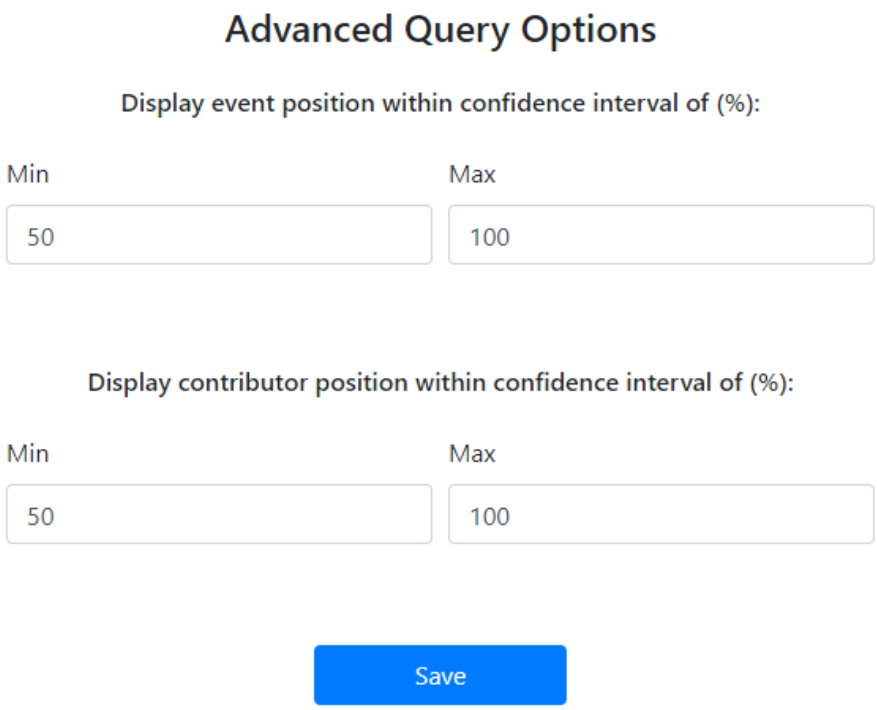

Figure 6.11: Advanced Query Options Component

These values can be edited at the settings page under 'advanced settings'. This decision was made so as not to clutter the main page with too many options.

6.2.4 User Management

Similar to the mobile app, the portal allows for the user to manage his account. These functionalities include changing the user's password, email, delete his account. These are integrated in the settings module.

Authentication

For a user to have access to any of these functionalities, he must be logged in, since all components are protected by the authentication guard. Given the integration of the authentication services with Firebase, we must comply with their best practice policies. One of them is the requirement that the user may not perform these operations if he has not logged in for a long period.

Firebase Integration

To perform any action, which alters the firebase user, we require a user instance. We can obtain one from the auth service, which is provided by the auth module. Having the user instance, it's only a matter of calling the relevant method from the user and pass the parameters if necessary.

As before, these actions take place in a specific service (settings.service), where we wrap the calls to firebase with the **from()** RxJS operator. This transforms the return value into an observable stream to be used with the effects. The effects will handle the calls to the service. However, unlike the authentication functions that return Promise<User> or Promise<UserCredential>, firebase only returns Promise<void> with these actions or an error code.

State Management

The user management execution flow is similar to the authentication process as described in section 6.2.1. We create a SettingsState feature store, with two features, errorCodes and options, as observed in Listing 6.23.

```
interface SettingsState {
  errorCodes: SettingsErrorCodeState;
  options: SettingsOptionsState;
}
```

Listing 6.23: Settings State Interface

Validation

Data inserted by the user is validated through similar patterns as described in the mobile app 6.1.5. This includes the same methods of inserting twice the password when changing it as well as using the same pattern for password and email validation. This reuse of basic services was possible to the design choice of using the same technology in the mobile app as previously discussed.

6.3 Implementation of the Geoportal API

In this section we describe the work developed in the context of this dissertation regarding the Geoportal API. This consisted in the implementation of the API endpoints for the authentication and the queries, which support the web portal. A restriction of the project was the use of the Django framework.

6.3.1 Authentication

Support for authentication is a crucial part of the API server. This ensures that access to the server's resources is given only to trusted clients.

In the authentication process of the backend server, we used google authentication. This comprehends the creation of an authentication backend that uses the Firebase Admin SDK, and the definition of an endpoint to receive requests from the portal.

Authentication Backend

In the Django framework the BaseBackend class should be extended for the integration of new authentication methods. So, for integrating the Firebase Admin SDK, we extend this class implementing two methods, which are **authenticate()** and **get_user()**. The purpose of the **authenticate()** method is to validate the credentials given and authenticate the user, while the purpose of **get_user()** is to return the authenticated user from the database.

To validate the credentials and authenticate the user, the **authenticate()** method receives a request and other arguments, such as the google token, which is the idToken generated by Firebase.

The validity of the idToken is verified with the Firebase Admin SDK. To use any Firebase admin functionality, we must initialize it with our client id. This is a JSON key file generated on the Firebase web console. We initialize our credential with the key and use it to initialize the Firebase SDK, using the **initialize_app()** function. This function returns an initialized instance of the Firebase app. This can be observed in Listing 6.24

```
cred = credentials.Certificate(CLIENT_ID);
firebase_app = initialize_app(cred)
```

Listing 6.24: Initialize Firebase Admin

With the Firebase app initialized, we can now use the authentication module and call the **verify_id_token()** method, passing the id token. The **verify_id_token()** method decodes the idToken and validates the token payload, such as the expiration date, issued-at-time, issuer, audience, subject and authentication time[1], any issue found with the token raises a ValueError exception. This is presented in Listing 6.25.

```
try:
  decoded_token = auth.verify_id_token(token)

  email = decoded_token['email']

  if not email:
    raise ValueError('No email present in token.')

except ValueError as e:
  return None
```

Listing 6.25: Validating the idToken

If the token is valid, we can access the decoded payload and retrieve some information about the user. In our case, we require the user's email. With the email we can verify if a Fireloc account is already created with that email. If no account is found, we create a new Fireloc user, using the **create()** method of the User model, saving it to the database through the **save()** method as seen in Listing 6.26.

[1]In accordance with the instructions at [32]

```
try:
  user = User.objects.get(email=email)
  return user

except User.DoesNotExist:
  user = User.objects.create(username=email, email=email)
  user.save()
  return user
```

Listing 6.26: Fetching the user

Lastly, it is necessary to configure the Django application to use this new backend for authentication. For this, we reconfigure the settings file to use the new backend by adding it to the list of authentication backends as indicated in Listing 6.27.

```
AUTHENTICATION_BACKENDS = ['authapi.auth_backend.AuthBackend']
```

Listing 6.27: Configuring Settings

When using the google authentication method, we encompass both authenticating the account and creating it in the same function. This is due to the fact that everything concerning the google authentication method coming from the frontend client, using the idToken.

API Endpoint

To interact with the Authentication Backend, we created a new API endpoint. This is accomplished by extending the APIView class to handle google authentication. In this class we implement the **post()** method, which will handle HTTP POST requests to the associated url.

Once the payload is extracted from the request, the APIView calls the authenticate method from the django.contrib.auth module. Since we have the authentication backend configured, and we are passing google_token as an argument, **authenticate** will validate the idToken present in the payload ('token_param'). Once the user is authenticated, the token for the user is created using the the Django rest framework app **rest_framework.authtoken**, as presented in Listing 6.28.

```
    if token_param in data:
        user = authenticate(google_token=data[token_param])
    if user is not None:
        token, created = Token.objects.get_or_create(user=user)
```

Listing 6.28: Token generation with google authentication

6.3.2 Queries

The queries are the main functionalities of the API endpoints. As described in the Web Portal section, the query process is twofold. First, the client makes a query to the server,

which responds with a list of fire events and the associated data. Second, the user will choose one fire event to obtain details about. These functionalities were implemented in the **events** Django app.

Fire Event List

To obtain the fire event list, we defined the **EventQuery** class by extending **APIView**, where we implemented the **get()** method, which processes GET requests. In the GET request payload, we expect three query parameters from the client, which are a location, a start and an end date. These are used to perform a query to the database as detailed in section 6.2.3.

```
geom = Concelhos.objects.get(dicofre=location)

event_result = FireEvent.objects.filter(
        extent__intersects=geom.geom
    ).filter(
        maxtime__lte=end_time
    ).filter(
        mintime__gte=start_time
    )

data = {'events': fire_event_query_result_cleaner(event_result)}
```

Listing 6.29: Event Query

The snippet present in Listing 6.29 shows the event query when using all query fields. First, we define the concelho restricting the fire event location. This is done by querying the Concelhos model and fetching the object with the dicofre value matching the one sent in the request. Since the dicofre value is unique, we only get one. With the concelho object, 'geom', we can now perform a query on **FireEvent**.

The **filter()** method returns a **QuerySet** object that allows the chaining of other filters, forming a complex query. So, in Listing 6.29, the first **filter()** call defines the condition on the geometry field, which here defines a fire where its geometry intersects the geometry of the location we have previously fetched, geom. This condition is further elaborated in the second filter. This defines the time that the fire event ended, which is equal or inferior to the given date. The last filter defines the condition of the start date of the fire, which is equal or greater than the given one.

The queryset will return the FireEvents matching the parameters, however, the user may not require the entire FireEvent object, so we pass the queryset to the helper function **fire_event_query_result_cleaner()**, which will return the events in a simpler format ready for the client to display. Among the changes done in the helper function, the most important is the conversion of the coordinate system that will convert the portuguese coordinate system to the standard coordinate system, which is defined in Listing 6.30. Since django has built-in tools facilitating this process we convert the points to be displayed to the 4326 standard system in the server before sending them to the client.

```
from django.contrib.gis.gdal import SpatialReference, CoordTransform
from django.contrib.gis.geos import Point
def convert_centroid_coord_system(centroid):

    pt_reference = SpatialReference(3763)
    display_reference = SpatialReference(4326)

    trans = CoordTransform(pt_reference, display_reference)

    new_point = Point(centroid.coords[0], centroid.coords[1], srid=3763)

    new_point.transform(trans)

    return new_point
```

Listing 6.30: Converting Coordinate Systems

Fire Event Details

To obtain the fire event details, we implemented the **SingleEventDetails** class extending **APIView** in which we implemented the **get()** method. The **get()** method receives the id of the event as a query parameter and returns as a response the geometry of the fire event, which is its extent area, and the list of contribution ids. The contribution ids are obtained by filtering the Points queryset for the fire event id received from the client as presented in Listing 6.31. From the query results we will need only the fid field.

```
    result = Points.objects.filter(
        idevent__exact=id
    )

    ids = result.values('fid')
```

Listing 6.31: Fetching contributions

The extent (area) of the fire event is slightly more complex, and this is because we need to serialize the data into the geojson format that leaflet can read. Fortunately Django permits performing such an operation. The snippet of Listing 6.32 shows this operation. We define the output format as geojson, pass the queryset, and specify the geometry field, which in our case is 'convexhull' and indicate the fields that appear in the property field. The output will be a geojson string ready to be processed by leaflet, which is returned in the response payload.

```
geom = serialize('geojson', FireEvent.objects.filter(fid=id),
                  geometry_field='convexhull', fields=('fid',)
                )
```

Listing 6.32: Serializing to geojson

6.4 Summary

In this chapter, we presented the implementation of the mobile application, the web portal and the Geoportal API, which included the discussion of the UI of the applications and the inner structure. The Ionic framework allowed some services and functionaties to be shared between the mobile app and the web portal, namely the validation of input data and forms, the visualization of the contribution data, and the supporting NgRx infrastructure. Also, the choice of the NgRx permitted agregating locally in a shared entity the data used by the components of each application. Also, another advantage of NgRx was the reduction the number of dependencies of each component. The combination of these technologies sped up the development time and increased the robustness.

In the mobile application, to improve the quality of the data collected we implement steps for calibrating and positioning the device, and ensure all required services are enabled.

In the implementation of the security policies we followed the best practices of Angular. Also, the use of Firebase permitted the inclusion of different forms of authentication, improving the usability of the web portal application. Also as a side effect, it decreased the cost of implementing GDPR.

Finally, we discussed the implementation of the Geoportal API.

Chapter 7

Testing the System

In this chapter we present the tests done to the mobile app and the web portal to ascertain the good implementation of several critical functionalities.

First, we will discuss the unit tests performed. This includes the test setup and the evaluated system parts. Next, we will discuss the non-functional requirement tests. These will focus on evaluating the mobile app and the web portal according to the metrics described in Chapter 4. Following the non-functional requirement tests, we will go over the user tests and their results. Here, we evaluate how well users have responded to real life experience with the system.

7.1 Unit Testing

The NgRx portion of the systems integrates the application components with each other and handles the state of the system, including the data collected in the mobile app and requested in the portal. Therefore, it is critical to ensure the NgRx components (actions, reducers and selectors) are working as expected, since any error can have serious reprecursions when we consider how sensitive the data we are handling is. To do this, we must ensure: the data in the store is securely modified at the correct time by the appropriate reducers; the actions dispatched activate only the reducers required for its intended operation; and the selectors are fetching only the required data. So, the unit tests were concentrated on the NgRx portions of the mobile app and web portal.

In the implementation of the tests, we use Jasmine [36] and Karma [37] both of which come with Angular and Ionic. Jasmine is a testing framework that allows describing test suit for each file, while Karma runs the Jasmine tests.

7.1.1 General test suit setup

Since the actions will modify the store state, for each reducer we create a test suit, which will evaluate the following parameters:

- An action only changes the necessary properties of the state.
- The properties changed by an action assume an expected value.
- Only actions the reducer listens to are altering the state.

- Each reducer creates a new state from the previous one, ensuring state immutability.

For the selectors a different approach was taken. We created a mock state with all the values available in the state and our test suit evaluated if each selector is only returning the values it is supposed to.

The test parameters described above follow NgRx best practices as described in the official documentation [44]. Also, Jasmine establishes a structure in the development of the test, requiring that each component tested should have an associated spec file. So, it is the conviction of the author that the adoption of these practices will help the development and testing of new integrated components in the system.

7.1.2 Results

Using the setup previously described, we created 156 tests. The application of these tests allowed the detection of some subtle bugs that were promptly corrected and the current passes all tests. This ensures the good internal functioning of both the mobile app and the web portal stores.

7.2 Non-Functional Requirement Tests

We now present the tests performed on the mobile app and web portal to ascertain their compliance with the non-functional requirements of usability and latency as specified in Chapter 4.

7.2.1 Usability

As discussed in Section 4.1.3 and 4.2.3, to evaluate the usability of both the mobile and web applications we use the Nielsen's Usability Heuristics [45].

Heuristics

From the various heuristics defined by Nielsen, we chose those that better fit our products. They were:

- Match between system and the real world: use of familiar concepts to the user.
- User control and freedom: giving the user control of the flow of work.
- Visibility of system status: the design should give users feedback on what is happening.
- Error prevention: checking for error causing situations, or asking users to confirm, etc.
- Help users recognize, diagnose, and recover from errors: simple error messaging in plain language and suggestions of solutions.

Mobile App Evaluation

- Match between system and the real world: In the mobile app, the need for the users to easily interpret the instructions in diverse situations led us to use simple terms, both in the instructions and when visualizing data. This principle was also considered in the UI design of the app, where the UI elements were placed in their natural positions.

- User control and freedom: In the mobile app, the user has complete control over the flow of actions. All pages have a 'go back' button, allowing the user to return to the previous page. This action is also enabled by the 'return' hardware button. In addition, the contribution page also has a 'cancel' button in the expected natural position, which is the upper right corner of the page. This allows the user to cancel the contribution process and return to the main menu without going through the 'go back' buttons. Both of these examples are depicted in Figure 7.1 and can be found in their expected place in the navbar.

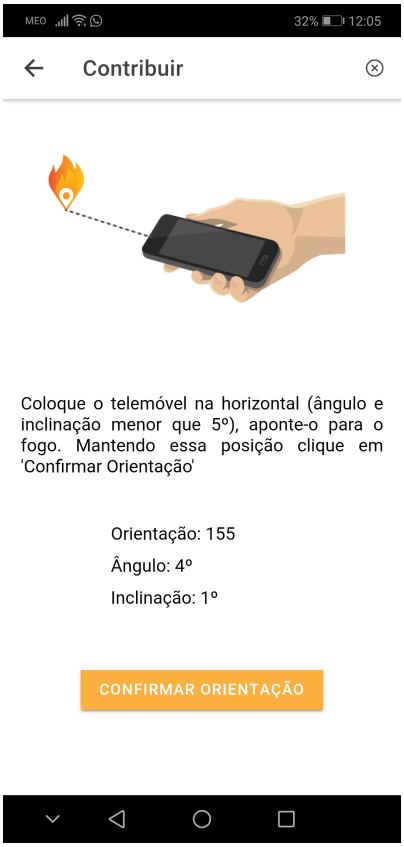

Figure 7.1: Cancel and Back Button example

- Visibility of system status: The mobile application is required to provide system feedback to the user. So, when the user is verifying his position, the provided feedback is the position clearly marked on the map. When he is measuring the orientation, the feedback is the orientation values, updated each time the device moves. Furthermore we also provide feedback on the status of system features, such as, loading states, the availability of the location and network services, which are used by the system. The location status is depicted in Figure 7.2.

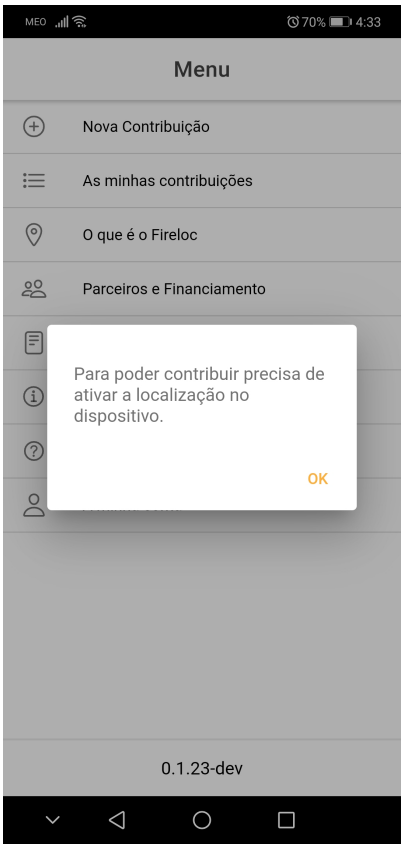

Figure 7.2: Location Disabled Warning

- Error prevention: The mobile app design was made with error prevention in mind. This principle is materialized, for instance, in the definition of predefined values in the input forms regarding the authentication of the user or in the creation/modification of his account, as presented in Figure 7.3.

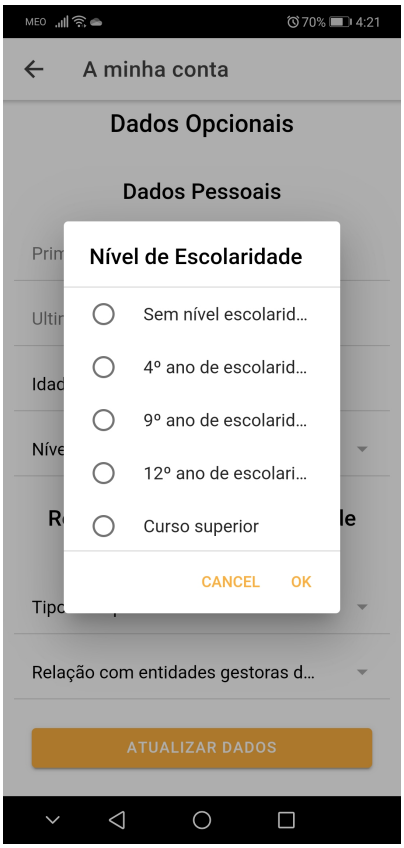

Figure 7.3: Account predefined fields

Another case of the application of this principle is found when the user interacts with the system to collect some data. In this, whenever possible, we established guards to help the user understand the correct way of collecting such data. This is the case depicted in Figure 7.4, where we display a warning when the user does not have the phone in the appropriate position.

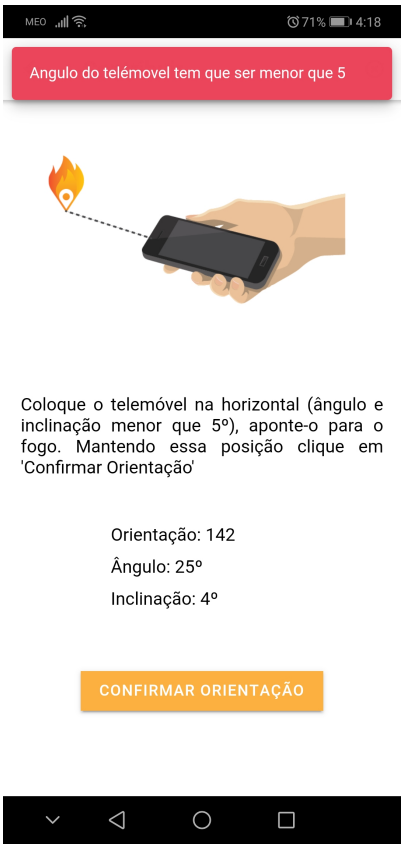

Figure 7.4: Angle Warning Message

- Help users recognize, diagnose, and recover from errors: Figure 7.4 is the perfect example of diagnosing and recovering from errors. Here the user receives a warning when he mispositions the phone, which allows him to recover and correct the phone position.

Web Portal Evaluation

- Match between system and the real world: In the portal design we tried to match our terms to the vocabulary most users are familiar with, such as using the term 'County' and 'District' instead of the less known official designations of NUT 2 and 3 [46]. Another consideration was matching the design with other similar products as described in Chapter 3.

- User control and freedom: In the portal, the user has complete freedom of his actions, being available the browser's capabilities of reloading, and going back and forward.

- Visibility of system status: The web portal hasn't as many interactions with the user as the mobile app, so the system status is relevant in the measure of error prevention and recovery, which we will detail in the next heuristics.

- Error prevention: The situations most likely to induce the user in error are those pertaining to input. User input is necessary when making a query, inserting user infor-

mation when registering, logging in, and changing settings. When making queries, the values are all predefined, thus preventing the user from inserting invalid errors, also, when a wrong value is inserted, the user always has the 'clear' button to reset the form values, as can be observed in Figure 7.5.

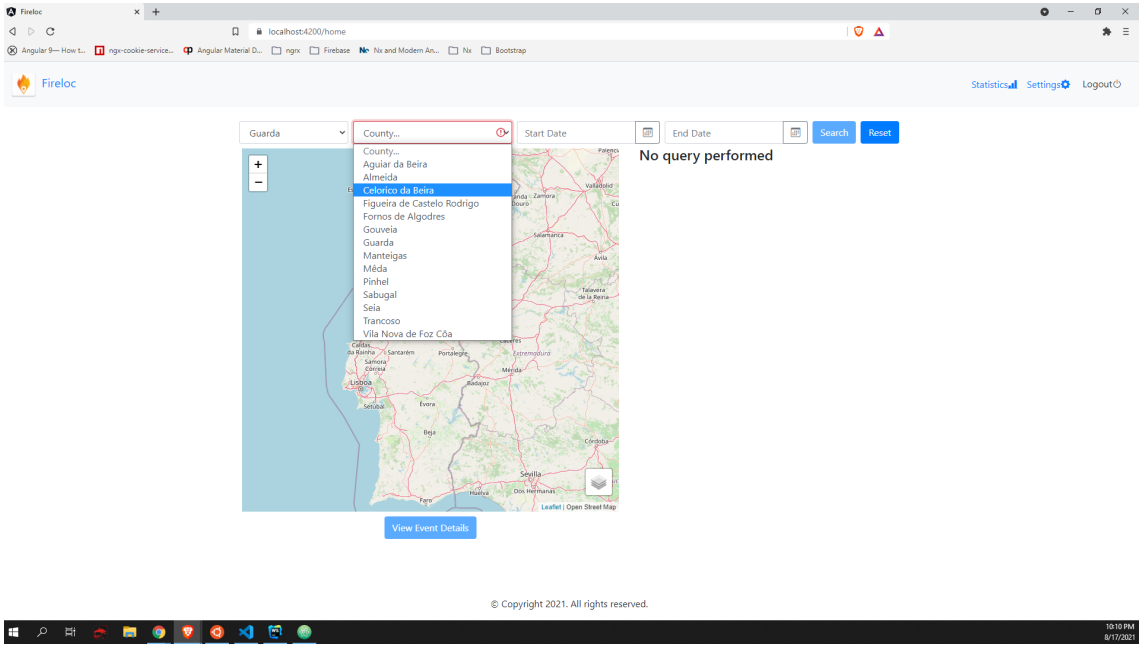

Figure 7.5: Web portal location form values

- Help users recognize, diagnose, and recover from errors: The portal has a wide array of status and error messages permitting the easy identification of any committed error. These enable him to understand if the error was committed by him or was a system error. In Figure 7.6, we present several possible errors during the authentication process. The reader will notice that these errors will not occur all at the same time and exemplify the thoroughness of the diagnostic.

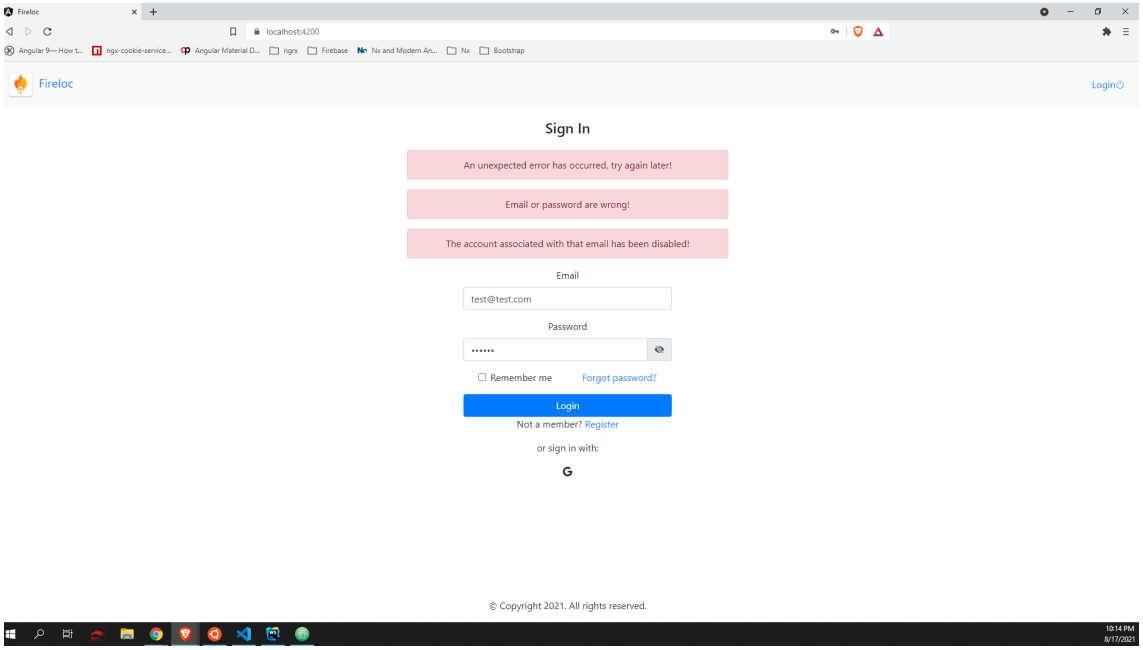

Figure 7.6: Web portal login error messages

7.2.2 Latency

The following box diagrams of Figures 7.7, 7.8 and 7.9 show the results of the measurements taken of the response time for the major requests done to the API.

Figure 7.7: Measurements for the mobile app

Figure 7.8: Measurements for the web portal

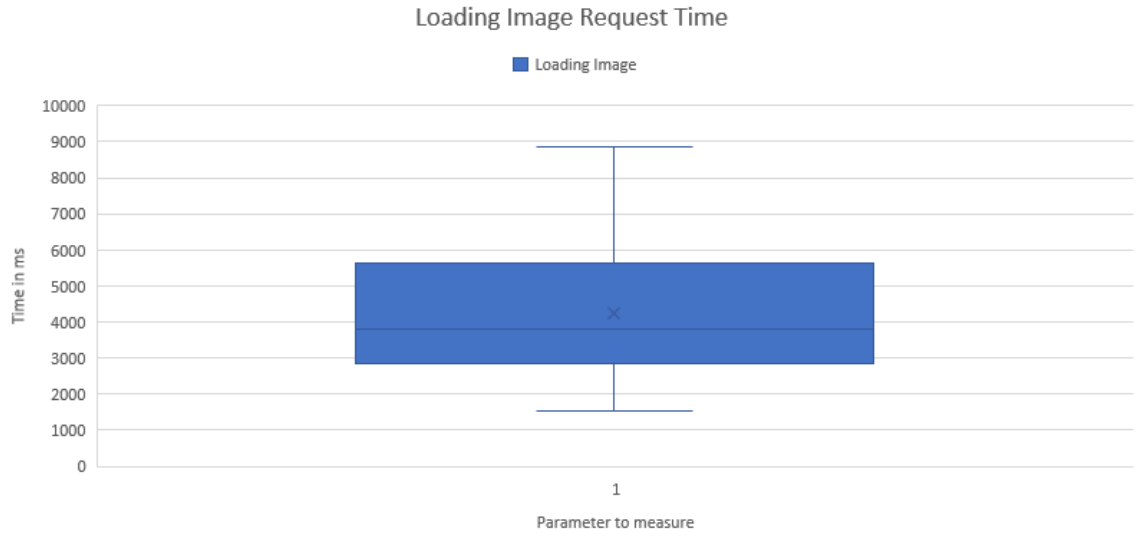

Figure 7.9: Measurements for loading images

As discussed in Section 4.1.3, the values for the request time should be lower than the 4 seconds threshold. Latencies less than 1 second are ideal for a more pleasant user experience. We have achieved this goal except for the specific case for loading images. We verify that when loading images, some cases have a latency superior to the 4 second limit. This cases can be explained by the size and resolution of the images, besides other factors such as the quality of the connection. As discussed in the section 6.1.4, we have mitigated this case by splitting the process. So, although this does not directly solve the problem of the image latency, it allows the user to see the page with information (text data) faster, which improves the usability in this case. As a future feature of the Fireloc System, the server could reduce the resolution of the images before sending them to the clients (mobile app and web portal).

7.3 User Tests

Thus far, the mobile app has been the part of the system which the team of the Fireloc Project has prioritized and thus has realized extensive field tests with real users of this system.

The Fireloc Project identified the mobile app as a critical component for field tests. So its test was prioritized and and extensive tests were performed. The teams that realized the tests have a diverse background in geography, geolocation, and informatics. They evaluated the user interactions, namely on the quality of the UI, UX and the provided information, which was considered in further adjustments. Also the robustness of the application was evaluated, which was its capacity to recover in situations with low quality and lack of signal connection.

7.3.1 Test setup

In performing these tests, the users were given a working version of the mobile app and access credentials. The team was split in groups of testers that were distributed to different locations in Coimbra, Figure 7.10. It was specified which type of photo to take and when to make the contribution.

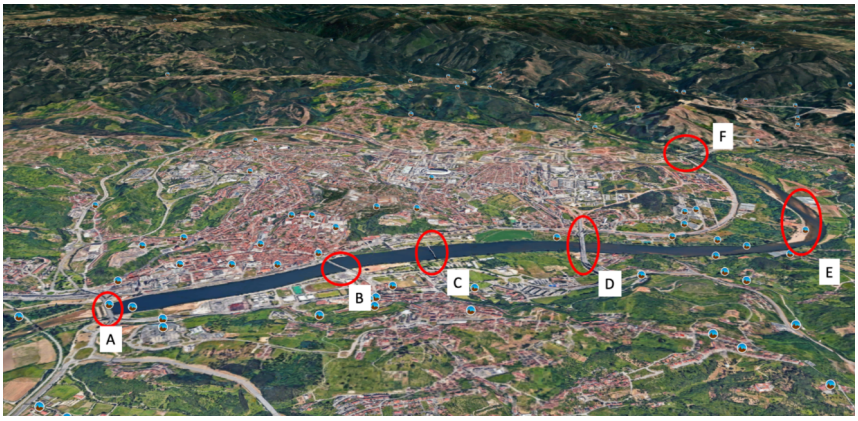

Figure 7.10: Locations of Interest for Field Tests

To evaluate the overall satisfaction with the mobile app, a questionnaire was given to the users after each test.

7.3.2 Results

The results summarized are on a scale of 0 to 5 with 5 being the highest satisfaction level:

- Overall ease in finding the contribution functionality: 4.9
- Overall definition of the contribution steps: 4.3
- Location step ease of use and comprehension: 4.5
- Orientation step ease of use and comprehension: 4.5
- Photo step ease of use and comprehension: 4.9

- Turning towards the shadow step ease of use and comprehension: 4.3
- Taking ten steps step ease of use and comprehension: 4.3
- Orientation after ten steps ease of use and comprehension: 4.2

7.4 Validation

In section 4.1.2 and 4.2.2 we defined the success criteria for the planned functionalities.

With the user tests described in the previous section, we were able to ascertain that the functionalities met the success criteria, as presented in Table 7.1.

Use Case	**Passes / Fails**
UC1	Passes
UC2	Passes
UC3	Passes
UC4	Passes
UC9	Passes
UC7	Passes
UC8	Passes

Table 7.1: Functionalities of the mobile app

Table 7.2 presents the functionalities of the portal and whether they fulfil or not the success criteria.

Use Case	**Passes / Fails**
UC1	Passes
UC2	Passes
UC3	Passes
UC4	Passes
UC12	Passes
UC13	Passes
UC14	Passes
UC15	Passes

Table 7.2: Functionalities of the portal

As presented in sections 6.1.2 and 6.2.2, the Angular security guidelines are being followed. The remaining security requirements, presented in Table 7.3 were validated in the unit tests previously described.

Designation	**Passes / Fails**
SEC1	Passes
SEC2	Passes
SEC3	Passes

Table 7.3: Security requrements validation

7.5 Summary

In this chapter we presented the tests performed to the system. These were functional tests, measurements to validate the non-functional requirements and field tests. The functional tests were unit tests.

The unit tests were performed on the mobile application and web portal. These cover the NgRx portion of both applications. Also, the tests validated the security requirements concerning to the data storage in the application and ensure the robustness of the solutions implemented.

The measurements performed verified that the applications fulfilled the success criteria of the non-functional requirements, with the exception of the image loading operation.

Lastly, the field tests performed ascertained the fulfilment of the success criteria for the application.

Chapter 8

Conclusion

In this chapter we start by presenting the main conclusions of the work done in this dissertation. Then, we finish with the suggestions of future work to be done.

8.1 Main Conclusions

The main objective of this dissertation was to implement the points of contact of the user with the Fireloc system, which were the mobile app and the web portal. These objectives were fully achieved, since both applications satisfy all the functional and non-functional requirements defined for the project. The mobile application is capable of reliably collecting fire reports with different types of positional data and photos and delivering them to the Fireloc servers; also, the web portal is capable of displaying data to the users, with the users able to search fire data stored in the server through diverse queries.

The use of Ionic greatly facilitated the development of the mobile app. With it, we were able to use web technologies, such as leaflet, to produce a working and complex interface in a short span of time. Ionic also helped in the generation of the android apk, providing the means to wrap the web application into a native application with native capabilities. Further benefits of using Ionic was the sharing of infrastructure, namely services and parts of the NgRx infrastructure, between the mobile app and the web application.

The use of Firebase to provide authentication in the web application proved to be an important asset. This allowed to quickly deploy in the web portal reliable and secure authentication, and user management functionalities, which were easily integrated in the backend server. However, Firebase has some edge cases that can hamper testing in the browser. This must be taken into consideration in further development.

NgRx proved to be a very useful and powerful state management framework. Prior experience in developing applications using Angular revealed some of the problems exposed in section 2.4.1, and was one of the main reasons to investigate a different method of handling state. While NgRx does require some aditional boilerplate code, when we consider the additional code we have to write (reducers, actions, selectors), this code is simpler to write and follows a clearly defined logic. This allowed the creation of complex and easily maintainable states, while keeping the components clean and pure.

For the development of the server, Django was of utmost use. When implementing the support API for the web portal, we relied heavily on the modules provided by Django, not just for the server infrastructure, but also for data manipulation. By providing modules

for GIS manipulation and geojson serialization, we were able to use a unified API for implementing all required operations. While there are many available and competitive options for developing server infrastructure, such as Flask or Nodejs, by providing these modules for handling geospatial data, Django proved to be the superior choice for the Fireloc project.

8.2 Future Work

While we achieve all the objectives proposed for this dissertation, we identify some opportunities for further development.

The next logical step of the mobile application will be the generation of the iOS native application. This can be accomplished using iOS building tools in conjunction with Ionic, similarly to the way the android apk was generated. We can also envisage further development and integration of new features. The development of the anonymous usage functionality is a promising concept from the point of view of privacy. While Fireloc takes steps in ensuring the safety of the data collected, some users may be deterred from using the mobile app by privacy questions. This functionality can boost the effectiveness of our system by increasing the number of users, which is in accordance with the philosophy of crowdsourcing. Another functionality that may be included in the mobile application is the integration of more authentication methods. Considering the use of google login as an example, we already have the logic available in the authentication module of the web application. Thanks to the shared technologies and architecture, it can be easily integrated with the mobile application. Should Firebase be used as an authentication provider, more methods can easily be added.

When considering the web application, a next step would be synchronizing the authentication APIs used in both the mobile and the web applications to allow a seamless integration of the data flow — mobile application/server/web portal. Further steps may include adding more map modes to view different kinds of data, such as critical infrastructure in the area surrounding active fires. This data should only be available to the authorities for security reasons. Another feature to add to the portal could be the inclusion of statistics, a feature useful to understand the impact of the fires and the role of the entities in their prevention and detection.

To support both the mobile application and the web portal, another useful feature would be to include the ability to reduce image sizes in the server. As already discussed in section 7.2.2, images take a long time to load from the server and for displaying purposes may not require their original size and detail. Reducing their size after processing will allow to reduce response times, enhancing the user experience in the mobile app.

References

[1] *16 metrics to ensure mobile app success*. URL: https://www.appdynamics.com/media/uploaded-files/1432066155/white-paper-16-metrics-every-mobile-team-should-monitor.pdf (visited on 08/09/2021).

[2] *angular/fire*. URL: https://www.npmjs.com/package/@angular/fire (visited on 06/06/2021).

[3] *arcGIS*. URL: https://www.arcgis.com/index.html (visited on 07/29/2021).

[4] *Áreas Ardidas Oficial*. URL: https://www.portugal.gov.pt/pt/gc22/comunicacao/comunicado?i=area-ardida-ficou-em-metade-da-media-dos-ultimos-10-anos (visited on 09/02/2021).

[5] *C4 Model*. URL: https://c4model.com/#coreDiagrams (visited on 12/20/2020).

[6] *Carta Administrativa Oficial de Portugal*. URL: https://www.dgterritorio.gov.pt/cartografia/cartografia-tematica/caop (visited on 04/01/2021).

[7] *Django*. URL: https://www.djangoproject.com/ (visited on 12/11/2020).

[8] *Docker*. URL: https://www.docker.com/ (visited on 12/22/2020).

[9] *Firebase Privacy Policy*. URL: https://firebase.google.com/support/privacy (visited on 08/30/2021).

[10] *Fireloc Website Homepage*. URL: https://fireloc.org/ (visited on 08/20/2021).

[11] *Flutter*. URL: https://flutter.dev/ (visited on 11/02/2019).

[12] *Flutter Docs*. URL: https://flutter.dev/docs/resources/technical-overview (visited on 11/02/2019).

[13] *Flutter Docs*. URL: https://flutter.dev/docs/deployment/web (visited on 11/02/2019).

[14] *Fotoquest Go*. URL: https://fotoquest-go.org/en/ (visited on 12/22/2020).

[15] *GeoNode*. URL: https://geonode.org/ (visited on 12/22/2020).

[16] *GeoServer*. URL: http://geoserver.org/ (visited on 12/22/2020).

[17] *Github: Appcelerator Tituanium*. URL: https://github.com/appcelerator/titanium_mobile (visited on 11/01/2019).

[18] *Github: Flutter*. URL: https://github.com/flutter/flutter (visited on 11/01/2019).

[19] *Github: Framework7*. URL: https://github.com/framework7io/framework7 (visited on 11/01/2019).

[20] *Github: Ionic*. URL: https://github.com/ionic-team/ionic (visited on 11/01/2019).

[21] *Github: JQuery Mobile*. URL: https://github.com/jquery/jquery-mobile (visited on 11/01/2019).

[22] *Github: Leaflet*. URL: https://github.com/Leaflet/Leaflet (visited on 12/05/2020).

[23] *Github: Mapbox*. URL: https://github.com/mapbox/mapbox-gl-js (visited on 12/05/2020).

[24] *Github: NativeScript*. URL: https://github.com/NativeScript/NativeScript (visited on 11/01/2019).

[25] *Github: Openlayers*. URL: https://github.com/openlayers/openlayers (visited on 12/05/2020).

[26] *Github: PhoneGap*. URL: https://github.com/phonegap/phonegap-app-developer (visited on 11/01/2019).

[27] *Github: React-Native*. URL: https://github.com/facebook/react-native (visited on 11/01/2019).

[28] *Github: Xamarin*. URL: https://github.com/xamarin/Xamarin.Forms (visited on 11/01/2019).

[29] *Google Maps*. URL: https://developers.google.com/maps/documentation/javascript/overview#Inline (visited on 12/05/2020).

[30] *Google Maps Pricing*. URL: https://cloud.google.com/maps-platform/pricing/?_ga=2.219981017.1784570227.1607461334-1609716321.1607461334 (visited on 12/05/2020).

[31] *Hackernoon*. URL: https://hackernoon.com/top-10-best-mobile-app-development-frameworks-in-2019-612b95cf930f (visited on 11/01/2019).

[32] *idToken verification instructions*. URL: https://firebase.google.com/docs/auth/admin/verify-id-tokens (visited on 07/04/2021).

[33] *Iflexion*. URL: https://www.iflexion.com/blog/top-hybrid-mobile-app-frameworks (visited on 11/01/2019).

[34] *IPMA Earthquakes*. URL: http://www.ipma.pt/pt/geofisica/sismicidade/ (visited on 12/22/2020).

[35] *IPMA Fires*. URL: http://www.ipma.pt/pt/riscoincendio/rcm.pt/ (visited on 12/22/2020).

[36] *Jasmine*. URL: https://jasmine.github.io/ (visited on 07/29/2021).

[37] *Karma*. URL: https://karma-runner.github.io/latest/index.html (visited on 07/29/2021).

[38] *Leaflet*. URL: https://leafletjs.com/ (visited on 12/05/2020).

[39] *Mapbox*. URL: https://www.mapbox.com/ (visited on 12/05/2020).

[40] *Mapbox Pricing*. URL: https://www.mapbox.com/pricing/ (visited on 12/05/2020).

[41] *Medium*. URL: https://medium.com/better-programming/best-hybrid-apps-frameworks-in-2019-f5120a35fac2 (visited on 11/01/2019).

[42] *Medium*. URL: https://medium.com/@rathod.atman/top-5-hybrid-mobile-applications-development-frameworks-in-2019-df16cdb10436 (visited on 11/01/2019).

[43] *NASA Fires*. URL: https://firms.modaps.eosdis.nasa.gov/map/#d:2020-12-20..2020-12-21;@-2.9,38.7,6z (visited on 12/22/2020).

[44] *NgRx*. URL: https://ngrx.io/docs (visited on 06/03/2021).

[45] *Nielsen Usability Heuristics*. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (visited on 07/29/2021).

[46] *Nomenclatura das Unidades Territoriais para Fins Estatísticos*. URL: https://www.pordata.pt/O+que+sao+NUTS (visited on 07/18/2021).

[47] *Openlayers*. URL: https://openlayers.org/ (visited on 12/05/2020).

[48] *OSGeo-GeoNode*. URL: https://www.osgeo.org/projects/geonode/ (visited on 12/22/2020).

[49] *Security - Angular*. URL: https://angular.io/guide/security (visited on 08/16/2021).

[50] *vue2-google-maps*. URL: https://www.npmjs.com/package/vue2-google-maps (visited on 12/05/2020).

[51] *WCS*. URL: https://www.opengeospatial.org/standards/wcs (visited on 12/22/2020).

[52] *Websoptimization*. URL: https://www.websoptimization.com/blog/hybrid-mobile-app-frameworks/ (visited on 11/01/2019).

[53] *WFS*. URL: https://www.opengeospatial.org/standards/wfs (visited on 12/22/2020).

[54] *WMS*. URL: https://www.opengeospatial.org/standards/wms (visited on 12/22/2020).

Appendices

Appendix A - Use Cases

UC1: Login	
Last Revision:	November, 21st 2020
Level:	User-goal
Priority:	Must-Have
Primary Actor:	Unauthenticated User

Stakeholders and Interests:

- Professionals and Volunteers

Preconditions:

- The user has opened the app

Postconditions (success guarantee):

- The User is logged in
- The Unauthenticated User becomes an Authenticated User

Main Success Scenario:

1. The user selects 'Login'
2. The user inserts his credentials in the appropriate fields
3. The system authenticates the user
4. Redirect to the Main Page

Extensions (Alternative Flows):

1.a The user selects 'Login with Google Account':

1. The system opens the google pop-up
2. The user follows the steps of Google Authentication

4.a Invalid login data:

1. The system verifies that the credentials don't match
2. The system displays an error message
3. The user remains in the Opening Page (Step 1)

4.b Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message
3. The user remains in the Opening Page (Step 1)

UC2: User Registration	
Last Revision:	November, 21st 2020
Level:	User-goal
Priority:	Must-Have
Primary Actor:	Unauthenticated User

Stakeholders and Interests:

- Professionals and Volunteers

Preconditions:

- The user has opened the app

Postconditions (success guarantee):

- The User is registered and is logged in
- The Unauthenticated User becomes Authenticated User

Main Success Scenario:

1. The user selects 'Register'
2. The system redirects to the Create Account Page
3. The user inserts his credentials in the appropriate fields
4. The user selects "Continue"
5. The system verifies the credentials inserted
6. The user accepts Terms and Conditions
7. The user selects "Continue"
8. The user inserts the optional data
9. The user clicks "Register"
10. The system creates the new user
11. The system redirects to the Main Page

Extensions (Alternative Flows):

5.a The user inserts incorrect credentials:

1. The system verifies that the credentials don't match the defined criteria
2. The system displays an error message
3. The system highlights the fields with those credentials

5.b The user doesn't fill all fields:

1. The system verifies a/some field(s) is/are missing
2. The system displays an error message
3. The system highlights the missing fields

6.a User doesn't check a box to agree to the Terms and Conditions:

1. The system verifies the user hasn't agreed to the Terms and Conditions
2. The system displays a warning emphasizing that acceptance of T&C is mandatory and that the account can not be created without such an agreement
3. The system highlights the field for Terms and Conditions

9.a The user inserts credentials already used by other users:

1. The system verifies that the credentials match others already in use
2. The system displays an error message
3. The system highlights the fields with those credentials
4. Return to step 3

9.b Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message
3. The system redirects to the Opening Page

UC3: Password Recovery	
Last Revision:	November, 21st 2020
Level:	User-goal
Priority:	Nice-To-Have
Primary Actor:	Unauthenticated User

Stakeholders and Interests:
- Professionals and Volunteers

Preconditions:
- The User has opened the app

Postconditions (success guarantee):
- The User recovers the password

Main Success Scenario:

1. The user selects "Recover Password"
2. System redirects to the Recover Password page
3. The user fills the appropriate recovery information
4. The system validates the recovery information
5. The system sends a link to the user's email
6. The user clicks the link
7. The system redirects to the Change Password page
8. The user inserts the new password
9. The system changes the user's password
10. The system invalidates the recovery link
11. The system displays a success message
12. Redirect to the Opening Page

Extensions (Alternative Flows):

4.a The user doesn't fill all fields:

1. The system verifies a/some field(s) is/are missing
2. The system displays an error message
3. The system highlights the missing fields

4.b Invalid recovery data:

1. The system verifies the credentials don't match
2. The system displays an error message
3. The system redirects the user to the Opening Page (Step 1)

4.b, 9.a Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message
3. The system redirects to the Opening Page (Step 1)

UC4: Delete Account	
Last Revision:	November, 22nd 2020
Level:	User-goal
Priority:	Should-Have
Primary Actor:	Authenticated User

Stakeholders and Interests:

- Professionals and Volunteers

Preconditions:

- The User must be logged in

Postconditions (success guarantee):

- The User logs out
- The User's account is deleted
- The Authenticated User becomes Unauthenticated User

Main Success Scenario:

1. The User selects "Change Settings"
2. The system redirects to the Settings Page
3. The User selects "Delete Account"
4. The system redirects to the Delete Account Page
5. The User inserts the appropriate credentials
6. The User selects "Continue"
7. The system validates the credentials
8. The system displays a confirmation message
9. The User confirms
10. The system redirects the user to the Opening Page

Extensions (Alternative Flows):

5.a The user doesn't fill all the necessary fields:

1. The system marks the empty fields

6.a The user doesn't fill all the necessary fields:

1. The system marks the empty fields
2. The User remains in the same page

7.a The User inserted wrong credentials:

1. The system displays an error message
2. The User remains in the same page

7.b, 9.b Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message
3. The system redirects to the Main Page

9.a The User cancels:

1. The system discards the request
2. The system displays a confirmation message
3. The system redirects to the Main Page

UC5: Logout	
Last Revision:	November, 22nd 2020
Level:	User-goal
Priority:	Must-Have
Primary Actor:	Authenticated User

Stakeholders and Interests:

- Professionals and Volunteers

Preconditions:

- The User must be logged in

Postconditions (success guarantee):

- The User logs out
- The Authenticated User becomes Unauthenticated User

Main Success Scenario:

1. The use case starts when the user is logged in and wants to exit the application
2. The user selects "Settings"
3. The system redirects to the Settings Page
4. The user selects "Logout"
5. The system redirects to the Opening Page

UC6: Request Personal Data	
Last Revision:	January, 21st 2021
Level:	User-goal
Priority:	Should-Have
Primary Actor:	Authenticated User

Stakeholders and Interests:
- Professionals and Volunteers

Preconditions:
- The user is logged in

Postconditions (success guarantee):
- The User receives his data

Main Success Scenario:

1. The user selects 'A minha conta'
2. The system redirects to the settings page
3. The user selects 'Pedir dados pessoais'
4. The system sends the request to the server
5. Redirect to the Main Page
6. The user receives the data through his email sometime later

Extensions (Alternative Flows):

4.a Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message
3. The system redirects to the Main Page

UC7: Report Fire	
Last Revision:	November, 23rd 2020
Level:	User-goal
Priority:	Must-Have
Primary Actor:	Authenticated User, Anonymous User

Stakeholders and Interests:

- Professionals and Volunteers

Preconditions:

- The User must be logged in
- The user must have his location turned on

Postconditions (success guarantee):

- The User reported the fire
- The User's contribution is marked

Main Success Scenario:

1. The User selects "Report Fire"
2. The system redirects to the Report Fire Page
3. The User verifies his location on the map
4. The User selects "Continue"
5. The User marks his orientation
6. The User selects "Continue"
7. The system opens the User's Camera
8. The User takes the photo of the fire
9. The User confirms the photo
10. The system shows the Continue Contribution page
11. The User selects "Finish Report"
12. The system collects and sends the data to the backend server
13. The system displays a success message
14. The system redirects to the Main Page

Extensions (Alternative Flows):

3.a, 5.a, 9.a The User cancels:

1. The system discards the request
2. The system redirects to the Main Page

3.b The User selects "go back":

1. Follow steps of extension 3.a

5.b The User selects "go back":

1. The system discards the data from this step
2. Go to step 3

9.b The User selects "go back":

1. The system discards the data from this step
2. Go to step 5

11.a The User selects "Continue Contributing":

1. Go to UC8

12.a Internet is unavailable:

1. The system can't connect with the server
2. The system redirects to the Main Page

UC8: Complete Report Fire	
Last Revision:	November, 23rd 2020
Level:	User-goal
Priority:	Must-Have
Primary Actor:	Authenticated User, Anonymous User

Stakeholders and Interests:
- Professionals and Volunteers

Preconditions:
- The User must have followed extension 11.a of the Report Fire UC7

Postconditions (success guarantee):
- The User reported the fire
- The User's contribution is marked

Main Success Scenario:

1. The System displays the instructions for the next steps
2. The User follows the steps
3. The User selects "Continue"
4. The User marks his position again
5. The User selects "Finish Report"
6. The system collects and sends the data to the backend server
7. The system displays a success message
8. The system redirects to the Main Page

Extensions (Alternative Flows):

1.a, 2.a, 4.a The User cancels:

1. The system discards the request
2. The system redirects to the Main Page

4.b The User selects "go back":

1. The system discards the data from this step
2. Go to step 2

6.a Internet is unavailable:

1. The system can't connect with the server
2. The system redirects to the Main Page

UC9: View Contribution	
Last Revision:	November, 23rd 2020
Level:	User-goal
Priority:	Must-Have
Primary Actor:	Authenticated User

Stakeholders and Interests:

- Professionals and Volunteers

Preconditions:

- The User must be logged in

Main Success Scenario:

1. The User selects "My Contributions"
2. The system redirects to the My Contributions Page
3. The system loads the contributions of the User
4. The User selects a contribution
5. The system redirects to the View Single Contribution page
6. The system fetches the details of the new contribution from the server
7. The system displays the details

Extensions (Alternative Flows):

3.a, 6.a Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message

3.b No contributions found:

1. The system displays a message
2. End of Use Case

UC10: Change Password	
Last Revision:	January, 21st 2021
Level:	User-goal
Priority:	Should-Have
Primary Actor:	Authenticated User

Stakeholders and Interests:
- Professionals and Volunteers

Preconditions:
- The user is logged in

Postconditions (success guarantee):
- The User changes his password

Main Success Scenario:

1. The user selects 'A minha conta'
2. The system redirects to the settings page
3. The user selects 'Mudar Password'
4. The user inserts his new password
5. The user inserts the new password again
6. The user clicks 'continue'
7. The system sends the request to the server
8. Redirect to the Main Page

Extensions (Alternative Flows):

4.a, 5.a Error in password string:

1. The system displays an error message
2. The system highlights the field
3. The system clears the field

6.a Passwords don't match:

1. The system displays an error message
2. The system clears both fields
3. Return to step 4

7.a Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message

something

UC11: Change Username	
Last Revision:	January, 21st 2021
Level:	User-goal
Priority:	Should-Have
Primary Actor:	Authenticated User

Stakeholders and Interests:
- Professionals and Volunteers

Preconditions:
- The user is logged in

Postconditions (success guarantee):
- The User changes his username

Main Success Scenario:

1. The user selects 'A minha conta'
2. The system redirects to the settings page
3. The user selects 'Mudar Username'
4. The user inserts his new username
5. The user clicks 'continue'
6. The system sends the request to the server
7. Redirect to the Main Page

Extensions (Alternative Flows):

4.a Error in username string:

1. The system displays an error message
2. The system highlights the field

7.a Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message

7.b Username already taken:

1. The system displays an error message
2. Return to step 4

UC12: View Occurring Fire	
Last Revision:	November, 2nd 2020
Level:	User-goal
Priority:	Must-Have
Primary Actor:	Authenticated User

Stakeholders and Interests:
- Professionals and Volunteers

Preconditions:
- The user is authenticated
- Fire event is shown on the map

Postconditions (success guarantee):
- The User views the relevant data
- A fire event is selected

Main Success Scenario:

1. The user selects the marker corresponding to the appropriate fire
2. The system is queried for the user contributions corresponding to the fire
3. The contribution panel displays the user contributions corresponding to the fire
4. The map displays a popup with relevant information

Extensions (Alternative Flows):

2.a Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message on the panel

UC13: View Occurring Fire Detail	
Last Revision:	November, 2nd 2020
Level:	User-goal
Priority:	Must-Have
Primary Actor:	Authenticated User

Stakeholders and Interests:

- Professionals and Volunteers

Preconditions:

- The user is authenticated
- Fire event is shown on the map
- A fire event has been selected

Postconditions (success guarantee):

- The User views the relevant data

Main Success Scenario:

1. The user selects 'Ver Detalhes do Fogo'
2. The system loads the relevant data from the server
3. The map zooms to the location of the fire
4. The map displays the projected area of the fire given by the system
5. The map displays the location of the relevant user contributions

Extensions (Alternative Flows):

2.a, 3.a Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message

UC14: Search Location	
Last Revision:	November, 15th 2020
Level:	User-goal
Priority:	Must-Have
Primary Actor:	Authenticated User

Stakeholders and Interests:

- Professionals and Volunteers

Preconditions:

- The user is authenticated

Postconditions (success guarantee):

- Map Data is updated
- Base Location is selected

Main Success Scenario:

1. The user selects the content for the 'Distrito' field
2. The user selects the content for the 'Conselho' field
3. The user clicks 'Pesquisar'
4. The system loads the relevant data
5. The map zooms to the selected location

Extensions (Alternative Flows):

1.a, 2.a No content selected:

1. The system displays an error message and highlights the missing field

3.b No content selected:

1. The system displays an error message and highlights the missing fields
2. Return to steps 1 or 2 depending on the missing fields

4.a Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message

UC15: Search Timeframe

Last Revision:	November, 2nd 2020
Level:	User-goal
Priority:	Must-Have
Primary Actor:	Authenticated User

Stakeholders and Interests:
- Professionals and Volunteers

Preconditions:
- The user is authenticated

Postconditions (success guarantee):
- Map Data is updated
- Base Time is selected

Main Success Scenario:

1. The user uses the time slider to select a given date
2. The system loads the relevant data from the server
3. The map updates with the fires active in the given time

Extensions (Alternative Flows):

2.a Internet is unavailable:

1. The system can't connect with the server
2. The system displays an error message

Appendix B - Contribution Process Specification

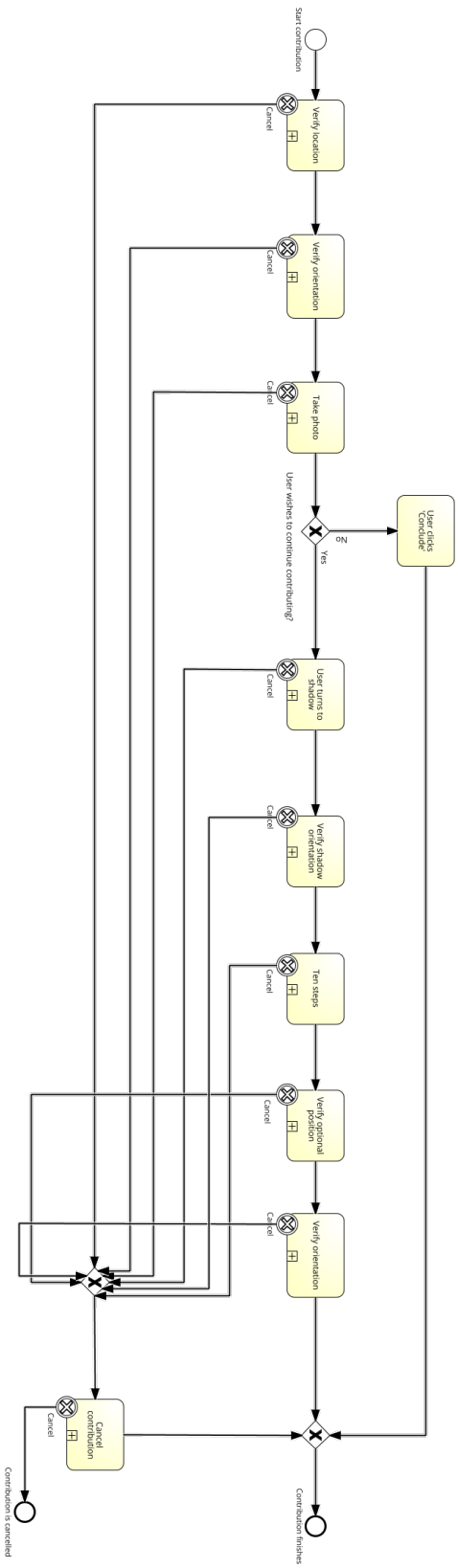

Figure 1: Full Contribution Process

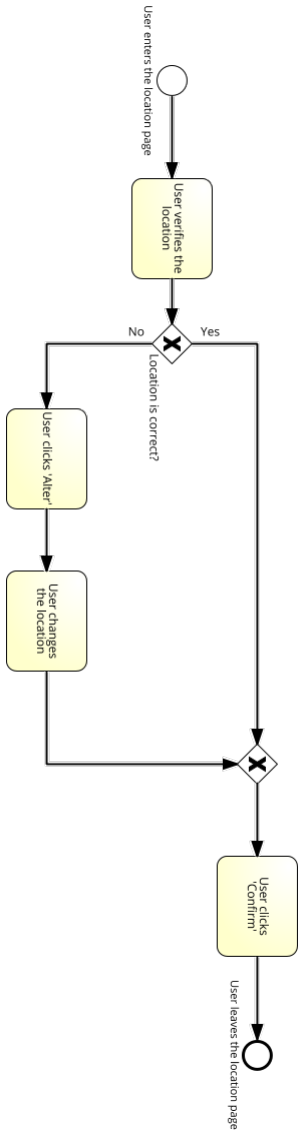

Figure 2: Verify Location Process

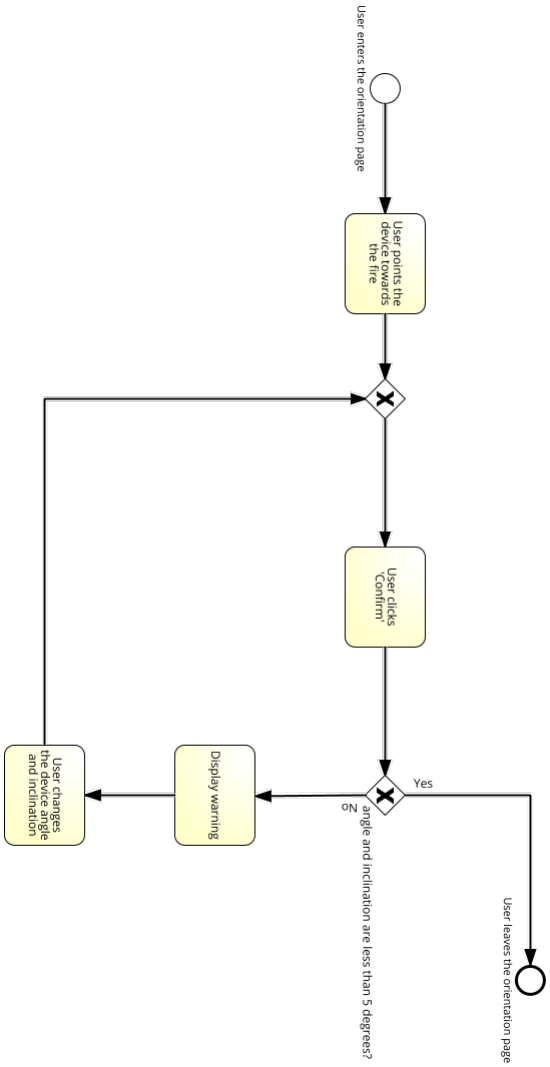

Figure 3: Verify Orientation Process

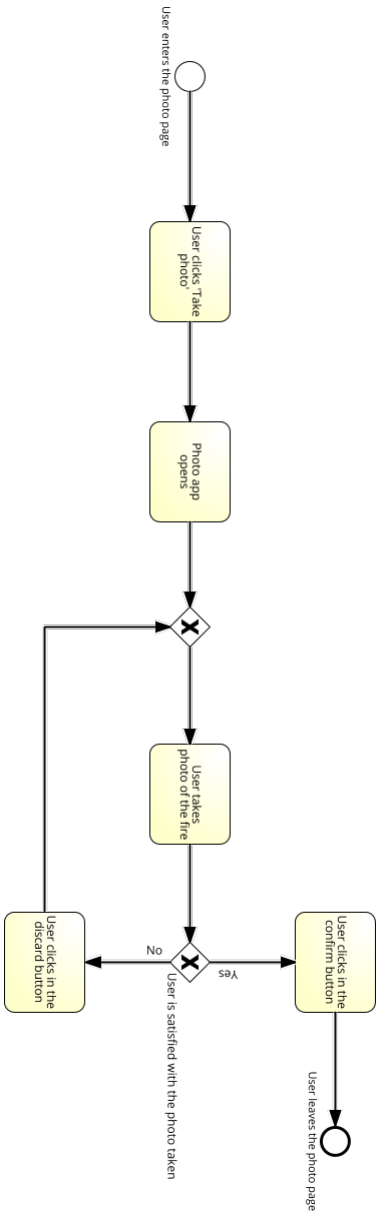

Figure 4: Take Photo Process

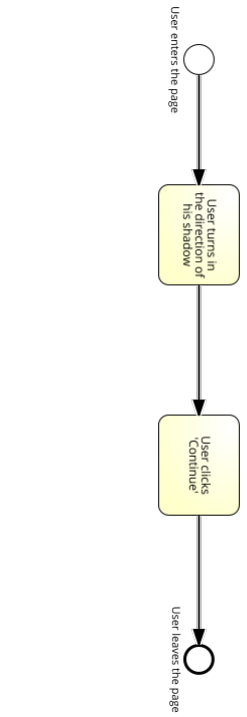

Figure 5: Turn to Shadow Step

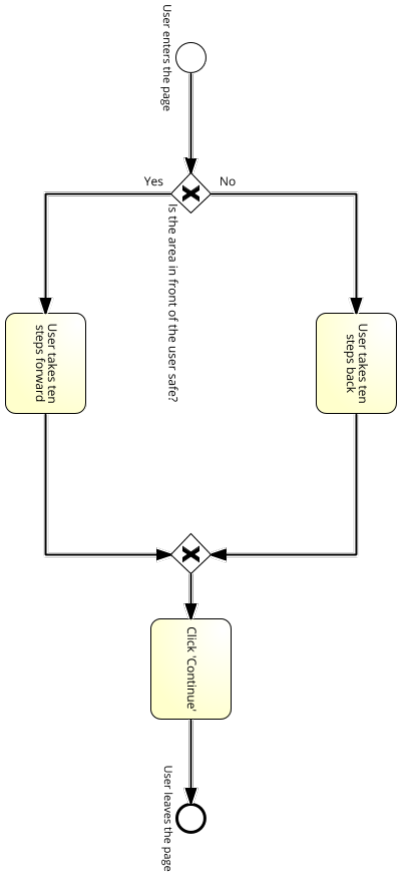

Figure 6: Take Ten Steps Step

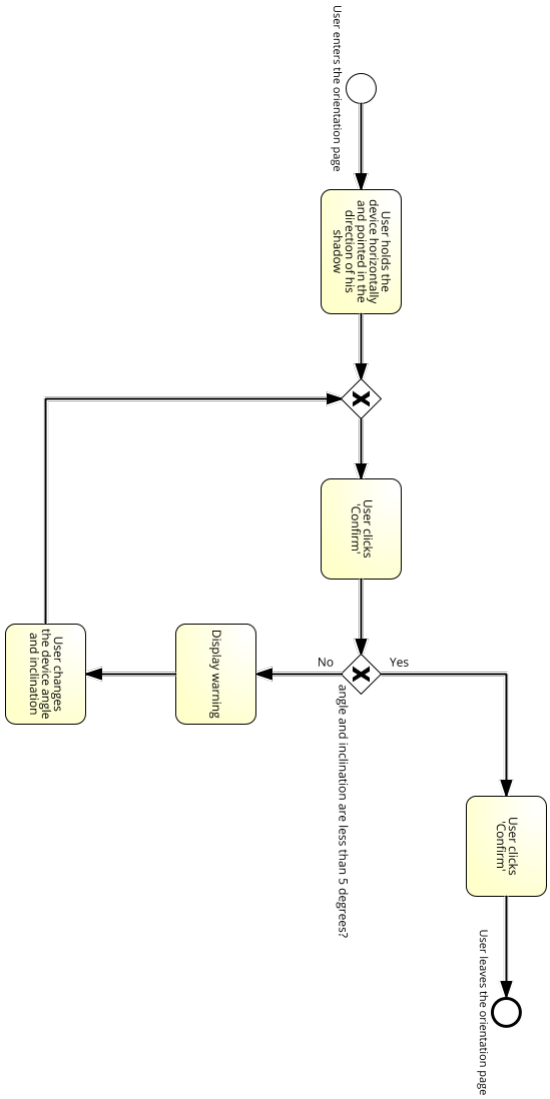

Figure 7: Verify Shadow Step

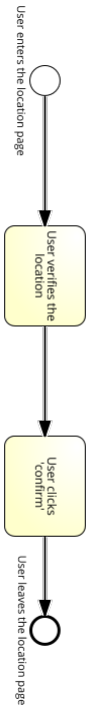

Figure 8: Verify Optional Position Step

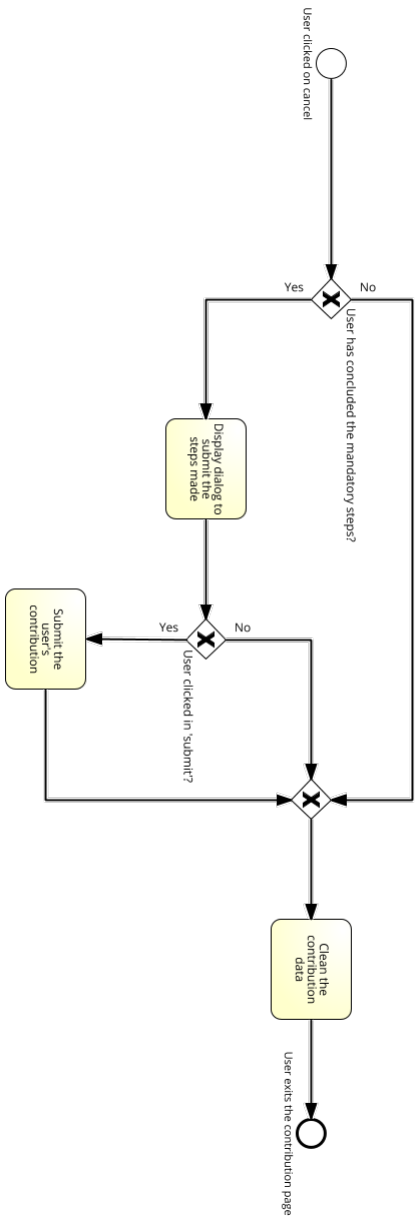

Figure 9: Cancel Contribution Step