Western Norway University of Applied Sciences
Department of Computer science, Electrical engineering and Mathematical sciences

Development and Evaluation of a Software System for Fire Risk Prediction

Master’s Thesis

Authors: Eivind Dagsland Halderaker, Andreas Evjenth

Supervisors: Lars Michael Kristensen, Ruben Dobler Strand

June 1, 2021

Abstract

Fire risks is a broad research topic that is important to many different stakeholders, from fire departments and municipalities, to homeowners. Reducing risks of fires includes not only infrastructure to extinguish flames, such as fire brigades, but also warning systems to alert of fires as early as possible, like smoke alarms. Risks can be further mitigated, by having a system to detect locations with high risk, both at present time but also predicted days ahead. It is this system and its concepts that is the core of this thesis. Stakeholders can be alerted of the high risk areas and take steps accordingly; Fire brigades can allocate more resources at a given place and time, and the general public can reduce or avoid risky behaviour, such as burning garden waste.

The DYNAMIC research project has developed a model for estimating relative indoor humidity based on outdoor weather conditions. The indoor humidity can in turn be used to determine the time to flash-over after an ignition has occurred. This thesis centers around implementing and extending the usability of this model. A system of micro-services, hosted in the cloud, makes the fire risks available for stakeholders. They can add locations of interest to the system, see current and predicted risks in a heat-map imposed on a geographical map, and subscribe to be notified if specific locations have high risks.

Historic weather data from the last 5 days are used to model the current risk for a location. By extending the data with weather forecasts, the predicted fire risks are also modelled. Experiments have been conducted to demonstrate that the predictions yields accurate fire risks. During February 2021, fire risks for 74 locations, spread across Norway, have been continuously collected in order to validate the system.

Acknowledgements

We would like to thank Lars Michael Kristensen and Ruben Dobler Strand for supervising and guiding us through this thesis. The insights about the DYNAMIC research project and the specifics of fire risks and risk modelling has provided great value to our work.

`Andreas Evjenth, Eivind Dagsland Halderaker`

June 1, 2021

Contents

1 Introduction **2**
1.1 The DYNAMIC Research Project . . . 2
1.1.1 Fire risk prediction model . . . 3
1.2 Research Questions . . . 4
1.3 Research Method . . . 6
1.4 Thesis structure . . . 7

2 Background and Related Work **9**
2.1 Fire Risk Prediction . . . 9
2.1.1 Partial Risk Indicies . . . 10
2.1.2 Chandler Burning Index . . . 10
2.2 Weather Data Sources . . . 11
2.2.1 The Norwegian Meteorological Institute . . . 12
2.2.2 Netatmo . . . 21
2.2.3 StormGeo . . . 23
2.2.4 The European Centre for Weather Forecasts . . . 23
2.2.5 OpenWeather . . . 25
2.2.6 Comparing Weather Sources . . . 25
2.3 Software Architecture . . . 25
2.3.1 Microservices . . . 26
2.3.2 RESTful services . . . 26
2.3.3 Spring Boot . . . 27
2.3.4 Cloud Hosting . . . 27

3 Design and System Requirements **28**
3.1 System Requirements . . . 28

3.2 Deployment and DevOps . . . 30
3.2.1 Fire Risk Model . . . 31
3.2.2 Fire Risk Services and Data Harvesting Service . . . 31
3.2.3 Middleware . . . 31
3.2.4 Frontend . . . 32

4 Fire Risk Model (FRM) 33
4.1 System Overview . . . 34
4.2 Mathematical Formulas and Implementation . . . 34
4.2.1 Modelling Relative Humidity . . . 34
4.2.2 Time To Flash-over . . . 41
4.2.3 Fire Risk Factor . . . 43
4.2.4 Interpolation . . . 44
4.3 Deployment and DevOps . . . 44

5 Fire Risk Services (FRS) 46
5.1 System Overview . . . 46
5.2 Endpoints . . . 47
5.2.1 Heartbeat . . . 48
5.2.2 Add locations . . . 48
5.2.3 Firerisk . . . 48
5.2.4 Firerisk factor . . . 49
5.3 Database . . . 49
5.4 Deployment and DevOps . . . 50

6 Data Harvesting Service (DHS) 51
6.1 System Overview . . . 51
6.2 Endpoints . . . 53
6.2.1 Measurements . . . 53
6.2.2 Location exists . . . 54
6.2.3 Add location . . . 55
6.2.4 Historical Measurements . . . 56
6.3 Scheduled Services . . . 56
6.4 Database . . . 57
6.5 Deployment and DevOps . . . 58

7 Middleware **60**
7.1 System Overview . . . 60
7.2 Endpoints . . . 61
7.3 Scheduled Services . . . 65
7.4 Deployment and DevOps . . . 65
7.5 Database . . . 66
7.6 Testing . . . 68

8 Front-end Single Page Web Application **70**
8.1 Project Structure . . . 70
8.1.1 React . . . 71
8.1.2 TypeScript . . . 72
8.1.3 Leaflet & React Leaflet . . . 73
8.2 Features . . . 74
8.3 Fetching Data from the Middleware . . . 76
8.4 DevOps . . . 77

9 Experiments and Evaluation **79**
9.1 Locations . . . 79
9.2 Experiment 1: Predicting Time to Flashover . . . 81
9.2.1 Collecting Fire Risks . . . 81
9.2.2 Analysing the Data . . . 82
9.2.3 Largest difference . . . 85
9.2.4 Prediction deviation . . . 86
9.3 Experiment 2: TTF Weather Data Sources . . . 87
9.3.1 Data . . . 87
9.3.2 Analysing the Data . . . 89
9.4 Experiment 3: Historical Fire Risk 2013 - 2021 . . . 89
9.4.1 Data . . . 90
9.4.2 Analysing the Data . . . 92
9.5 Experiment 4: Winter of 2020/2021 . . . 92
9.5.1 Analysing the Data . . . 94
9.6 Experiment 5: Data Storage and Computation Time . . . 96
9.6.1 Processing utilization . . . 96
9.6.2 System test . . . 96

9.7 Conclusion 99

10 Conclusion and Future Work **101**

10.1 Conclusion on experiments 101

10.2 Research Questions Revisited 103

10.3 Threats to validity 104

10.4 Discussion 105

10.5 Future Work 106

Bibliography **114**

Appendices **119**

A Locations used for Experiments **121**

B URL to the Source Code of implementations **123**

Chapter 1

Introduction

Fire has for a long time been a resource for humanity, but it also poses great risks. Conflagrations and large fires are a threat to human lives and wildlife, and can lead to great monetary damage as well. In Norway, wooden houses is prevalent, in existing structures, as well as new buildings. Reducing fire risk in and near houses is therefore a worthwhile effort.

1.1 The DYNAMIC Research Project

DYNAMIC [37] is an interdisciplinary research project focusing on fire risk assessment in wooden houses, heathland, and the Wildland-Urban Interface (WUI) zone. The project places people at the center, and aims to protect against threats involving fires. The tasks of the DYNAMIC Project are divided into four Work Packages (WPs):

WP1: *Risk modelling and warning systems.* The project has developed [23] and implemented [44] a model for predicting fire risk in wooden houses in cold climates. The model uses applied physics for modelling a generic house in an area. DYNAMIC aims to create similar models for heathland WUI fire risk and conflagration risk.

WP2: *Adaptive management of Calluna heathland to mitigate WUI fire risk.* Calluna heathland is a cultural landscape common along the European west coast from

Protugal to Norway. As it is not widespread globally, there are currently no proper approaches to assess fire risk in these landscapes. WP2 emphasizes collaboration with prescribed burner groups to help them mitigate the risk of accidental larger fires, and by studying their organisational structure.

WP3: *Risk-based emergency planning and dimensioning.* By combining the results from WP1 and WP2, one can develop applications for fire risk prediction to be used by emergency services. One such application can inform fire brigades of local high fire disaster areas, another can be developed for fire fighters to conduct virtual reality training scenarios.

WP4: *Synthesis, knowledge transfer, stakeholder involvement and communication.* The DYNAMIC Project aims to increase awareness of fire risk by involving stakeholders and organize it with an eight member advisory group. These stakeholders consist of researchers, fire fighters and their officers, farmers, policy makers, and administrators.

This thesis contributes to WP 1 and WP 3 by continuing the work of the fire risk model and expanding its features, by developing an early warning fire risk system delivered as a cloud-based solution.

1.1.1 Fire risk prediction model

In 2019, Stokkenes [44] implemented Log's model for fire risk [23]. The model calculates Time To Flash-over (TTF) in a single wooden structure. The term flash-over has no set definition in the literature, but one commonly used definition is: "The transition from a localized fire to the general conflagration within the compartment when all fuel surfaces are burning." [9]

As the fire risk in these structures is a result of the relative humidity in wood, the TTF is in turn affected by the humidity in the air [23] [21]. Because of this, outdoor relative humidity and outdoor air temperature are the two weather elements supplied as data to the model.

Stokkenes [44] developed a prototype and conducted an initial validation based on weather data collected during the winter of 2018/2019. The prototype application consists of three microservices. One service is responsible for harvesting the weather data, another service handles the fire risk calculation, and the third service acts as a controller by handling the communication between external services and the users, and the other two services. The three microservices communicate via REST APIs.

The fire risk prediction model was evaluated by collecting data from four locations: Bergen, Haugesund, Gjøvik and Lærdal from December 2018 to April 2019. This data was then used to compute results from the model to see how the Time To Flash-over varied during the season. The data harvesting service used APIs from The Norwegian Meteorological Institute (MET) and two Netatmo stations as sources of data. Stokkenes [44] used Netatmo stations to validate the fire risk model. These Netatmo stations are consumer grade, in contrast to The Norwegian Meteorological Institute's quality assured measurements [6].

Another part of evaluation was to compare Time To Flash-over based on forecasts to ones based on historical data. In addition, the fire risk prediction model can be calibrated using a few days of weather data before starting the model on other data. As such, Stokkenes compared how the model calculates fire risk based on forecast data, after being calibrated using a measurements, to measurements from the the same period. The last experiment calculated Time To Flash-over during historical fires, Lærdal at 18th of January 2014 and Kongsberg at 24th of December 2017. By collecting the measured weather data from the days leading up to the fires, the model expectedly returned low Time To Flash-over, around 4 minutes TTF leading up to both fires.

1.2 Research Questions

As described in section 1.1, the proposed research questions aims at contributing to WP 1 and WP 3 in the DYNAMIC research project. The research questions proposed below for this thesis are directed towards this effort. Below we list and discuss the research questions underlying this thesis.

RQ1: How can different weather services be combined to obtain one unified data service, that can provide weather measurements and forecasts for the fire risk model?

RQ2: How can a software architecture that implements a complex fire risk notification system composed of multiple independent cloud-hosted services be designed?

RQ3: Can the fire risk model be combined with weather forecast data to predict accurate fire risk indications?

RQ1

There are multiple considerations when utilizing weather data for fire risk indications. First, different weather services present their data in different formats, which needs to be converted prior to be input into the model. It may be available by download at a website, REST APIs, or GraphQL APIs [17], and needs to be parsed and converted into a single format used by the application. Another consideration is the type of weather data provided, a service may offer forecasts, measurements from a station, or interpolated data. The different types of data must be distingushed and combined correctly, in order to produce desired output. When collecting from specific stations, the distribution of these is an important factor as the application should calculate fire risk based on the desired location. Weather data is a broad term, and as such, not all weather data services provide the necessary weather parameters needed for fire risk prediction. Further, not all services provide their data for free, and there can be limitations on how often data can be requested. The update frequency of a service, and whether or not previous measurements or forecasts are available is also aspects that needs to be considered.

RQ2

A robust design and software architecture is vital when designing complex systems. The system has multiple requirements and should preferably be hosted as a cloud service. In general, the system shall compute fire risks, collect weather data and notify subscribers. It can be developed as one monolith or several microservices, and each application needs to be configured and hosted in a suitable cloud service.

RQ3

The fire risk indications can provide great value if it produces accurate fire risk prediction for the near-future. By using forecast data as input to the model this can be achieved and compared to fire risk indications computed from measurements. Knowing how the fire risk indications will be in the coming days allows stakeholders to initiate proactive measures, as to mitigate the upcoming risk, be it fire brigades allocating crew or public authorities notifying the general public of high risk periods.

1.3 Research Method

In this thesis, we propose Design Science [52] as the foundation for the research methodology. The guidelines proposed by Hevner et al. [19], serves as a solid framework for iteratively developing and evaluating an artefact that can be used to answer the research questions. The artefact is required to answer all of the questions, but the rigorous evaluation of it is especially important in RQ3 which aims at extending the fire risk model in the DYNAMIC research project.

The seven guidelines of design science research and how they are related to this thesis is as follows:

- **Design as an Artefact:** In this thesis, we develop and implement a fire risk notification system that will be used to answer the research questions
- **Problem Relevance:** Fire risks and mitigation are important topics, which affects human lives. Increasing the knowledge base with regards to fire risks and how weather influences the risks, is of high relevance to the application domain.
- **Design evaluation:** Following best practices for developing the software system that is the artefact, combined with experiments, will demonstrate the utility of the design.
- **Research Contributions:** The answers to the research questions are useful to the DYNAMIC research project.
- **Research Rigour:** Using the design artefact, experiments can be conducted on a large data set for evaluation of the artefact and its output.

- **Design as a Search Process:** The design artefact will consist of multiple software services that can be iteratively developed and improved.
- **Communication of Research:** This thesis serves as the main communication of the research conducted.

1.4 Thesis structure

This thesis is a collaborative effort of two authors, as a result of merging two problem statements, that partially overlapped, as evident through the research questions. Originally, each author focused on different part of the system, but after the merging, the entire system, that will be implemented in this thesis, are considered. The rest of this introduction outlines the chapters of the thesis, and if relevant, specify which author was the lead author of that chapter.

Chapter 2 surveys existing research on the topic of fire detection, risk, and management. Data sources for fetching weather measurements and forecasts, which are related to predicting fire risks, are presented. Finally, key concepts useful for understanding the other parts of the thesis are detailed. Section 2.2 is written by Halderaker.

Chapter 3 gives an overview of the software architecture and system that has been developed to answer the research questions.

Chapter 4 outlines the Fire Risk Model (FRM), the mathematical formulas and the corresponding implementation. Both Fire Risk Model (FRM) and Fire Risk Services (FRS) originally was the parts of the system that overlapped between the theses.

Chapter 5 present the Fire Risk Services (FRS), which use the FRM to calculate a fire risk.

Chapter 6 concerns the Data Harvesting Service (DHS), which is implemented to provide weather data to the FRM. It uses the open weather APIs detailed in Chapter 2. The software and chapter have been written by Halderaker.

Chapter 7 is concerned with providing the business logic to provide fire risks provided by FRS to the front-end application, as well as notifications. The software and chapter are written by Evjenth.

Chapter 8 provide the interface to fire risks provided by the Middleware to the end-users, through front-end applications. The software and chapter is written by Evjenth.

Chapter 9 describes a series of experiments that uses the developed systems to investigate their usefulness and to provide data to answer the research questions.

Chapter 10 concludes and summarize the work carried out in this thesis. Some reflections are given to related and future work and how one may utilize the solutions developed in this thesis.

In Appendix B URLs to the implementations can be found, as well as data collected in the experiments.

Chapter 2

Background and Related Work

Fire risk is a international research topic, and multiple models have been developed for predicting fire risks. In this chapter, we discuss the various models and approaches in the literature, and lastly potential sources of weather data for fire risk prediction.

2.1 Fire Risk Prediction

Fire risk research is such a broad research field, and a large variety of research and research methods exists. Different researchers and projects tackle the problem in varying ways.

Cortez and Morais [4] used data mining techniques, such as Support Vector Machines and Random Forests, with real-time data gathered from local sensors to predict small fires in Portugal. The features of their data consisted of four measurements from sensors (rain, wind, temperature, and humidity), combined with components of the Canadian Fire Weather Index (FWI) collected from previous fires. Their model was capable of predicting the burned area of smaller fires as they occurred.

San-Miguel-Ayanz et al. [41] at the European Commission developed The European Forest Information System (EFFIS). This system consists of multiple subsystems handling different aspects of forest fire management, from forest fire danger forecast

to active fire detection and post-fire analysis. Their forest fire danger forecast uses weather data to compute a common European fire danger index for over 30 countries in Europe. The weather data is collected from two meteorological forecast models, the French Météo-France and the Deutsche Wetter Dienst, combined with daily observations from weather stations across Europe. Similar to Cortez and Morais, EFFIS calculate the forest fire danger using the Canadian Fire Weather Index. Rusu et al.[40] employs unsupervised machine learning by doing outlier detection to data simulated by the Weather and Research Forecasting model (WRF-ARW) [7] to predict fire risk.

The Australia's 2019-2020 mega-fires burnt unprecedented large areas of vegetation, and Ward et al. [51] discusses the impact of such fires on wildlife and the need for recovery of threatened populations in both burnt and un-burnt areas. For wildfires in Australia, one model has been developed to determine likely speed and direction of the front of the fire [3]. It uses wind speed and dead fuel moisture content as inputs to calculate likely development of the fire. The wind speed is measured, and dead fuel moisture is either measured or estimated based on other factors.

Tsipis et al. [49] demonstrated a distributed Cloud/Fog IoT Solution for detecting wildfire while managing network throughput and energy consumption by reducing data collection. The distributed Wireless Sensor Network (WSN) of devices collected temperature and relative humidity data, and they could therefore utilize the Chandler Burning Index, which is a popular metric for assessing fire ignition risks.

2.1.1 Partial Risk Indicies

Another solution that do not rely on WSNs was proposed by Anezakis et al. [1]. It focuses on wild-fires in Greece and develops four partial risk indices that considerers weather, drought, topography and vegetation, amongst other factors, to determine an overall fire risk index.

2.1.2 Chandler Burning Index

The Chandler Burning Index (CBI) is a tool used to calculate a numerical index of fire danger. It only uses relative humidity and temperature as input, and is therefore simple to calculate. Since it does not take into account historic weather or fuel moisture

content of available material, it can lose some accuracy. A modified version of the CBI uses monthly predicted humidities and temperatures to obtain a value.

The modified equation [43] is

$$CBI = (((110 - 1.373RH) - 0.54 * (10.20 - T))(124 * 10^{-0.0142*RH}))/60$$

where

RH = forecast monthly mean afternoon relative humidity (percent)
T = forecast monthly mean afternoon temperature (degrees Celsius)

A number between 50-75 is categorized as a moderate fire risk, and lower values are lower risk, and higher values equates to higher risks. A value above 97.5 is categorized as an extreme fire risk.

The CBI can be a good tool for calculating forest fires. The fire risk model developed by the DYNAMIC research project, details a numeric approach for modelling relative indoor humidity and fuel moisture content, which can be a more useful tool for predicting indoor fire risks.

2.2 Weather Data Sources

The models discussed above use different data sources as their foundation. Particularly, meteorological data is a core data source when predicting fire risk, with temperature and relative humidity being important factors. Do et al. [8] interpolated high resolution meteorological data by combining data from local weather sensors with satellite data, which can be used by the fire prediction models. Bodrozic [2] employed sensor networks, where each network consisted of video cameras and meteorological stations to collect data to predict forest fires in Croatia. Common for many of these are IoT devices collecting weather data combined with quality controlled observations, which gets aggregated and analysed to achieve a deeper insight.

Choosing the right sources of weather data is important. The fire risk model developed in the DYNAMIC research project, and implemented in this thesis, uses outside temperature, wind speed, and outside relative humidity as input at specific time intervals. The quality of these sources need to be validated according to how they reflect the real weather. Some sources use industrial meteorological stations, while others use consumer grade sensors for their measurements. A second factor to quality is how widespread the measurements are, i.e., for a given location, how close is the nearest measurement of meteorological data. A data point measured by a weather station might not be representative for a place at a 10 km distance of the weather station due to local variations.

The different weather services update their measurements and forecasts on distinct intervals of time. To query a weather service more frequently than their update intervals is meaningless, since one would obtain identical data to the previous query. As such, a weather service's update interval needs to be taken into account.

2.2.1 The Norwegian Meteorological Institute

The Norwegian Meteorological Institute (MET) is a leading actor in Norway and is a government funded organisation. Together with the Norwegian Broadcasting Corporation (NRK), MET runs the website Yr.no, a public forecast service. Even though MET is a Norwegian actor, they provide coverage all around the globe. To provide this global coverage, MET use data from partner institutions, such as the European Centre for Medium-Range Weather Forecasts (ECMWF) and the European Organisation for the Exploitation of Meteorological Satellites (EUMETSAT). However, the forecasts are still updated more frequently in Scandinavia and Finland. Their weather forecast can be accessed either via their website Yr.no or the associated REST API [30] for broader use in applications and systems. Each forecast consist of air pressure at sea level, cloud area fraction, wind speed and direction, and more, but the most relevant for our system are air temperature, wind speed, and relative humidity. The REST API provide forecasts with hourly intervals for the first three days, and six hour intervals for the next seven days.

An example of a request to consume their API is a HTTP GET request to:

`https://api.met.no/weatherapi/locationforecast/2.0/complete?altitude=50&lat=60.3&lon=5.3`.

The forecast is provided in three formats: compact, complete, and classic. The compact format response contains only the most used weather parameters in a JSON body; complete is equivalent, but contains a forecast for every weather parameter MET offer; and lastly classic is a legacy XML response. Altitude is an optional parameter to specify meters above sea level, while `lat` and `long` are abbreviations for latitude and longitude, respectively, and are required parameters. As such, the example provided is a request for the weather forecast including all weather parameters at coordinate latitude 60.3 degrees and longitude 5.3 degrees and 50 meters above sea level. The JSON object returned from a compact forecast can be seen in Listing 2.1.

```json
{
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [
            5.3,
            60.3,
            42
        ]
    },
    "properties": {
        "meta": {
            ...
        },
        "timeseries": [
            {
                "time": "2021-04-22T10:00:00Z",
                "data": {
                    "instant": {
                        "details": {
                            "air_pressure_at_sea_level": 1021.5,
                            "air_temperature": 6.9,
                            "cloud_area_fraction": 27.6,
                            "relative_humidity": 58.3,
                            "wind_from_direction": 338.0,
                            "wind_speed": 6.7
                        }
                    },
                    ...
                }
            },
            {
                "time": "2021-04-22T11:00:00Z",
                "data": {
                    "instant": {
                        "details": {
                            "air_pressure_at_sea_level": 1022.0,
                            "air_temperature": 7.2,
                            "cloud_area_fraction": 24.9,
                            "relative_humidity": 56.7,
                            "wind_from_direction": 334.4,
                            "wind_speed": 7.1
                        }
                    },
                    ...
                }
            },
            ...
        ]
    }
}
```


Listing 2.1: Example of a JSON response body from the MET forecast REST API

Another service The Norwegian Meteorological Institute provide is the Frost API [30]. Through this API, MET provide access to their archive of historical weather and climate data, along with metadata for each meteorological station. Its base path is `https://frost.met.no/` and requires user credentials to use. These credentials can be requested by accessing

```
https://frost.met.no/auth/requestCredentials.html
```

and entering an email. The response contains a client ID and client secret. The secret is only needed when requesting data not open to the public [25], while the client ID needs to be provided as username in *Authorization* header of all HTTP requests to the Frost API. If the client ID is not provided in the *Authorization* header, a HTTP 401 Unauthorized response is returned.

To request weather observations for a location, one can request IDs of nearby stations, and then query the API with the station ID, the desired types of observations, and the desired time interval. Nearby MET stations can be find using the endpoint:

```
/sources/v0.jsonld?types=SensorSystem&geometry=nearest(POINT(5.7331 58.97)
)&nearestmaxcount=10&elements=air_temperature,relative_humidity,
wind_speed
```

The `types` parameter specifies the type of station, with `SensorSystem` being station with measuring sensors. Other types are `InterpolatedDataset` and `RegionDataset`. `geometry=nearest(POINT(5.7331 58.97))` defines the coordinate to find nearby stations, with the first number being longitude and the second number being latitude. The parameter `nearestmaxcount=10` is how many stations the response should contain. Lastly `elements=air_temperature,relative_humidity,wind_speed` requires each station to have data of the specified elements. The different types of elements include air temperature, wind speed, grass temperature, air pressure at sea level, relative humidity, and more. The nearby stations returned can be seen in Listing 2.2

```
{
    "@context": "https://frost.met.no/schema",
    "@type": "SourceResponse",
    "apiVersion": "v0",
    "license": "https://creativecommons.org/licenses/by/3.0/no/",
    "createdAt": "2021-04-22T10:50:59Z",
    "queryTime": 3.498,
    "currentItemCount": 10,
    "itemsPerPage": 10,
    "offset": 0,
    "totalItemCount": 10,
    "currentLink": "...",
    "data": [
        {
            "@type": "SensorSystem",
            "id": "SN44640",
            "name": "STAVANGER - VÅLAND",
            "shortName": "Våland",
            "country": "Norge",
            "countryCode": "NO",
            "wmoId": 1416,
            "geometry": {
                "@type": "Point",
                "coordinates": [
                    5.7278,
                    58.9563
                ],
                "nearest": false
            },
            "distance": 1.55338065213,
            ...
        },
        {
            "@type": "SensorSystem",
            "id": "SN44560",
            "name": "SOLA",
            "shortName": "Sola",
            "country": "Norge",
            "countryCode": "NO",
            "wmoId": 1415,
            "geometry": {
                "@type": "Point",
                "coordinates": [
                    5.637,
                    58.8843
                ],
                "nearest": false
            },
            "distance": 11.01035180888,
            ...
        },
        ...
    ]
}
```

Listing 2.2: Example of a JSON response from the Frost nearby stations endpoint

When the desired stations are found, the Frost API has an endpoint to the find meta data about each station. This endpoint returns time series available at the station, i.e. the periods the station has collected measurements:

```
/observations/availableTimeSeries/v0.jsonld?elements=air_temperature,
relative_humidity,wind_speed&timeresolutions=PT1H&sources=SN50540&
referencetime=2021-01-01T00:00/2021-01-31T00:00
```

The parameter `elements` is mentioned previously, `timeresolutions` specifies the resolution of the time series. PT1H is one hour between each data point and is an ISO-8601 time value. The `sources` parameter is the specific station ID, while `referencetime` specifies the time interval to check. The two timestamps are separated by a `/`, where the first timestamp is the start time and the second is the end time of the series. An example response can be seen in Listing 2.3. The endpoint can be used to check if a specific station still collects measurements, or it merely previously did, and it does not return actual measurements. Each time series contain a field *elementId* specifying the type of observation, another called *validFrom* specifying that the station started collected measurement at the timestamp, and an optional field *validTo*. The field *validTo* is the timestamp at the end of a period of collecting said type of measurements. When a time series does not contain *validTo*, the station currently collects this type of measurement, as the time series has no set end.

```
{
  "@context": "https://frost.met.no/schema",
  "@type": "ObservationTimeSeriesResponse",
  "apiVersion": "v0",
  "license": "https://creativecommons.org/licenses/by/3.0/no/",
  "createdAt": "2021-04-22T11:07:59Z",
  "queryTime": 0.204,
  "currentItemCount": 3,
  "itemsPerPage": 3,
  "offset": 0,
  "totalItemCount": 3,
  "currentLink": "...",
  "data": [
    {
      "sourceId": "SN50540:0",
      "level": {
          "levelType": "height_above_ground",
          "unit": "m",
          "value": 2
      },
      "validFrom": "1997-03-13T00:00:00.000Z",
      "validTo": "2008-05-07T00:00:00.000Z",
      "timeOffset": "PT0H",
      "timeResolution": "PT1H",
      "timeSeriesId": 0,
      "elementId": "relative_humidity",
      "unit": "percent",
      ...
  },
  {
      "sourceId": "SN50540:0",
      "level": {
          "levelType": "height_above_ground",
          "unit": "m",
          "value": 2
      },
      "validFrom": "2008-05-07T00:00:00.000Z",
      "validTo": "2014-07-08T06:45:30.000Z",
      "timeOffset": "PT0H",
      "timeResolution": "PT1H",
      "timeSeriesId": 0,
      "elementId": "relative_humidity",
      "unit": "percent",
      ...
  },
  {
    "sourceId": "SN50540:0",
    "level": {
        "levelType": "height_above_ground",
        "unit": "m",
        "value": 2
    },
    "validFrom": "2005-09-22T00:00:00.000Z",
    "timeOffset": "PT0H",
    "timeResolution": "PT1H",
    "timeSeriesId": 0,
    "elementId": "air_temperature",
    "unit": "degC",
    ...
},
...
  ]
}
```


Listing 2.3: Example of a JSON response from the Frost available timeseries endpoint

To request actual observations the `/observations/` endpoint can be used:

```
/observations/v0.jsonld?sources=SN50500&referencetime=2020-11-
16T11:00:00/2020-11-26T11:20:03.593696633&elements=air_temperature,
relative_humidity,wind_speed&timeresolutions=PT1H
```

These parameters act similar to the `/observations/availableTimeSeries/v0.jsonld` endpoint with `sources` being the station id, `referencetime` the time periode, `elements` specifies the types of weather data, and `timeresolutions` is the desired resolution of the weather data. An example response can be seen in Listing 2.4, where *elementId* is the type of measurement, and *value* contains the measured value, e.g. 9.1 degrees air temperature, or 80% relative humidity. The observations are hourly in most cases, but can be every six hours in cases of missing data.

The Norwegian Meteorological Institute provide multiple other endpoints in the Frost REST API. Some endpoints include complex calculations, such as maximum and minimum values of an observation over a timespan, mean values and summations.

```json
{
    "@context": "https://frost.met.no/schema",
    "@type": "ObservationResponse",
    "apiVersion": "v0",
    "license": "https://creativecommons.org/licenses/by/3.0/no/",
    "createdAt": "2021-04-22T11:14:19Z",
    "queryTime": 0.902,
    "currentItemCount": 241,
    "itemsPerPage": 241,
    "offset": 0,
    "totalItemCount": 241,
    "currentLink": "...",
    "data": [
        {
            "sourceId": "SN50500:0",
            "referenceTime": "2020-11-16T11:00:00.000Z",
            "observations": [
                {
                    "elementId": "air_temperature",
                    "value": 9.1,
                    "unit": "degC",
                    "level": {
                        "levelType": "height_above_ground",
                        "unit": "m",
                        "value": 2
                    },
                    ...
                },
                {
                    "elementId": "relative_humidity",
                    "value": 80,
                    "unit": "percent",
                    "level": {
                        "levelType": "height_above_ground",
                        "unit": "m",
                        "value": 2
                    },
                    ...
                },
                {
                    "elementId": "wind_speed",
                    "value": 3.4,
                    "unit": "m/s",
                    "level": {
                        "levelType": "height_above_ground",
                        "unit": "m",
                        "value": 10
                    },
                    ...
                }
            ]
        },
        ...
    ]
}
```


Listing 2.4: Example of a JSON response from the Frost observations endpoint

2.2.2 Netatmo

Another actor is Netatmo, which provide Smart Home Weather Stations [26] for the general public to set up at home for an affordable price. Each station consist of two devices, an indoor module and an outdoor module, where the inside module acts as a control hub connected to Wi-Fi. The indoor module captures the indoor climate in terms of temperature, relative humidity, CO_2 in ppm, and noise level in decibel. The outdoor module measure temperature, relative humidity, and barometric pressure. One can connect additional Netatmo modules, such as smart anemometer and smart rain gauge, to enable measurements of wind speed and rain. Netatmo stations record measurements every five minutes and are stored on the station [27]. Owners of a station can request data from the indoor module via the REST API with an access token, and permit applications to access measurements from their station. The latest measurement from the outdoor module, however, is publicly available online via a weather map [28] and through the weather REST API [29]. The base path for the API is `https://api.netatmo.com/`.

To gain access to Netatmo's REST API, one needs to create an account at their website and then register an application by filling in a form. The form requires an application name, a description, name, and email to the application's data protection officer. Having done this, the application receives a client identifier and a secret.

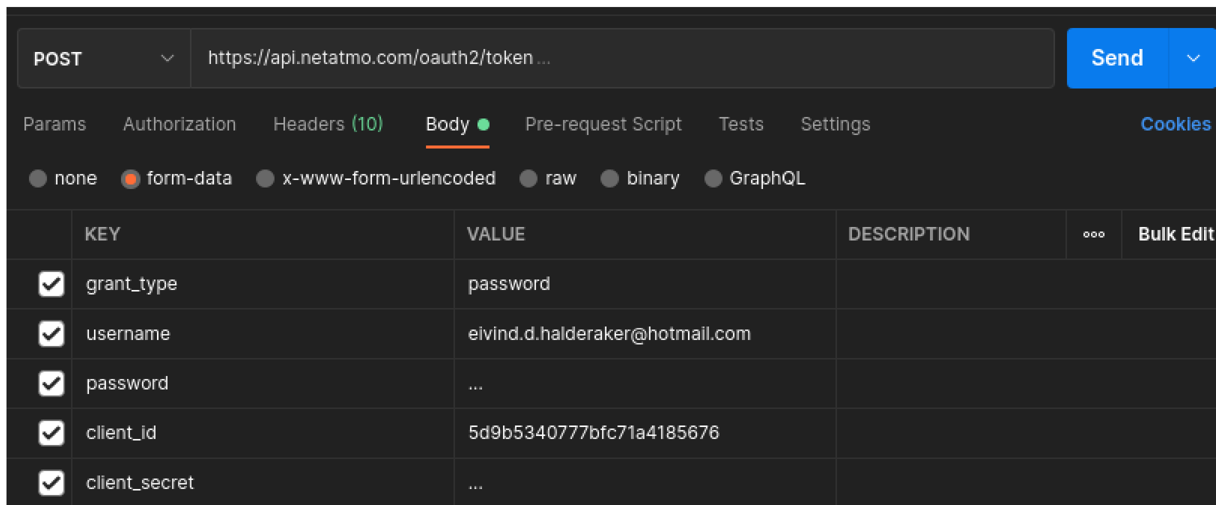

Figure 2.1: Requesting a Netatmo token in Postman

These can be used to authenticate and generate an Oauth [31] access token for the REST API by sending a HTTP POST request to the endpoint `/oauth2/token` with a body consisting of form data including client secret, client identifier, Netatmo username, Netatmo password, and a request to be granted a Oauth password that can be used as token. This can be seen in Figure 2.1.

If the request is correct, the response shown in Listing 2.5 is returned as a 200 OK HTTP response, and if the request is incorrect a 400 Bad Request is returned.

```
1 {
2   "access_token": "5d9b4f2643c228000b7bdd99|883374864ab06009ed9b2bc303f9566f",
3   "refresh_token": "5d9b4f2643c228000b7bdd99|ca778d606336a96c5dc18de5c666a9b7",
4   "scope": ["read_station"],
5   "expires_in": 10800,
6   "expire_in": 10800
7 }
```

Listing 2.5: Response from requesting a Netatmo access token.

Access tokens are only available for a periode, and before the access token expires, a new HTTP POST request can be sent to `/oauth2/token` to renew it. The body of the request must contain client identifier, client secret, refresh token, and a field to specify a request to refresh the access token, as seen in Figure 2.2. The response for renewing a token is identical to Listing 2.5

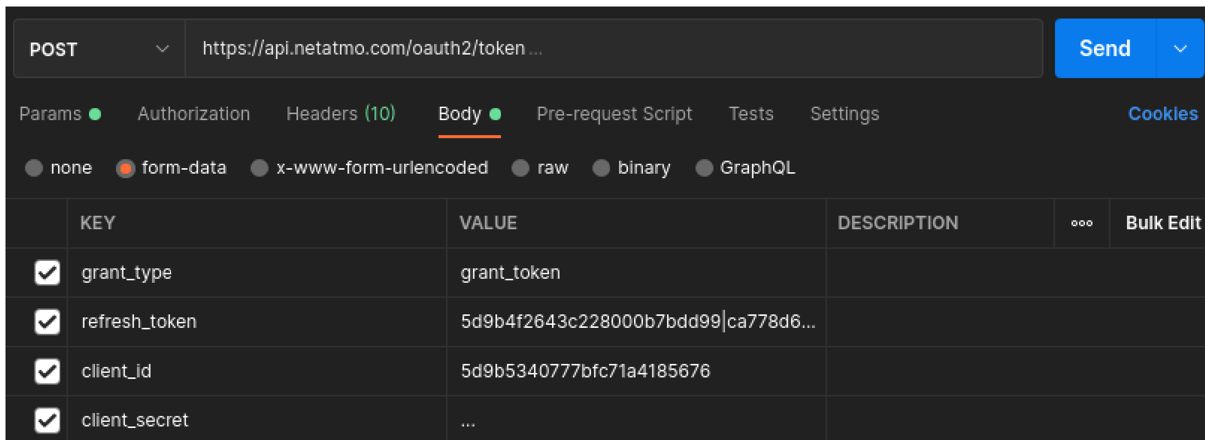

Figure 2.2: Refreshing a Netatmo token in Postman.

When an access token has been obtained, one can request public measurements from the outdoor modules using the url:

```
/api/getpublicdata?lat_ne=60.2&lon_ne=5.3&lat_sw=60.1&lon_sw=5.2&filter=
true
```

When collecting measurements from Netatmo, one has to specify two coordinates, one in north eastern corner of a polygon and the other in the south western corner. The parameter `lat_ne` specifies the latitude of the north eastern corner, while `lon_ne` specifies the longitude of the north eastern corner, and similarly for the south western corner. These are then used to find all stations inside this polygon. The JSON containing the measurements are shown in Listing 2.6. The filter parameter deicides whether or not to filter out abnormal measurements. The filter Netatmo use is The Norwegian Meteorological Institute's TITAN (auTomatIc daTa quAlity coNtrol) [48][5].

2.2.3 StormGeo

A third actor is StormGeo, a privately held weather service provider. Their main focus is to provide weather services for the industry [47]. StormGeo offer a weather REST API [46] and through this API one can access short-term and long-term weather forecasts as well as climate information for a location. They also analyse different forecasting model, such as ECMWF's, at specific locations for customers [45], and many additional services.

2.2.4 The European Centre for Weather Forecasts

The European Centre for Medium-Range Weather Forecasts (ECMWF) is an independent intergovernmental organisation [10]. As the name suggests, ECMWF produce weather forecasts, air quality analysis, ocean wave predictions, and more. They have a number of datasets available, however, not all are open to the public ECMWF-datasets, and they focus on partner organisations.

```
{
    "body": [
        {
            "_id": "70:ee:50:52:ee:8c",
            "measures": {
                "02:00:00:53:06:4a": {
                    "res": {
                        "1619102755": [
                            10.2,
                            50
                        ]
                    },
                    "type": [
                        "temperature",
                        "humidity"
                    ]
                },
                ...
            },
            ...
        },
        {
            "_id": "70:ee:50:20:9d:96",
            "measures": {
                "02:00:00:33:f3:48": {
                    "res": {
                        "1619102748": [
                            7.5,
                            61
                        ]
                    },
                    "type": [
                        "temperature",
                        "humidity"
                    ]
                },
                ...
            },
            ...
        },
        ...
    ],
    ...
}
```

Listing 2.6: Measurements from Netatmo's public data endpoint

2.2.5 OpenWeather

A fifth actor is OpenWeather. OpenWeather is a private company and provide weather data via both their website `https://openweathermap.org/` and their REST API. These APIs contain various data such as current weather, historical weather, and daily forecasts. To use their API, one has to sign up to their site, and users have limits based on their subscription level which includes Free, Startup, Developer, Professional, and Enterprise [32].

2.2.6 Comparing Weather Sources

For this thesis, the Netatmo API, MET's Frost API and MET's forecast API were chosen as sources of meteorological data. These provide their data for free via publicly available REST APIs, compared to OpenWeather being free up to a limit model, StormGeo's focus on businesses, and ECMWF's focus on partner organisations. In addition, Netatmo and The Norwegian Meteorological Institute was used by Stokkenes [44]. Forecasts are provided by MET, while both Netatmo and MET's Frost API provide historical measurements. MET have high quality controlled stations covering all of Norway, while Netatmo's customer grade have a high density in urban areas, close to houses.

2.3 Software Architecture

The software systems presented in chapter 3 outlines a fairly complex architecture of independently cloud-hosted components. This section prefaces the chapter, by providing some fundamental explanations of the core features.

2.3.1 Microservices

Microservices [11] are a loose term for describing the development of applications and systems as multiple, independently deployable services. The services can be created with different frameworks and languages, and hosted on different platforms. They can also all be created by the same framework and hosted on the same platform. Each microservice serves a distinct purpose, and they often communicate through lightweight mechanisms, such as RESTful interfaces.

Microservices can offer many benefits: development and deployment can be achieved more independently than in a larger monolithic application. Load balancing and resource management can be optimized more easily. For a system that uses many microservices and where some of them get a significant amount of requests, many cloud hosting services can automatically deploy copied instances of the specific service to handle the traffic, without having to instantiate other services that are not experience high traffic.

2.3.2 RESTful services

Some of the systems developed in this thesis, are developed as RESTful web APIs, otherwise known as REST APIs. Each of them have a base-URL from where they are deployed, and exposes some endpoints which can be used to interact with the service. The services are stateless, meaning they do not keep track of prior interactions with the same client. The client must therefore provide all the required parameters for the service to complete the request. The underlying communications between the services are through HTTP, which uses standardized methods, such as GET, POST and OPTIONS, to complete the requests.

2.3.3 Spring Boot

Spring boot is an application framework that can be used to build RESTful web services in Java. It has a lot of built-in libraries and resources, and simplifies the setup and configuration of a web service. The Spring Initializr [42] can be used to setup a Maven or Gradle project with many of the useful Spring libraries included. By setting up the projects as Maven or Gradle projects, continuous integration and deployment (CI/CD) and hosting can be simplified, since all of the external dependencies are declared within the project.

2.3.4 Cloud Hosting

When developing a system that will be made available to end-users in a browser, the software needs to be deployed somewhere. This can be done on a dedicated server, but another popular option is to use a cloud-provider to host the software. There are many cloud-providers with Google Cloud, Microsoft Azure and AWS being some of the most popular providers.

Another popular cloud-provider is Heroku, which is what has been used for the systems detailed in chapter 3. Heroku offers hosting of applications to a base URL. This allows the different sub-systems to be developed as separate applications that communicates with each other through an RESTful protocol. Heroku supports different frameworks and languages, most interesting Maven/Gradle (Java) applications as well as Node (Javascript) applications. This has allowed us to build the entire technology stack in Heroku, from front-end and Middleware, to RESTful web APIs for the back-end.

Chapter 3

Design and System Requirements

This chapter presents the components of the fire risk prediction system. There are five components: The Fire Risk Model (FRM), the REST API service using the model as a library called Fire Risk Services (FRS), the Data Harvesting Service (DHS) to collect the weather data, the front-end web application, and finally the middleware enabling communication between the FRS and the front-end. All these services run as independent microservices. To the end user, however, the application appear as one, with only a couple of endpoints.

Figure 3.1 illustrates how the systems are organized and communicate with each other. On the left, the front-end systems are connected to the Middleware. The Middleware uses information from the database to return results to the users, as well as sending requests to the FRS system for calculating new fire risks. The FRS in turn, requests weather data from the DHS in order to complete the fire risk calculations.

3.1 System Requirements

A large system as this with several components has certain requirements set while developing. These are detailed below.

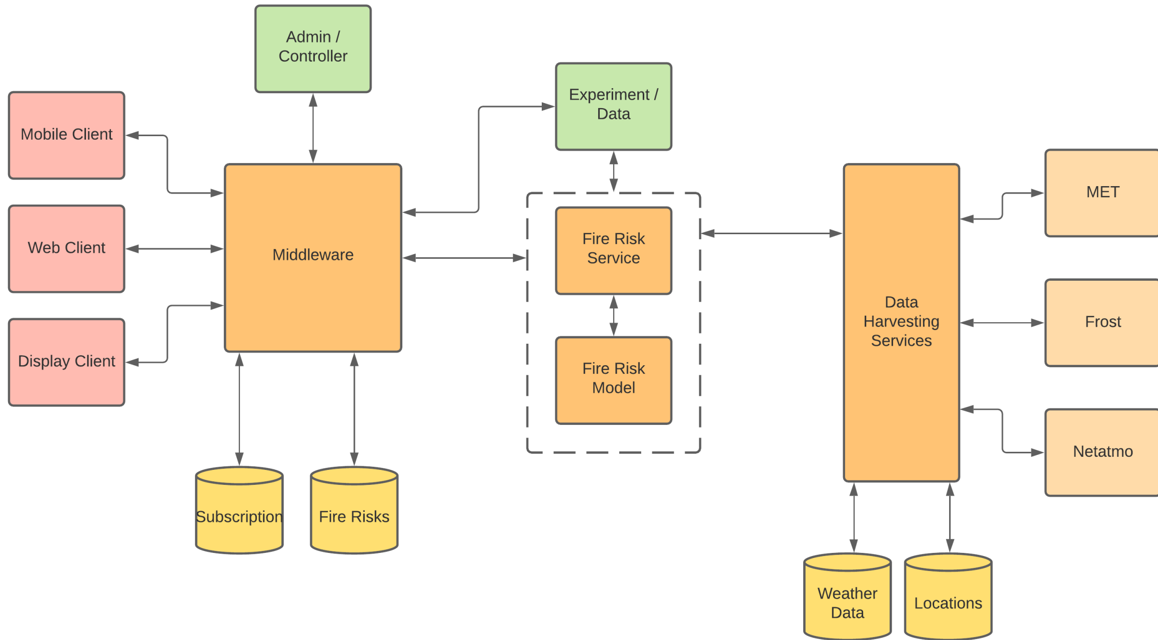

Figure 3.1: A birds eye view of the software architecture of the system.

The system should be modular with separate concerns for each service and the different parts should be loosely coupled. Defining interfaces, such as REST API end-points, for each service and the format of the intended data sent between them, helps when multiple developers work on the same system. The business logic within each service could be changed, updated, or rewritten without it affecting other services.

When a user request a fire risk prediction, it should not take long to compute the result. Instead of computing it on a received request, the system can save time by pre-computing fire risks.

The computed fire risks should be visualised to the user. A user needs to understand what is presented and what the system is capable of. A front-end, e.g. a web application is needed as an interface to the rest of the system. In addition, users will be interested in fire risk at specific locations.

The whole system is built around the fire risk model mentioned previously. It is used to estimate indoor humidity, time to flash-over, and a general fire risk. The model consists of several mathematical formulas, constants, and variables. These need to be

thoroughly tested with known input and results. Variables should be clearly defined in the code, and easily changed to deploy a new version of the model.

Figure 3.2 illustrates how data flows when a user requests a fire risk by selecting a location in the front-end application.

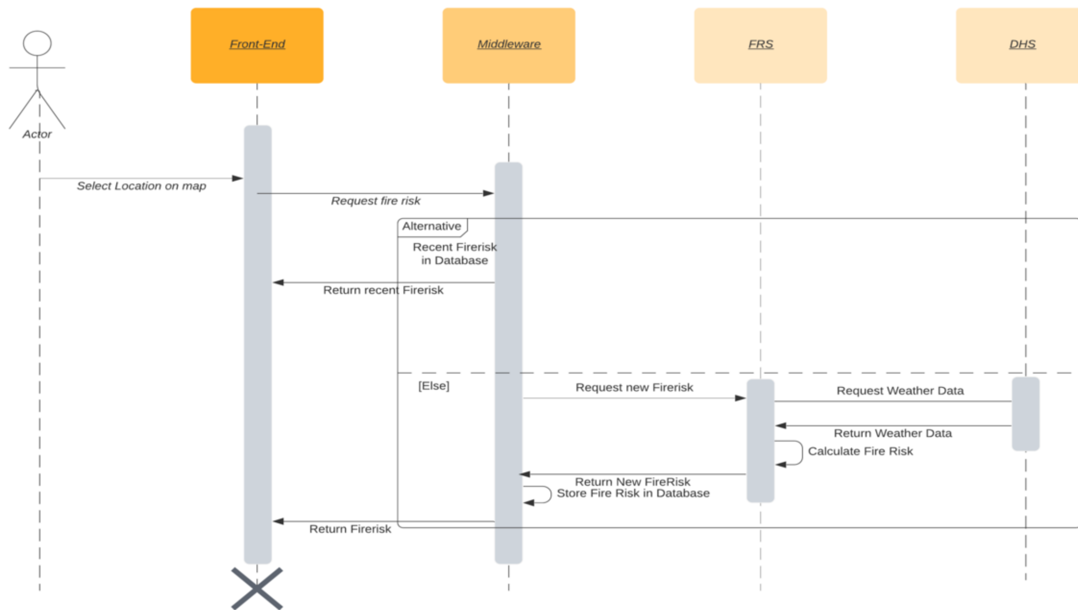

Figure 3.2: Data flow for fetching a fire risk for a location.

3.2 Deployment and DevOps

All the applications of the system, except the Fire Risk Model, are hosted on the cloud service Heroku. When deploying, Heroku builds the application and replaces the previously hosted version. The data in the database is not affected by deployment process and the deployment process does not invalidate any client sessions. If a user starts a series of requests that first targets the older version of the application, and then the newer one, there should not be any cause of errors. This is partly due to the fact that the handling of the client is stateless. The applications do not store or remember any previous interactions with the client, so the client must include any and all details required to complete a request.

3.2.1 Fire Risk Model

The Fire Risk Model is not per se hosted as an application running in the cloud, but rather as a library to be used by the other applications of the system, mainly Fire Risk Services. To be available to other applications, the model is hosted as a Maven [24] library using GitHub Packages [54]. Github has a feature called Actions that can be used to build, run, and test the source code before deploying it to a private Maven repository. After deployment, any Maven compatible application can configure access to the repository. However, a GitHub secret is needed.

3.2.2 Fire Risk Services and Data Harvesting Service

Both the Fire Risk Services and Data Harvesting Service applications are hosted on Heroku and import the Fire Risk Model library. As the Fire Risk Model is hosted in a private Maven repository, the Heroku Command Line Interface (CLI) is used to deploy the applications. This is due to Maven needing the GitHub secret to access the private Maven repository containing the FRM, which is easier to setup locally using the Heroku CLI compared to GitHub Actions.

3.2.3 Middleware

The Middleware application is deployed to Heroku as a Gradle project. The Heroku project is connected to the source code, hosted in GitHub, and any time there are changes on the default branch (merges and commits), a new version is built and deployed. Heroku integrates well with GitHub Actions, such that if a unit test fails during the testing stage, the deployment stage will not execute. Testing is described more in section 7.6.

3.2.4 Frontend

The frontend uses a similar pipeline for development, testing, and deployment. Dependencies are declared in a *package.json*-file and a tool, called Yarn, is used for installing the dependencies and building and running the project. Changes on the default branch are detected by the GitHub Action, and if it passes the tests, Heroku will build a production optimized bundle of the code and deploy it.

Chapter 4

Fire Risk Model (FRM)

The Fire Risk Model application is an implementation of Log's [23] modelling of indoor humidity in a single-family house. The indoor humidity can be used to estimate Time To Flash-over (TTF), which can serve as a fire risk indication. Computing the TTF based on forecasted weather data, further enables fire risk predictions. Then, combining the predicted TTF with forecasted wind speeds, may express a conflagration risk, which could be used to initiate proactive measures, and hence, reduce upcoming risk peaks. This conflagration risk can further be categorized into levels of risk. In general, the computed fire risk express a risk level for a single-family house, depending on the specific construction principle and internal materials. Since first published, the model has since gotten minor improvements by the DYNAMIC research project. In this chapter, we present the Fire Risk Model, the formulas behind the model, and their implementation.

To model indoor humidity, the model iteratively estimates the effect of three main factors; (1) Internal humidity production by humans or appliances inside the house, (2) the exchange of water vapour from hygroscopic materials inside the house, such as wood and gypsum, and (3) air exchange between indoor and outdoor air via different ventilation principles. Each iteration is separated by a time-step and the initial indoor humidity is set to 35%. The outdoor temperature, relative humidity, and optionally wind speed is given as parameters at every time-step from sources of weather data. Typically, the collected weather data is interpolated to fit the required timestep

in order to achieve numerical stability within the mode. The humidity produced by humans is set constant as it is meant to be an approximation of water vapour within an average living room.

Considering that the initial indoor humidity is a "best guess" at 35%, the model need some time to adjust. The modelled indoor humidity and associated concentration of water, called Fuel Moisture Content (FMC), in the outermost layer of the hygroscopic linings can be used to calculate an estimate of the Time To Flash-over (TTF) in minutes. As mentioned, TTF can be combined with forecasted wind speed to express a conflagration risk.

4.1 System Overview

The implementation of the FRM is developed as a Java library. Having the model as a plain Java library has several benefits. The Fire Risk Model can be published and easily imported into other projects. Being independent of frameworks makes it straight forward for new developers to contribute and require less framework specific experience. A library can be versioned to let developers of it update the library and publish new versions incrementally. Meanwhile, applications importing the library can specify versions and update on their own accord. The concrete implementation of the Fire Risk Model is available via [15].

4.2 Mathematical Formulas and Implementation

In this section, the formulas and their implementation in Java is described.

4.2.1 Modelling Relative Humidity

The first iteration of the model is based on estimated initial values and "best guesses", while later iterations model the new relative humidity based on the previous relative

humidity. The implementation the initial values can be seen in Listing 4.1. Line 2-6 specifies the constant indoor temperature, initial indoor relative humidity, and initial water concentration of indoor wooden panels, based on the specified indoor RH.

```
1     public static RelativeHumidityModelStep modelInitialHumidityStep(Observation
          ↪ firstObservation) {
2         // Constants
3         Temperature temperatureInside = ModelHumidity.temperatureInside;
4         double RH_in = ModelHumidity.InitialRH_in;
5         double C_w_sat_inConstantly =
              ↪ WaterExchangeInVentilation.C_w_sat(temperatureInside);
6         double c_w_in = RH_in * C_w_sat_inConstantly;
7
8         // Calculations to find c_wall
9         double[] waterConcentrationInWall = initialWaterConcentration(RH_in);
10
11        double concentrationInFirstLayer = waterConcentrationInWall[0];
12        double concentrationInSecondLayer = waterConcentrationInWall[1];
13
14        double C_surface = WaterExchangeInWall.C_surface(concentrationInFirstLayer,
              ↪ concentrationInSecondLayer);
15        double RH_wall = WaterExchangeInWall.RH_wall(C_surface);
16        double beta =
              ↪ WaterExchangeInVentilation.airDilutionRateBeta(firstObservation.getTemp());
17        double c_wall = WaterExchangeInWall.c_wall(RH_wall, RH_in,
              ↪ C_w_sat_inConstantly);
18        double c_ac = WaterExchangeInVentilation.c_ac(firstObservation.getTemp(),
              ↪ firstObservation.getHumidity().getHumidityPercentage());
19        LocalDateTime timestamp = firstObservation.getTimeObserved();
20
21        WallDto wallDto = new WallDto(waterConcentrationInWall, C_surface, RH_wall,
              ↪ c_wall);
22
23        VentilationDto ventilationDto = new VentilationDto(beta, c_ac);
24
25        StepDto stepDto = new StepDto(c_w_in, RH_in, timestamp);
26
27        return new RelativeHumidityModelStep(ventilationDto, wallDto, stepDto);
28    }
```

Listing 4.1: Implementation of setting the initial values.

The initial RH of the modelled house is set to 35%. The wall's indoor wooden panels, with thickness L, are modelled through N sublayers with respective water concentrations, called Fuel Moisture Content (FMC). In line 9 the initial water concentration is set in all the layers of panel in the wall by calling the method `initialWaterConcentration`, as shown in Listing 4.2. Initially, all layers have equal FMC. The number N is set as $N = \frac{L}{\Delta x}$, where Δx is the thickness of each sublayer.

```
1  private static double[] initialWaterConcentration(RelativeHumidityModelStep
       ↪ relativeHumidityModelStep) {
2      double initialWaterConcentration = WaterExchangeInWall.
3          convertFMCToConcentration(WaterExchangeInWall.
4          FMCForInitialConcentrationInFirstLayer(relativeHumidityModelStep.RH_in));
5
6      double[] waterConcentrationInWall = new
           ↪ double[WaterExchangeInWall.NPanelSubLayers];
7      Arrays.fill(waterConcentrationInWall, initialWaterConcentration);
8
9      return waterConcentrationInWall;
10 }
```

Listing 4.2: Method for setting initial water concetration.

The method `FMCForInitialConcentrationInFirstLayer` calculates the FMC specified in Equation 4.1.

$$FMC = 0.0017 + 0.2524 \cdot RH - 0.1986 \cdot RH^2 + 0.0279 \cdot RH^3 + 0.167 \cdot RH^4 \tag{4.1}$$

In line 17, the mean FMC, or water concentration at the surface, $C_{surface}$, of the first 2mm depth of the panels are calculated.

RH_{wall} is the relative humidity of the air at the wall surface and in line 18 it is calculated based on Equation 4.2.

$$RH_{wall} = 0.0698 - 1.258 \cdot FMC + 125.35 \cdot FMC^2 - 809.43 \cdot FMC^3 + 1583.8 \cdot FMC^4 \tag{4.2}$$

The value $beta$ is a measure of air exchange, given by

$$\beta = 1 - e^{(\frac{-ACH \cdot \Delta t}{3600s})} \tag{4.3}$$

```
1 public static double airDilutionRateBeta(Temperature temperature) {
2     double ACH = ACH(temperature);
3
4     return 1 - Math.exp((-ACH * WaterExchangeInWall.deltaT) / 3600);
5 }
```

Listing 4.3: Air exchange beta

The implementation can be seen in Listing 4.3 and is based on the number of air changes per hour (ACH),

$$ACH = \gamma \cdot \sqrt{\frac{\frac{1}{T_{out}} - \frac{1}{T_{in}}}{T_{out}}} \tag{4.4}$$

where T_{in} and T_{out} is the temperature inside and outside, respectively, at the first time step of modelling. The indoor temperature is set to a constant 22°C, while outdoor temperature is a variable from observations.

$\dot{c}_{wall}$ in line 22 is a measure of the change of water concentration in the bulk air given off by the wooden walls in the house,

$$\dot{c}_{wall} = \frac{\dot{m}_{wall}}{V_h} \tag{4.5}$$

Listing 4.4 shows the implemented method using the value $\dot{m}_{wall}$.

```
1 public static double c_wall(double previousRHwall, double previousRHin, double
      ↪ CwsatIn) {
2     double mWall = mWall(previousRHwall, previousRHin, CwsatIn);
3
4     return mWall / volume;
5 }
```

Listing 4.4: Calculation of $\dot{c}_{wall}$

$\dot{m}_{wall}$ is the transfer of moisture across the wooden wall surfaces, measured in kg/s, and V_h is the volume of the modelled living room, in m^3. V_h is set to a default volume of $150m^3$, while $\dot{m}_{wall}$ is calculated as,

$$\dot{m}_{wall} = \frac{A_{ex} \cdot D_{w,a} \cdot \Delta C \cdot \Delta t}{\delta} \tag{4.6}$$

```
1 public static double mWall(double previousRHwall, double previousRHin, double
      ↪ CwsatIn) {
2     double deltaC = deltaC(previousRHwall, previousRHin, CwsatIn);
3
4     return (A_ex * Dwa * deltaC * deltaT) / boundaryThicknessLayer;
5
6 }
```

Listing 4.5: Calculation of $\dot{m}_{wall}$

A_{ex} is the total area of the wood in contact with the bulk air, exchanging humidity. $D_{w,a}$ is the diffusion coefficient of water in air at 22°C and is a constant of $2.5 \cdot 10^{-10} m^2/s$. A thin layer of air adjacent to the wall surface is know as the boundary layer, δ, and it takes the value $0.01m$ [23]. This layer separates the bulk air and the wooden panels, while Δt is the time step between each iteration of the model.

ΔC is computed as

$$\Delta C = (RH_{wall} - RH_{in}) * C_{w,sat,in} \tag{4.7}$$

$\dot{c}_{ac}$ is the last value to calculate in the first iteration, on line 24. It is a measure of the change in water vapour concentration due to ventilation,

$$\dot{c}_{ac} = \frac{\dot{m}_{ac}}{V_h} \tag{4.8}$$

The implemented method is shown in Listing 4.6.

```
1 public static double c_ac(Temperature temperatureOutside, double RHout) {
2     double beta = airDilutionRateBeta(temperatureOutside);
3     double CwsatOut = C_w_sat(temperatureOutside);
4     double Cwout = Cw(RHout, CwsatOut);
5
6     return (beta * Cwout * (temperatureOutside.kelvin() /
        ↪ temperatureInside.kelvin()));
7
8 }
```

Listing 4.6: Calculation of $\dot{c}_{ac}$

Similarly to $\dot{m}_{wall}$, $\dot{m}_{ac}$ represents the change of air mass through different ventilation principles. Standard rates within the model is based on natural ventilation,

$$\dot{m}_{ac} = \beta \cdot V_h \cdot C_{w,out} \cdot \frac{T_{out}}{T_{in}} \tag{4.9}$$

$C_{w,out}$ can be expressed by RH_{out}, as a fraction between 0-1, and $C_{w,sat,out}$,

$$C_{w,out} = RH_{out} * C_{w,sat,out} \tag{4.10}$$

where saturated water concentration in kg/m^3 is,

$$C_{w,sat,out} = \frac{P_{w,sat,out} \cdot M_w}{R \cdot T_{out,k}} \tag{4.11}$$

M_w is the molecular weight of water vapour, R is the universal gas constant, and $T_{out,k}$ is the outside temperature expressed in Kelvin, and $P_{w,sat,out}$ is saturated vapour pressure,

$$P_{w,sat,out} = 610.78 \cdot e^{\frac{17.2694 \cdot T_{out,c}}{T_{out,c} \cdot 238.3}} \tag{4.12}$$

$T_{out,c}$ is outside temperature in Kelvin.

Having set the initial variables in the first iteration, the model needs to calculate dC, the change in water concentration inside the living room, for every subsequent iteration. The implementation is presented below in Listing 4.7, which rely on both the observation and previous model iteration to calculate the updated RH.

```
1     public RelativeHumidityModelStep modelRelativeHumidity(Observation observation,
          ↪ RelativeHumidityModelStep previousResultStep) {
2         Temperature temperatureInside = ModelHumidity.temperatureInside;
3
4         // ventilation
5         double CwsatInConstantly =
              ↪ WaterExchangeInVentilation.C_w_sat(temperatureInside);
6         VentilationDto ventilationDto = WaterExchangeInVentilation.step(observation);
7
8         // Wall
9         WallDto wallDto = WaterExchangeInWall.step(previousResultStep,
              ↪ CwsatInConstantly);
10
11        // Rest of the values
12        StepDto stepDto = step(observation, ventilationDto, previousResultStep,
              ↪ CwsatInConstantly);
13
14        return new RelativeHumidityModelStep(ventilationDto, wallDto, stepDto);
15    }
16
17    StepDto step(Observation observation, VentilationDto ventilationDto,
          ↪ RelativeHumidityModelStep previousResultStep, double CwsatInConstantly) {
18        double c_w_in = ModelHumidity.updateCwIn(ventilationDto.getBeta(),
              ↪ previousResultStep);
19        double RH_in = ModelHumidity.updateRHin(c_w_in, CwsatInConstantly);
20        LocalDateTime timestamp = observation.getTimeObserved();
21        return new StepDto(c_w_in, RH_in, timestamp);
22    }
```

Listing 4.7: Model current step.

Most of the logic is similar to `initialRelativeHumidity`, but the values from the previous step are used to update the current step.

In line 9, in the `step` method, the method `updateWaterConcentrationInWallLayers` is used as each layer is affected by their neighbouring layers.

```
private static double[] updateWaterConcentrationInWallLayers(RelativeHumidityModelStep previousResultStep, double CwsatInConstantly) {
    double[] updatedWaterConcentrations = new double[WaterExchangeInWall.NPanelSubLayers];

    double[] previousWaterConcentration = previousResultStep.waterConcentrationInWall;

    updatedWaterConcentrations[0] = WaterExchangeInWall.concentrationFirstLayer(previousWaterConcentration[0], previousResultStep.RH_in, previousResultStep.RH_wall, CwsatInConstantly, previousWaterConcentration[1]);

    for (int layer = 1; layer < updatedWaterConcentrations.length - 1; layer++) {
        updatedWaterConcentrations[layer] = WaterExchangeInWall.
            concentrationMiddleLayers(previousWaterConcentration[layer], previousWaterConcentration[layer - 1], previousWaterConcentration[layer + 1]);
    }

    updatedWaterConcentrations[WaterExchangeInWall.NPanelSubLayers - 1] = WaterExchangeInWall.
    concentrationInnermostLayer(
        previousWaterConcentration[WaterExchangeInWall.NPanelSubLayers - 1], previousWaterConcentration[WaterExchangeInWall.NPanelSubLayers - 2]
        );

    return updatedWaterConcentrations;
}
```

Listing 4.8: Updating the water concentration in the diffrenent layers

The concentration in the first layer, at index 0, can be found by,

$$C_{1_{t+\Delta t}} = C_{1_{(t)}} + \frac{\Delta t}{\Delta x} \cdot \left(\frac{D_{w,a}}{\delta} \cdot (RH_{in_{(t)}} - RH_{wall_{(t)}}) \cdot C_{sat,in} + \frac{D_{w,s}}{\Delta x} \cdot (C_{2_{(t)}} - C_{2_{(t)}}) \right) \quad (4.13)$$

where $t + \Delta t$ refers to the next iteration, while t refers to the previously calculated iteration.

Layer 2 to $N - 1$ is affected by the neighbouring panels and the previous water concentration of the layer. The water concentration in the layers can be determined from,

$$C_{n_{(t+\Delta t)}} = C_{n_{(t)}} + Fo \cdot (C_{n-1_{(t)}} - 2C_{n_{(t)}} + C_{n+1_{(t)}}) \quad (4.14)$$

Finally, the Nth layer is the last layer and only affected by its one neighbour.

$$C_{N_{(t+\Delta t)}} = C_{N_{(t)}} + Fo \cdot (C_{N-1_{(t)}} - C_{N_{(t)}}) \tag{4.15}$$

In line 18 of `modelRelativeHumidity`, the method `updateCwIn` is invoked to update the water concentration inside the house living room in the new iteration using Equation 4.16.

$$C_{w,in_{(t+\Delta t)}} = (1 - \beta) \cdot C_{w,in_{(t)}} + \dot{c}_{wall_{(t)}} + \dot{c}_{ac(t)} + \dot{c}_{supply_{(t)}} \tag{4.16}$$

The implementation can be seen in Listing 4.9.

```
1 public static double updateCwIn(double beta, double previous_Cw_in, double
    ↪ previous_c_wall, double previous_c_ac, double previous_c_supply) {
2     return (1 - beta) * previous_Cw_in + previous_c_wall + previous_c_ac +
        ↪ previous_c_supply;
3 }
```

Listing 4.9: Updating the water concentration inside the house

At last, line 19 sets the updated relative humidity indoor. This is done by converting it from a concentration to a percentage, as seen in Listing 4.10.

```
1 public static double updateRHin(double CwIn, double CwsatIn) {
2     return CwIn / CwsatIn;
3 }
```

Listing 4.10: Updating relative humidity inside the house

4.2.2 Time To Flash-over

Modelling the relative humidity indoor is only the first step towards a meaningful measure of fire risk, Time To Flash-over. The term flash-over is defined previously as "The transition from a localized fire to the general conflagration within the compartment when all fuel surfaces are burning." [9]

```java
public static TimeToFlashover ttf(double[] waterConcentrationInWall, double deltaX,
    ↪ LocalDateTime timestamp) {
    int n = numberOfLayersInTargetDepth(deltaX, metersOfWoodPanelToUseForTTF);

    double averageWaterConcentrationInFirstNLayers = 0;
    for (int layerIndex = 0; layerIndex < n; layerIndex++) {
        averageWaterConcentrationInFirstNLayers +=
            ↪ waterConcentrationInWall[layerIndex] / n;
    }

    // Calculate fuel moisture content
    double fmc =
        ↪ ModelHumidity.fmcInPercentage(averageWaterConcentrationInFirstNLayers);

    // Calculate ttf
    double ttf = ttfFromFMC(fmc);
    return new TimeToFlashover(ttf, timestamp);
}
```

Listing 4.11: Calculation of Time To Flash-over.

In the equation for Time To Flash-over, the Fuel Moisture Content (FMC) in the first 2 mm of the wooden panels is used. FMC is either given as a weight percentage or as a value in the unit interval. In the equation for Time To Flash-over, Equation 4.17, the former is used, weight percentage, $FMC_{wt\%}$. The method is presented in Listings 4.11, while the formula itself is shown in Listings 4.12. Note that the equation is valid in the interval 20% - 60% RH [21].

$$t_{FO} \approx 2.0 \cdot e^{0.16 \cdot FMC_{wt\%}} \tag{4.17}$$

As the model splits the wooden panels into discrete layers, the number of layers within the first 2 mm is calculated first. Secondly, the average water concentration of these layers are found, and converted to $FMC_{wt\%}$. Finally, the Time To Flash-over is calculated.

```java
private static double ttfFromFMC(double fmc) {
    return 2.0 * Math.exp(0.16 * fmc);
}
```

Listing 4.12: Calculation of Time To Flash-over based on Fuel Moisture Content.

4.2.3 Fire Risk Factor

As Time To Flash-over can be hard to grasp by itself, the DYNAMIC research project has described a fire risk factor, which combines TTF with wind to produce a general expression of conflagration risk. In urban environments, structures are constructed in close proximity to each other and, as such, wind poses a real risk in spreading fires to nearby buildings or over large distances.

The fire risk factor, R_f combines the area coverage dry wood factor, α, with wind energy influence, v, as seen in Equation 4.18, and the implementation is presented in Listing 4.13.

$$R_f = \alpha \cdot v \tag{4.18}$$

```
1 public static FireRiskFactor riskFactor(TimeToFlashover ttf, WindSpeed
      ↪ measuredWindSpeed, LocalDateTime timestamp) {
2     double alpha = alpha(ttf, timestamp);
3
4     double v = v(measuredWindSpeed);
5     return new FireRiskFactor(R_f(alpha, v));
6 }
```

Listing 4.13: Calculation of Fire Risk Factor.

The area coverage dry wood factor uses TTF at 50% as a reference level, compared to the relative humidity (RH).

$$\alpha \approx \left(\frac{t_{FO(50\%RH)}}{t_{FO(X\%RH)}}\right)^3 \tag{4.19}$$

The wind energy influence, v, is defined if greater than the critical wind velocity, u_c. If it is lower, the influence is 1 and does not add to the Fire Risk Factor.

$$v \approx \left(\frac{u}{u_c}\right)^3, u > u_c \tag{4.20}$$

In the implementation u_c is set to $10m/s$.

The resulting Fire Risk Factor can be categorized into levels of risk, as seen in Table 4.1. These categories lets non-technical people relate to the fire risk factor.

Value R_f	**Description**
$0 \leq R_f < 4$	Low risk
$4 \leq R_f < 10$	Medium risk
$10 \leq R_f < 30$	High risk
$30 \leq R_f < 60$	Very high risk
$60 \leq R_f$	Extreme risk

Table 4.1: Fire risk categories

4.2.4 Interpolation

The model requires weather observation for each of the steps in the computation, but since each step is much shorter (720 seconds) than the window of the weather observations (1 hour), the list of weather observations needs to be filled with interpolated data. To get the correct frequency of observations, of 720 seconds, there needs to be 4 interpolated observations between each of the observation provided to the model. Temperature, humidity, and optionally wind, are interpolated linearly. E.g., if the first observation is at 13:00 with an air temperature of 10°celsius and the second is at 14:00 with 12°celsius, The first will be at 13:12 with 10.4 degrees, second at 13:24 with 10.8, third at 13:36 with 11.2, and lastly fourth at 13:48 with an air temperature of 11.6 degrees. Each with 12 minutes between them and $(12 - 10)/N = 0.4$ degrees difference.

4.3 Deployment and DevOps

The Fire Risk Model is published as a Java library via GitHub Packages [54] and Maven [24]. In the *pom.xml* file that Maven uses to handle all dependencies and configurations for building the project, there is a version number. When code changes are committed to the main branch in GitHub, a script runs automatically to build the project and deploy it as a package with the new - incremented - version number.

Listing 4.14 shows the script used for this. The two top-nodes in the script, `on` and `jobs` describes when the action will run, and what it does when it runs. Here, `on` says that this action will run every time code are pushed to the main branch. This happens

when local commits are pushed to the repository, and when pull requests are merged into the branch. Line 13-35 declared a series of steps to run. 13-21 are steps for setting up the runtime environment with Java, and configuring the settings that Maven uses. Line 22-35 are the steps that deploys the package using Maven. First, a *.jar*-file are deployed locally on the runtime environment. On line 25, a secret is provided from the GitHub repository, such that it has read and write access to the packages directory. Then finally, on line 31-35 the local file is uploaded to GitHub Packages to a specific directory, here named 'myPackage'.

```
1  name: Java CI with Maven
2
3  on:
4    push:
5      branches:
6        - main
7  jobs:
8    build:
9
10     runs-on: ubuntu-latest
11
12     steps:
13       - uses: actions/checkout@v2
14       - uses: actions/setup-java@v1
15         with:
16           java-version: 14
17           settings-path: ".m2/"
18       - uses: actions/cache@v2
19         with:
20           path: ~/.m2/repository
21           key: ${{ runner.os }}-${{ hashFiles('**/pom.xml') }}
22       - name: Publish package
23         run: mvn $MAVEN_CLI_OPTS clean deploy
24         env:
25           GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
26           MAVEN_CLI_OPTS: "-s .m2/settings.xml --batch-mode"
27       - name: Copying target jar
28         run: |
29           mkdir myTarget
30           cp target/*.jar myTarget
31       - name: Uploading jar
32         uses: actions/upload-artifact@v2
33         with:
34           name: myPackage
35           path: myTarget
```

Listing 4.14: GitHub Action for deploying new package to GitHub Packages.

Chapter 5

Fire Risk Services (FRS)

The Fire Risk Model (FRM) is implemented as a library in java, and are not, by default, usable as a RESTful web service. The Fire Risk Services (FRS) wraps the implementation of the FRM and exposes its features through a selection of endpoints. Some additional endpoints are implemented to add new locations to the DHS. In this chapter, we present the Fire Risk Services and its endpoints.

5.1 System Overview

Following Figure 3.1, the FRS is a service that negotiates information between the Middleware and the Data Harvesting Service (DHS). Requests from the Middleware are fulfilled by requesting data from the DHS, and returning calculated results from the FRM based on that data. When the Middleware requests new locations to be tracked by the DHS, the requests are simply passed along without modifications.

The system is implemented as a RESTful web API using the Spring Boot framework. Dependencies, such as the FRM, are listed in a file *pom.xml* which allows Maven to build the project without any local dependencies. Figure 5.1 shows a class diagram for some of the functionality implemented in the project. The concrete implementation of the Fire Risk Services is available via [16].

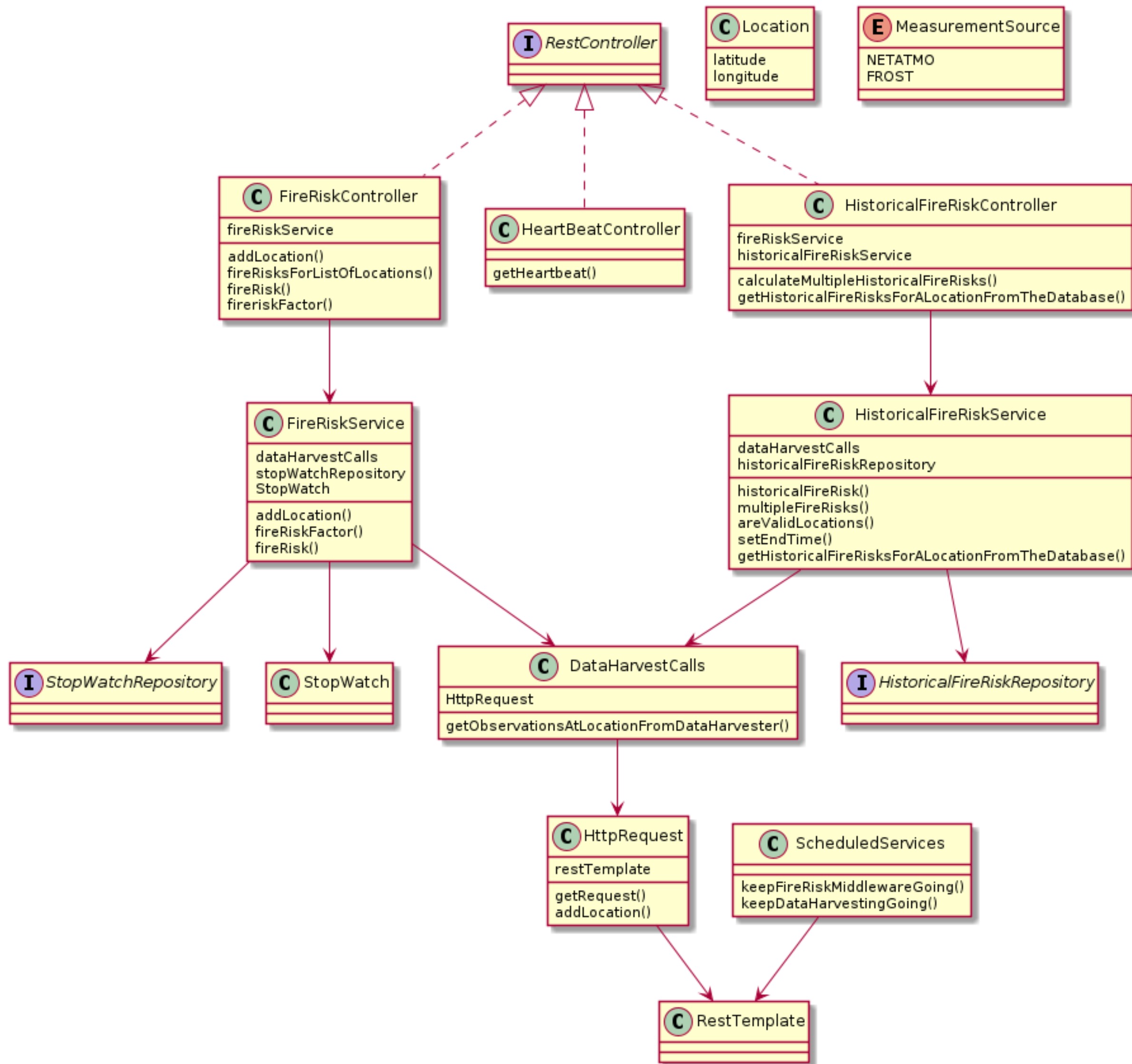

Figure 5.1: Class diagram for the FRS.

5.2 Endpoints

The list below list every endpoint in Fire Risk Services. The following subsections explain the endpoints in more details.

- **GET** `/heartbeat`
- **POST** `/location/add`
- **POST** `/firerisk`
- **POST** `/fireriskfactor`

5.2.1 Heartbeat

The heartbeat-endpoint is a request that simply returns a "200 OK" status with a random unique identifier (UUID). It is used by the Fire Risk Services to prevent the deployed applications, Middleware and DHS, from going into a sleep mode. This was a necessity for collecting data for the experiments detailed in chapter 9, as they required scheduled services, and Heroku applications automatically enters sleep mode after 30 minutes of inactivity.

5.2.2 Add locations

The `/location/add` endpoint accepts a JSON body with a location. It passes along the location to the corresponding endpoint of the DHS for adding locations. This endpoint is mainly used by the Middleware when users subscribe to a location, but can also be used to inject new locations that bypasses the Middleware.

The Middleware could request the endpoint of the DHS directly, but following the separation of concerns, and the overall architecture described in Figure 3.1, the FRS should transparently handle any requests from the Middleware.

5.2.3 Firerisk

The `/firerisk` endpoint accepts a JSON body with a location. It requests weather data from the DHS for the locations, and models a fire risks that is returned. If the DHS does not have data for that location, an exception is thrown.

In Listing 5.1, code for usage of Fire Risk Model (FRM) is shown. The observations of air temperature, relative humidity, and wind speed are requested from Data Harvesting Service, mapped into the FRM Observations, and interpolated before given as parameter to Fire Risk Model's method to compute a fire risk result.

```
1
2 List<ObservationResponse> observationResponse =
    ↪ dataHarvesterCalls.getObservationsAtLocationFromDataHarvester(location,
    ↪ measurementSource.toString(), previousDaysOfMeasurements, resolution);
3
4 List<Observation> observations =
    ↪ ObservationMapper.mapObservationResponseToObservations(observationResponse);
5
6 List<Observation> interpolatedObservations = Interpolation.generate(observations,
    ↪ (int) WaterExchangeInWall.deltaT);
7
8 stopWatch.start();
9
10 fireRiskResult = FireRiskModel.computeFireRiskResult(interpolatedObservations);
11
12 stopWatch.stop();
```

Listing 5.1: Usage of the Fire Risk Model in the Fire Risk Services

5.2.4 Firerisk factor

The `/fireriskfactor` endpoint does the same as the The `firerisk` endpoint, but uses the FRM to model a factor instead of the TTF value. It include wind in the calculations, and is useful combined with the TTF to model risk of fire spreading between structures.

5.3 Database

The Fire Risk Services does not need to be connected to any database for the services that it provides, but during development and the experiments, which are conducted in Chapter 9, the time used to compute each fire risk by the FRM was stored in a database, in addition to storing historical Time To Flash-over calculations. The configurations for connecting to a database are defined in the file *application.properties*. For simplicity, this system uses the same database as the Middleware and Data Harvesting Service, but is not required to do so.

5.4 Deployment and DevOps

The FRS is deployed as a stand-alone web API in a cloud-provider, named Heroku. Deployment is done through a CLI tool called Heroku CLI [18]. Heroku uses a wrapper for Maven, which is included in the project structure, to build and serve the web API. The source code is versioned and maintained in GitHub.

Chapter 6

Data Harvesting Service (DHS)

As the Fire Risk Model requires various weather data as input, the Data Harvesting Service is responsible for collecting it at the specified locations. It collects weather measurements to use from The Norwegian Meteorological Institute's Frost API and Netatmo's API. In this chapter we present the Data Harvesting Service and its endpoints used to interface the service.

6.1 System Overview

The application is, similar to Fire Risk Services, written in Java with the Spring Boot framework. Figure 6.1 lists most of the top-level packages in the project, following recommended best practices for structuring a Spring Boot application. Figure 6.2 gives an overview of the implementation. Classes with the `Controller` suffix are responsible for defining endpoints, classes with the `Service` suffix implement the main business logic, while the `Repository` suffix is used for classes interfacing the database, and the `Calls` suffix is used for classes responsible for requesting data via external APIs. The concrete implementation of the Data Harvesting Service is available via [12].

```
data-harvester
├── apicalls
├── config
├── controller
├── exception
├── model
├── repository
├── scheduledtasks
├── service
└── Application
```

Figure 6.1: File structure of Data Harvesting Service

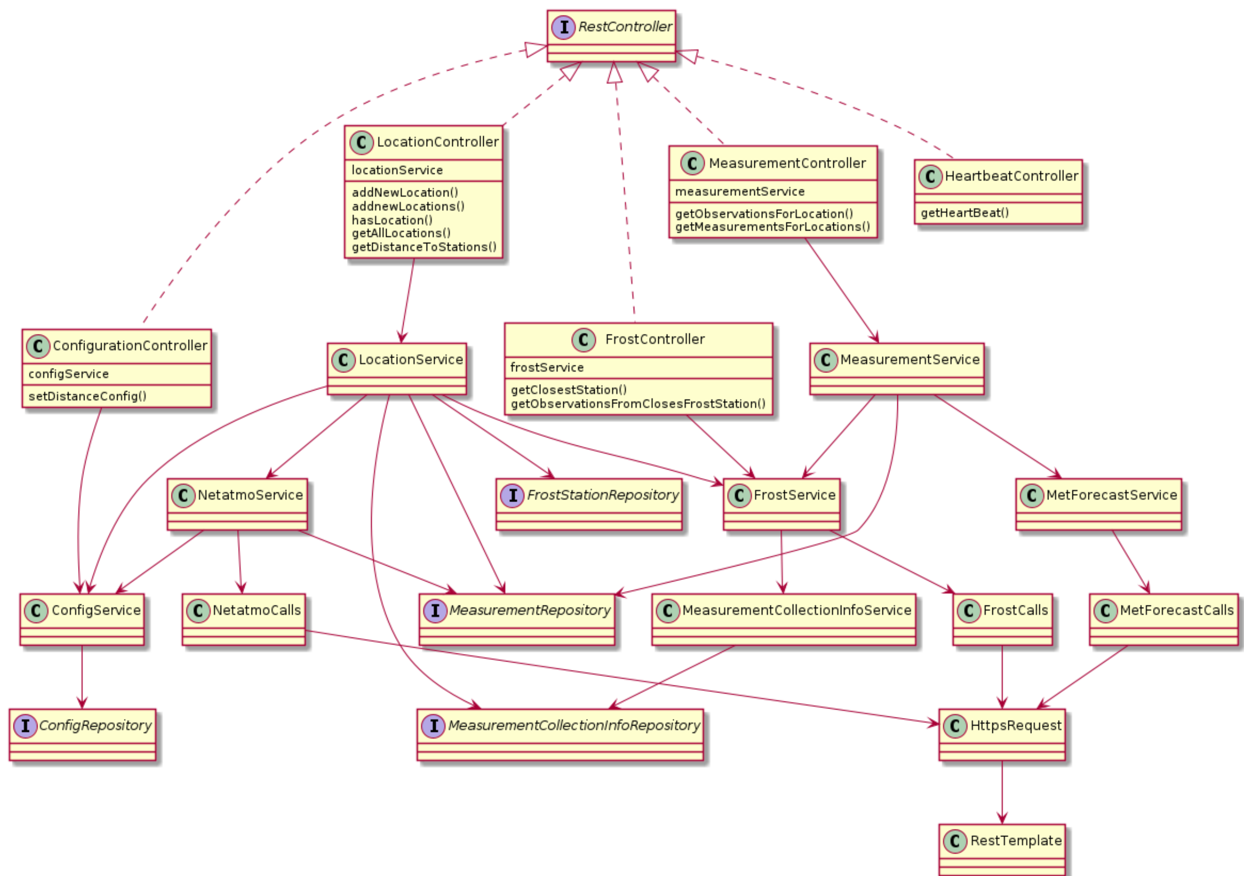

Figure 6.2: Class diagram for the DHS.

6.2 Endpoints

The application has four main REST API endpoints:

- **GET** `/measurements/({latitude},{longitude})?historicalMeasurementSource={string}&previousDaysOfMeasurements={integer}&historicalMeasurementResolution={integer}&forecastResolution={integer}`
- **GET** `/location/exists/({latitude},{longitude})`
- **POST** `/location`
- **GET** `/frost/observations/({latitude},{longitude})?startTimeString=timestamp&endTimeString=timestamp`

Below we provide more details on the services provided by the endpoints.

6.2.1 Measurements

The endpoint `/measurements/({latitude},{longitude})` is the one Fire Risk Services sends a HTTP GET request in order to get the weather data needed to calculate a fire risk. Based on sensitivity studies undertaken by the DYNAMIC research project, the Fire Risk Model needs approximately five days to achieve a sufficiently accurate relative humidity. As a result, to calculate an accurate indication of the fire risk in an area for the present time, modelling relative humidity the previous five days is preferable.

To predict fire risk for the future, the DHS uses weather forecasts from MET as a replacement for the measurements. Figure 6.3 shows how DHS combines weather measurements and forecasts. A database is used by the DHS to store the gathered weather measurements and forecast. Weather data from all three weather services are collected and stored as the service runs. The code can to get weather data from the database can be seen in Listing 6.1. Measurements from Netatmo and Frost are collected every hour by the DHS, while the MET forecasts are collected every sixth hour.

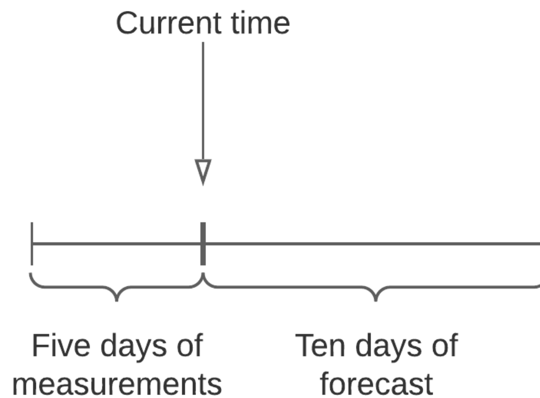

Figure 6.3: Usage of relevant weather data

```
1  public List<Measurement> getHistoricalMeasurementsAndForecast(Location location, long
    ↪ previousDaysOfMeasurements) throws NoFrostStationNearbyException,
    ↪ NoMeasurementsException, MeasurementCollectionException,
    ↪ MetForecastRequestException {
2
3      List<Measurement> historicMeasurements = measurementRepository.
4      findMeasurementByMeasurementId_LocationAndMeasurementSource
5          AndMeasurementId_TimestampBetween(location, MeasurementSource.FROST,
              ↪ LocalDateTime.now().minusDays(previousDaysOfMeasurements),
              ↪ LocalDateTime.now());
6
7      List<Measurement> forecast = measurementRepository.
8      findMeasurementByMeasurementId_LocationAndMeasurementSource
9          AndForecastCollectedAtBetween(location, MeasurementSource.MET_FORECAST,
              ↪ LocalDateTime.now().minusHours(6), LocalDateTime.now(),
              ↪ LocalDateTime.now());
10
11     return Stream
12             .concat(historicMeasurements.stream(), forecast.stream())
13             .collect(Collectors.toList());
14 }
```

Listing 6.1: Requesting measurements and forecast from the database.

6.2.2 Location exists

The endpoint `/location/exists/({latitude},{longitude})` responds to whether not the Data Harvesting Service application actively collects weather data for a specific location.

6.2.3 Add location

The endpoint `/location` is to add a new location for which weather data is to be retrieved. The location coordinates are contained in the request body. When adding a new location for the service to collect data for, one sends a HTTP POST request to the endpoint. The request consists of a JSON body with three fields: latitude, longitude, daysToCollectMeasurements, shown in Figure 6.2. Latitude and longitude represent the coordinates of the location and daysToCollectMeasurements describes how many days the Data Harvesting Service should store the measurements after being collected.

```
1 {
2   "latitude": 62.1453,
3   "longitude": 6.0748,
4   "daysToCollectMeasurements": 5
5 }
```

Listing 6.2: An example JSON body of a request to add a location.

As the Data Harvesting Service receives the request, it first checks the associated database for existing locations. If the requested location already is stored, then a HTTP 400 Bad Request response is returned, as there is no use in collecting the same measurements twice. If not, the closest MET station is found from the Frost API. The API lets one request nearby stations with observations of temperature, relative humidity, and wind speed. However, it does not differentiate between stations actively collecting measurements and those that previously did. To avoid this, the Data Harvesting Service request the ten closest stations. Based on this, stations with no time series of these three weather observation types of temperature, relative humidity, and wind speed, in the last couple of days are filtered out by using the `/observations/availableTimeSeries` endpoint.

Finally, the station with lowest distance to the requested location is chosen. Measurements collected at this station from the past days are requested and stored. How many days of past measurements is dependent on the field daysToCollectMeasurements in the request.This design makes it easy to test different values. After the measurements are stored, the location is stored with information regarding the MET station and how long to keep measurements. If everything was successful, an HTTP 200 OK response is returned.

6.2.4 Historical Measurements

The endpoint `/frost/observations/({latitude},{longitude})` is used in the experiments to request historical measurements from MET's Frost API for multiple days. The code can be seen in Listing 6.3. Line 4 requests meta data about the closest Frost station and Line 6 and 7 requests the measurements for the station. Line 9-11 throws an exception in case no measurements are returned. Finally, the measurements are mapped to Fire Risk Model's format.

```
1  public List<Observation> getObservationsFromClosestFrostStation(Location
       ↪ location,LocalDateTime startTime,LocalDateTime endTime)
2      throws NoFrostStationNearbyException,MeasurementCollectionException,
           ↪ NoMeasurementsException {
3
4      FrostStation frostStation = getClosestStation(location,startTime,endTime);
5
6      List<Measurement>previousFrostMeasurements =
7          getMeasurementsAtLocation(frostStation,location,startTime,endTime);
8
9      if(previousFrostMeasurements.isEmpty()){
10         throw new NoMeasurementsException();
11     }
12
13     return FireRiskServiceMapper.mapToObservationResponse(previousFrostMeasurements);
14 }
```

Listing 6.3: Collecting historical measurements from Frost.

6.3 Scheduled Services

Weather measurements and forecasts are periodically collected from the stored locations. To distribute the workload, data is collected at specific times for each source by setting up cron jobs, as seen in Listing 6.4. Every sixth hour at five minutes past the hour, 06:05, 12:05, 18:05, and 24:05, the forecast from MET is collected for each location and stored in the connected database. Similarly, every hour at 17 minutes past, a measurement from the closest MET station is collected. Lastly, every hour at 42 minutes past, measurements from Netatmo stations near each location is collected and averaged to produce one average measurement for each location.

To define the corners required by Netatmo's endpoint `/getpublicdata`, a 10km by 10km polygon is calculated with the desired location in the center. A filter option is used when requesting stations to filter out stations with abnormal measurements. The forecast from MET is requested less frequent, as its purpose is to replace the previous forecast with a slightly more accurate one. We assume that forecasts rarely change significally from hour to hour. When collecting from the Frost API, the closest station for each location is requested.

```
1  @Scheduled(cron = "0 41 * * * ?")
2  public void getMeasurements() {
3      System.out.println("New measurements being collected");
4
5      List<Location> locations = locationService.getAllLocations();
6
7      for (Location location : locations) {
8          try {
9              Measurement averageMeasurementsFromArea =
                   ↪ netatmoService.getAverageMeasurementsFromArea(location);
10             measurementRepository.save(averageMeasurementsFromArea);
11             System.out.println("Collected measurement: " +
                   ↪ averageMeasurementsFromArea);
12         } catch (NoNetamoStationNearbyException e) {
13             System.out.println("Found no measurement at location " + location);
14         }
15     }
16 }
```

Listing 6.4: A scheduled service as a cron job collecting Netatmo measurement.

6.4 Database

The Data Harvesting Service uses a database to store measurements, information about each location, and the configured minimum distance between each location. For the experiments, a MSSQL database hosted on HVL was used. The Measurement class can be seen Listing 6.5 and is used as an internal representation of a requested observation or forecast, and the id class is presented in Listing 6.6. As measurements can only be created from Netatmo, The Norwegian Meteorological Institute's Frost API, or The Norwegian Meteorological Institute's forecast, they each have a method to create a Measurement object without directly using the constructor, which is private. The `@Entity` notation is used in Spring Boot to indicate classes to be mapped into tables in the database.

```java
@Entity
public class Measurement {

    @EmbeddedId
    private MeasurementId measurementId;
    private double temperature;
    private double humidity;
    private Double windSpeed;
    private MeasurementSource measurementSource;
    private LocalDateTime forecastCollectedAt;

    public static Measurement asNetatmoMeasurement(LocalDateTime date, Location location, double temperature,
            double humidity, Double windSpeed) {
        return new Measurement(new MeasurementId(date, location), temperature, humidity, windSpeed,
                MeasurementSource.NETATMO, null);
    }

    public static Measurement asFrostMeasurement(LocalDateTime date, Location location, Optional<Double> temperature,
            Optional<Double> humidity, Optional<Double> maybeWindSpeed) throws MeasurementCollectionException {
        Double windSpeed = null;
        if (maybeWindSpeed.isPresent()) {
            windSpeed = maybeWindSpeed.get();
        }
        return new Measurement(new MeasurementId(date, location),
                temperature.orElseThrow(MeasurementCollectionException::new),
                humidity.orElseThrow(MeasurementCollectionException::new), windSpeed, MeasurementSource.FROST, null);
    }

    public static Measurement asMetForecast(LocalDateTime date, Location location, double temperature, double humidity,
            Double windSpeed) {
        return new Measurement(new MeasurementId(date, location), temperature, humidity, windSpeed,
                MeasurementSource.MET_FORECAST, LocalDateTime.now());
    }
}
```

Listing 6.5: Measurement class stored in the database

6.5 Deployment and DevOps

The application is deployed as a web API on Heroku. As DHS uses Fire Risk Model as a library to send measurements in the correct format, the application is deployed via the Heroku CLI [18]. Similarly to Fire Risk Services and the implementations to other services, the DHS source code is hosted on GitHub.

```java
public class MeasurementId implements Serializable {
    @NotNull
    private LocalDateTime timestamp;
    @NotNull
    private Location location;
}
```

Listing 6.6: Measurement class id

Chapter 7

Middleware

The Middleware is a system that makes the Fire Risk Services (FRS) and Fire Risk Model accessible for the end-users in the front-end clients. It is a Spring Boot application with a similar structure as the FRS and the DHS, and it provides a separation of concerns between the front-end logic and the FRS. It handles database connections and fetching of fire risks from the database and from the FRS. It exposes and maintains a database of subscribed locations and associated fire risks, as well as notifying subscribed users when fire risks are high for their locations. In this chapter, we present the Middleware application, its endpoints, and scheduled services.

7.1 System Overview

In broad terms, the Middleware accepts requests from the front-end for fetching locations and fire risks, as well as adding new locations to the DHS, and fetching aggregated fire risk predictions and historical data.

The application is a RESTful web API built with the Spring Boot framework, and share a lot of similarities with the DHS and FRS. It uses a pipeline of *source code -> git -> GitHub Actions -> Heroku Deployment*.

All external dependencies that the application uses (e.g. Gson and Lombok), are declared in a Gradle file. By declaring all external dependencies, the project can be cloned from the git repository and Gradle will download all the dependencies. This is a neat feature to be able to develop the code application from any computer, but it is also a necessity for building the application in a CI environment (GitHub Actions) and for automatic deployments in Heroku.

```
middleware
 ├─ controller
 ├─ dto
 ├─ model
 ├─ repository
 ├─ service
 └─ exception
```

Figure 7.1: File structure of the Middleware project

Figure 7.1 lists the top-level packages in the project. It follows the same best practices as the DHS for structuring Spring Boot web projects. Figure 7.2 gives an overview of the implementation.

The next sections describe the key features of the Middleware in greater details, and how it was modified to facilitate the experiments detailed in Chapter 9. The implementation of the Middleware is available via [14].

7.2 Endpoints

The application exposes services through a set of endpoints and operations. The main endpoints are *location*, *firerisk*, *statistics* and *subscriptions*. Each of them are implemented as a separate REST-controller in the Spring Boot-application, using the folder structure presented in Figure 7.1.

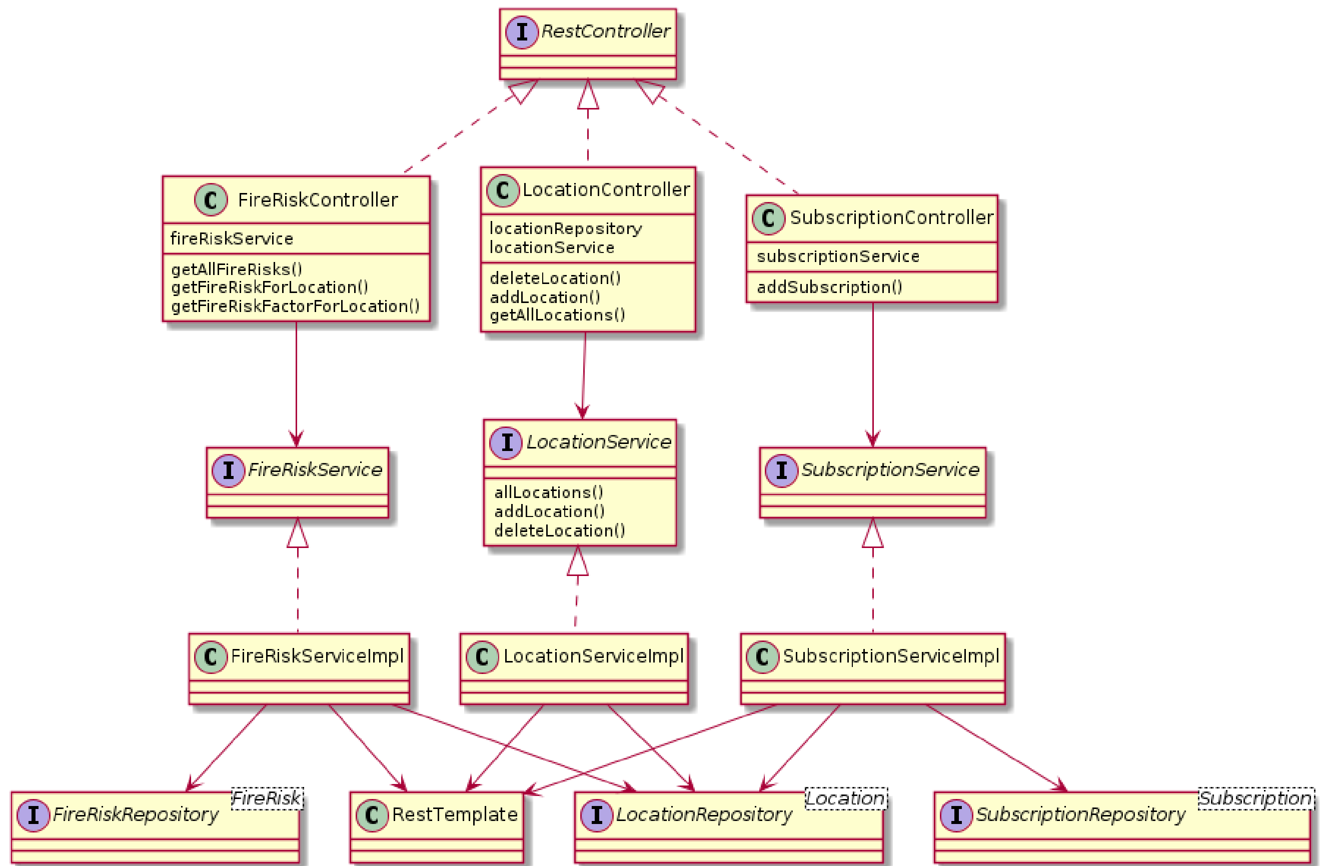

Figure 7.2: Class diagram for the Middleware.

- **GET** /locations
- **POST** /location
- **POST** /firerisk/factor
- **POST** /firerisk/ttf
- **GET** /firerisk/ttf/report.csv
- **GET** /firerisk/ttf/report.csv?latitude={string}longitude={string}

The first four endpoints are used by the Frontend applications. */locations* is used to fetch a list of all the locations that the Middleware keeps track of. It allows the Frontend to fetch firerisks for different locations, and the locations can be presented to the user as a list or on a map, which is detailed more in Chapter 8.

The endpoint returns a JSON-array of location entities converted to strings, and a typical response is shown in Listing 7.1.

```
[
    {
        "latitude": 62.1453,
        "longitude": 6.0748,
        "id": 1,
        "radius": 150
    },
    {
        "latitude": 59.0017,
        "longitude": 3.9472,
        "id": 2,
        "radius": 150
    },
    {
        "latitude": 58.9853,
        "longitude": 11.2748,
        "id": 3,
        "radius": 150
    },
]
```

Listing 7.1: An example JSON body of a request to get all locations

The */firerisk* endpoint takes a location as input (which is provided by the previous endpoint), and returns the current fire risk for that location. If no recent fire risks are stored in the database for that location, a request is sent to the Fire Risk Services (FRS) for calculating a new fire risks. The endpoint returns a fire risk with information about date, status, measurement source, and a list of TTF values, where the first value in the list is the current risk, while the rest is predicted TTF values with one hour intervals up to 9 days into the future. Table 7.1 is an example of such a call, represented in a table format. The values in the first column are the timestamps (with resolution down to hours), and the right column are their corresponding TTF values in minutes.

The last two endpoints are implemented as a way of collecting a larger amount of data from the services for statistical analysis. The usage of these endpoint are to facilitate the analysis done in Chapter 9. The */reportall* endpoint finds all the fire risks in the database that are collected with the Scheduled Services, described in section 7.3, and groups them together by location id and measurement source, and returns a downloadable csv file with each row containing a list of ttf values. The last endpoint is just a filtering for the same functionality, that only collects fire risks for a specific location.

locationId	1
measurementSource	FROST
2021-02-10 01	3.871
2021-02-10 02	3.868
2021-02-10 03	3.861
2021-02-10 04	3.851
2021-02-10 05	3.841
2021-02-10 06	3.835
2021-02-10 07	3.833
2021-02-10 08	3.838
2021-02-10 09	3.846
2021-02-10 10	3.853
2021-02-10 11	3.858
2021-02-10 12	3.861
2021-02-10 13	3.865
2021-02-10 14	3.868
2021-02-10 15	3.870
2021-02-10 16	3.869
2021-02-10 17	3.864
2021-02-10 18	3.857
2021-02-10 19	3.849
2021-02-10 20	3.841
2021-02-10 21	3.832
2021-02-10 22	3.823
2021-02-10 23	3.814
2021-02-11 00	3.804
2021-02-11 01	3.794

Table 7.1: Data stored for a single fire risk request. The full request includes more rows than displayed.

7.3 Scheduled Services

In addition to the RESTful services that the Middleware offers, the Middleware also implements logic that runs at fixed intervals, by fetching fire risks for every location once an hour. This ensures that up-to-date fire risks are available to the end-users, and it is a scalable solution that only increases in resource utilization linearly by the amount of locations that are subscribed to, and not by the number of users.

There are currently two functions running as scheduled services: one is fetching fire risks for all locations with Netatmo as data source, and another with Frost. Both runs once every hour, but are spaced in time to prevent overloading the Fire Risk Services and the Data Harvesting Services. The first runs at the time `X:29` and the other at `X:53`, where X is the hour.

The scheduled times can be arbitrary, but they are based on the scheduled services that the Data Harvesting Services are using to fetch weather data. By running the Netatmo call *after* new data has been fetched for Netatmo data in the Data Harvesting Services, the fire risks that are calculated are as updated as possible. This is equivalent for the call to Frost.

7.4 Deployment and DevOps

The Middleware application is deployed to Heroku as a Gradle project. The Heroku project is connected to the source code, hosted in GitHub, and any time there are changes on the default branch (merges and commits), Heroku rebuilds the application and redeploys it. The data in the database is not affected by deployment process. The deployment process does not invalidate any client sessions. If a user starts a series of requests that first targets the older version of the application, and then the newer one, there should not be any cause of errors. This is partly due to the fact that the handling of the client is stateless. The application does not store or remember any previous interactions with the client, so the client must include any and all details required to complete a request.

```
1  name: Java CI with Gradle
2
3  on:
4    push:
5      branches: [ main ]
6    pull_request:
7      branches: [ main ]
8
9  jobs:
10   build:
11
12     runs-on: ubuntu-latest
13
14     steps:
15     - uses: actions/checkout@v2
16     - name: Set up JDK 1.8
17       uses: actions/setup-java@v1
18       with:
19         java-version: 1.8
20     - name: Grant execute permission for gradlew
21       run: chmod +x gradlew
22     - name: Build with Gradle
23       run: ./gradlew build
```

Listing 7.2: The GitHub Action used for building and testing in CI

Github has a feature called Actions that can be used to build, run and test the source code in a cloud-hosted environment. Heroku integrates well with this feature, such that if a unit test fails during the testing stage, the new application will not be built and deployed on Heroku. Testing is described more in section 7.6. Listing 7.2 show the script that the GitHub Action is running. It specifies which branches in git that it should run on, which environment it should initialize in the cloud environment (Linux Ubuntu with JDK 1.8 and permissions to run gradlew), and what commands to run. The command *run: ./gradlew build* builds the project and runs the unit tests.

7.5 Database

The application is connected to the same database as the FRS and the Data Harvesting Services. This was merely a convenience, rather than a requirement, since the microservices do not read and write to the same tables. The entities that the Middleware stores in the database are *Locations* in Table 7.2 and *Firerisks* in Table 7.3. The fire risk entities are stored with a reference to the locations, to make it possible to retrieve fire risks for specific locations.

Column_name	Type	Length	Nullable
id	numeric	9	no
latitude	float	8	no
longitude	float	8	no
name	varchar	255	yes
radius	float	8	no

Table 7.2: Subscribed Locations SQL table

Column_name	Type	Length	Nullable
id	numeric	9	no
date	datetime	8	yes
location	numeric	9	no
measurement_source	int	4	yes
status	varchar	255	yes
tag	varchar	255	yes
ttfs	text	16	yes

Table 7.3: Firerisks SQL table

The database is a MSSQL database hosted on HVL's servers. It was attempted to use Heroku's free SQL-databases for each of the services, but the free tier is limited to 10.000 rows of data in the database, and storing weather data and fire risks exceeds that within a few days. Following good code practices with low coupling between the components, switching database was fairly easy. The Spring Boot framework uses an `application.properties` file to inject arguments to the run-time environment during startup of the application. The file contain pairs of keys and values, such as sql-dialect, database-url, username, password, and other environment values.

When running on localhost, and during unit-testing, the Middleware utilizes an in-memory database that is instantiated and destroyed between each sessions. This simplifies development, since functionality can be tested with a database, without affecting data in the production database, and bad test-runs will not be kept in successive runs, causing hard-to-debug issues. By using a separate `application.properties` file for testing, the application are unaware of the underlying database structure.

7.6 Testing

By separating logic out from the RESTful services, some of the code can be unit tested in a simple manner. Testing and asserting a certain behavior on the endpoints and scheduled services are more tricky. However, the Spring Framework provides features that simplifies the testing. By using mocking, external dependencies can be ignored, and the code can be tested in isolation.

The Spring Boot framework uses the `@Autowired` annotation to automatically inject dependencies. It can be used for both fields and constructors, but if there is only one constructor, the annotation can be omitted. The dependency injection (DI) can be utilized in test classes as well. By creating mocks for database handlers and rest templates, the controllers and services can be instantiated with their normal constructors, but using mocks instead of the normal dependencies.

The Middleware requests data from the FRS for firerisks through the RestTemplate-class. By mocking it, using the Mockito-library, the requests can have mocked responses, and the tests can assert that the code handles the expected cases. This does not validate that the Middleware and FRS communicate correctly, only that the code that handles the requests and responses operates well on the data it is developed for. In order to test that the systems communicate as intended, larger integration tests have to be made. The unit tests are intended to test the Middleware in isolation.

Listing 7.3 demonstrates a test suite that tests the FireRiskController. The project is configured to use a separate in-memory database for testing og running on localhost, and therefore, the repositories can be injected by the framework instead of being mocked. The RestTemplate is mocked to isolate the code under test.

```java
package com.dynamic.middleware.restservice;

import ...

@SpringBootTest
public class FireRiskServicesTest {

        @Mock
    RestTemplate restTemplate;

    @Autowired
    FireRiskRepository fireRiskRepository;

    @BeforeEach
    void init() {
        fireRiskRepository.deleteAll();
    }


    @Test
    void consumeFireRisk_returnsFireRisk() {
        // arrange
        Location location = getArbitraryLocation();
        String url = GlobalVariables.GET_FIRE_RISK_SERVICES_BASE_URL() + "/firerisk";
        FireRiskResponse mockedResponse = getDefaultFireRiskResponse();
        FireRiskServices fireRiskServices = new FireRiskServices(fireRiskRepository,
            ↪ restTemplate);
        when(restTemplate.postForObject(eq(url), any(),
            ↪ eq(FireRiskResponse.class))).thenReturn(mockedResponse);

        // act

        FireRisk fireRisk = fireRiskServices.consumeGetFireRisk(location,
            ↪ MeasurementSource.FROST, "my tag");

        // assert
        assertThat(fireRisk).isNotNull();
        assertThat(fireRisk.getTag()).isEqualTo("my tag");

    }

}
```

Listing 7.3: Unit testing with mocks

Chapter 8

Front-end Single Page Web Application

The Fire Risk Services (FRS) provides a service for computing fire risks for locations. The Middleware provides a set of endpoints for adding and removing locations that fire risks should be calculated for, as well as sending notifications when risks are above a threshold. The front-end detailed in this chapter focuses on informing the risks in an intuitive way, and to provide an graphical interface for the services in the Middleware. The source code for this project can be found via [13] and the deployed application at `https://firerisk-frontend.herokuapp.com/`

8.1 Project Structure

The application is a single page web application, implemented with React [34] and TypeScript [50], which was chosen for their popularity in the industry and because of prior knowledge of the frameworks and languages. The folder structure is shown in Figure 8.1 as a reference point for the further discussion of the technologies and how the front-end is intended to work. Some entries are not included for brevity.

package.json contains all dependencies used in the project, and other configurations required to run the project. Yarn [53] is used to build and serve the project on localhost (or in a deployed environment), starting from *index.tsx* as the entry point.

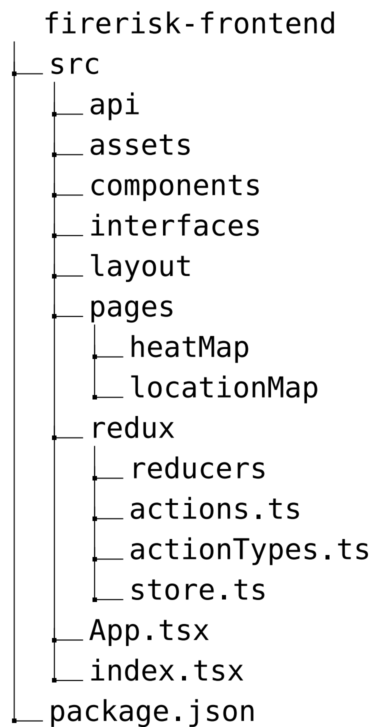

Figure 8.1: Directory for the web application

8.1.1 React

React is a JavaScript library for building interactive user interfaces. It uses a special syntax called Javascript XML (JSX) combined with JavaScript to build components of the application. Each component returns a single JSX element which can contain other JSX elements and normal HTML syntax that will be rendered. Each component can maintain its own state (but is not required to), and the React library will re-render a component and its children upon state changes. By only re-rendering portions of the component tree instead of the entire application, a static web page can become dynamic without the need for refreshing the entire page or navigating to other pages provided by a server.

An example of a React Component is Listing 8.1. The component is one of the top-level components, and even though it contains few lines of code, it has a lot of logic packed under the hood. The return statement (line 11-22), is a single JSX element, where the Router element is the root. The Router allows for URL manipulation in the application. Components inside a Router can use the URL to fetch values, and the

URL can be changed to let the Router know that some other component should be rendered. The Switch component (line 15-19) does just that. Depending on the value of the URL, it renders the first Route component which has a path parameter that matches the value. The last Route element has a path value of just /, which means that it will match any values.

The Switch component is wrapped in the Provider component, line 13-20. The Provider is part of the React Redux [36] library, which provides a global store for all components that are below it in the component tree. This acts like a local database that components can access and will not be lost between re-renders. In this project, the Redux store maintains locations and fire risks provided by the Middleware.

```
1  import React from "react";
2  import { BrowserRouter as Router, Switch, Route } from "react-router-dom";
3  import Navbar from "./layout/Navbar";
4  import Timeline from "./pages/timeline/Timeline";
5  import { Provider } from "react-redux";
6  import store from "./redux/store";
7  import LocationMap from "./pages/locationMap/LocationMap";
8  import HeatMap from "./pages/heatMap/HeatMap";
9
10 function Routes(): JSX.Element {
11   return (
12     <Router>
13       <Provider store={store}>
14         <Navbar />
15         <Switch>
16           <Route path="/timeline/:id" component={Timeline} />
17           <Route path="/locations" component={LocationMap} />
18           <Route path="/" component={HeatMap} />
19         </Switch>
20       </Provider>
21     </Router>
22   );
23 }
24
25 export default Routes;
```

Listing 8.1: A React component used for navigation

8.1.2 TypeScript

The application is written in TypeScript [50], which is a typed superset of JavaScript. By having strong typing and type inference at development time, a lot of errors, traditionally associated with JavaScript, like type errors or wrong shape of objects (methods and fields are different than expected). Type errors can still occur, especially

where the application has to integrate with other non-typed JavaScript libraries, and wrong assumption are made, like declaring an interface of an object expected from the Middleware, where the fields does not match reality. The benefits are still quite large; the internal parts of the application should be without type-errors. Another large benefit is that, once an error in the interface is detected, and updated, all implementation of that interface with any errors will be failed by the compiler, thus refactoring code becomes safer.

Line 10 in Listing 8.1 declares that the component should return a *JSX.Element*. If something else is returned instead, like a string, number or void, the compiler will not compile the component.

8.1.3 Leaflet & React Leaflet

Leaflet [22] is a JavaScript library for interactive maps. React Leaflet [35] provides bindings between between React and Leaflet. The library does not provide the map data itself, but rather displays an overlay over maps. For this project OpenStreetMap [33] is used because it easily integrates with Leaflet, and, in addition, is free and open to use, as long as their name is credited within the map.

Listing 8.2 shows the implementation of a map component in a React application. The *heatMap* page describe in figure 8.1 is implemented using the same components but adds on a lot more logic and other presentational components. It renders a map component that starts with its center at the geographical center of Norway, and initial values, like zoom level, are provided.

```
import React from "react";
import { MapContainer, TileLayer } from "react-leaflet";
import { LatLng } from "leaflet";

const openMapAttribution = '&copy; <a
    ↪ href="http://osm.org/copyright">OpenStreetMap</a> contributors';
const openMapUrl = "https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png";
const centerOfNorway = new LatLng(60.472024, 8.468946)

function Map() {

  return (
    <MapContainer
    center={centerOfNorway}
    zoom={10}
    minZoom={6}
    >
        <TileLayer
            url={openMapUrl}
            attribution={openMapAttribution}
        />
    </MapContainer>
  );
}

export default Map;
```

Listing 8.2: React component for displaying map.

8.2 Features

The main page for the web application is a map, implemented with Leaflet, as described in the previous section. The component uses Redux to fetch all the locations that the Middleware keeps track of, and if the user navigates the map over any of those locations, a request is made to fetch the current fire risk factor for that location.

When there is a fire risk for a location available, and the location is within the boundaries of the map, the risk factor will be shown as a colour on the map grid. The colours are mapped to the Table 4.1, where *Low* is green, *Medium* is yellow, *High* is orange, *Very High* is red, and finally *Extreme* is dark red. Figure 8.2 shows an example of this.

When a user clicks on a grid cell, a component appears to either display information about current risks for that location, or a form to add that location to the list of locations that the Middleware uses. In the current prototype, it is also possible to

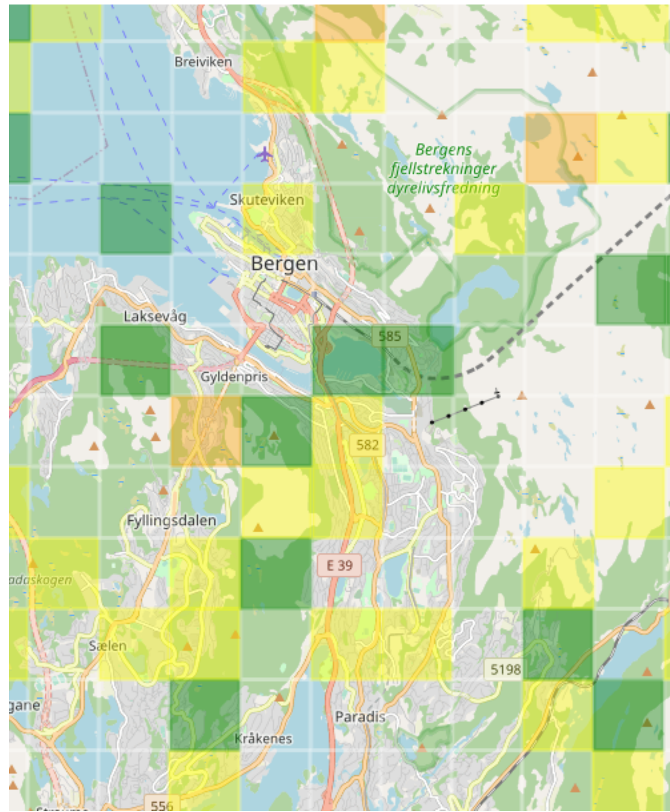

Figure 8.2: Map with a superimposed grid of fire risk factors.

delete locations from the Middleware, but that functionality should be restricted to either some form of admin users, or to the user who added the location. Figure 8.3 shows screenshots of an example of this.

Figure 8.3: Screenshots of adding and removing locations.

The grid that is superimposed on the map, is intended to be fixed relative to the map. Each grid cell is 1-by-1km, and when the user pans around or zooms in or out, it has to be recalculated. It is necessary because the grid can not be set directly from the coordinates or the map itself. The distance between 1 degree of longitude is different

depending on latitude (111km at 0° latitude, and 55km at 60°), and the map uses mercator projections to project the spherical earth to a 2D-map, so locations further north and south are more stretched.

When the grid needs to be recalculated, it starts at an fixed point (center of Norway), and increments a coordinate in the direction of the current position in exact 1km-intervals. When it has reached the south-west corner of the map boundaries, it iterates over the visible map and calculates an array of positions that are 1km away from each other. Each position is then rendered as a translucent (or with a colour if they contain a fire risk) rectangle on the map. The function for calculating a position at an relative distance of another position is is shown in Listing 8.3, and if called with an input of **deltaCoordinate([60.0, 5.0], 1000, 0)**, it would return a location that is 1000 meters east of the position *[60.0, 5.0]*. The calculation is not completely correct, as it makes some assumptions about the radius of the earth, which is not a fixed values. Since the earth is an oblate spheroid, the radius gets slightly smaller as we approach the north and south pole. The errors are small enough to not be a critical problem for this application.

```
1  const DEGREES_TO_RADIANS = 180.0 / Math.PI;
2  const RADIANS_TO_DEGREES = 1.0 / DEGREES_TO_RADIANS;
3  const r_earth = 6371000.0;
4
5  export const deltaCoordinate = (
6    [latitude, longitude]: LatLngTuple,
7    deltaX: number,
8    deltaY: number
9  ): LatLngTuple => {
10   const newLatitude = latitude + (deltaY / r_earth) * DEGREES_TO_RADIANS;
11   const newLongitude =
12     longitude +
13     ((deltaX / r_earth) * DEGREES_TO_RADIANS) /
14       Math.cos(latitude * RADIANS_TO_DEGREES);
15   return [newLatitude, newLongitude];
16 };
```

Listing 8.3: Calculate a position offset from another.

8.3 Fetching Data from the Middleware

The front-end application needs data about locations, subscriptions and fire risks, in order to be useful. The Middleware has, a described, endpoints that can be used

to fetch this data. The front-end application does this by dispatching actions with a library called Redux thunks [38]. The library allows asynchronous manipulation of the global redux store, making it suitable for tasks like fetching data from the Middleware API.

8.4 DevOps

The React components uses Javascript and a special JSX syntax to allow for dynamic loading of the web page, but in the end, it all compiles down to HTML and Javascript, and it can be tested in the same way as described for the other systems.

Certain Javascript functions are pure – idempotent and without side effects – and can easily be tested through unit tests. A framework called Jest [20] has been used for this project. It uses the keyword *Describe* to wrap a test suite, and *it* for a single unit test. Jest comes with a command line tool that detects all test files that are either in a __*tests*__-directory, or uses the file extension *.test.js* or *.spec.js* anywhere in the project.

Listing 8.4 shows a test for fetching a list of subscribed locations. In order to test the logic in isolation without relying on the Middleware, the fetch-package that is used for requests to the Middleware is mocked. The tests asserts the expected results for normal behaviour (a list of locations is returned), as well as when the fetch fails.

The pattern of arranging the tests in the format of arrange/act/assert is independent of the programming language, and is used here in the same way as the tests implemented in Java in the Spring Boot applications. An example test using Jest is shown in Listing 8.4.

```
import { getAllLocations } from "../getAllLocations";
import { Location } from "../../interfaces/Location";
import fetch from "jest-fetch-mock";

describe("getAllLocations", () => {
  beforeEach(() => {
    fetch.resetMocks();
  });

  it("should fetch all locations", async () => {
    const mockedResponse: Location[] = [
      { id: 1, longitude: 10, latitude: 10, name: "ten" },
      { id: 2, longitude: 20, latitude: 20, name: "twenty" },
    ];
    fetch.mockResponseOnce(JSON.stringify(mockedResponse));

    const locations: Location[] = await getAllLocations();

    expect(locations).toHaveLength(2);
    expect(locations[0]).toEqual(mockedResponse[0]);
    expect(locations[1]).toEqual(mockedResponse[1]);
  });

  it("should return empty array if fetch fails", async () => {
    fetch.mockReject(() => Promise.reject("a generic error message"));

    const locations: Location[] = await getAllLocations();

    expect(locations).toHaveLength(0);
  });
});
```

Listing 8.4: Testing fetch with mocked responses.

Chapter 9

Experiments and Evaluation

In this chapter, a series of experiments are conducted to investigate the correctness, usefulness and scalability of the system that have been developed. During the month of February, fire risks were calculated for 74 locations, each hour, every day. In addition, fire risks were predicted every hour up to ten days into the future, and stored together with its corresponding fire risk. Experiment 1 and 2 is an analysis of the computed data, and Experiment 3, 4 and 5 tests specific parts of the system developed.

9.1 Locations

In order to get realistic and varied data for the experiments, the locations were selected to achieve an even distribution across the country. The largest towns in each county were added, as well as locations of special interest, mainly wooden structures worthy of preservation, like "Bryggen i Bergen". Appendix A lists all of the locations with their coordinates used for the experiments. Figure 9.1 shows the distribution of locations.

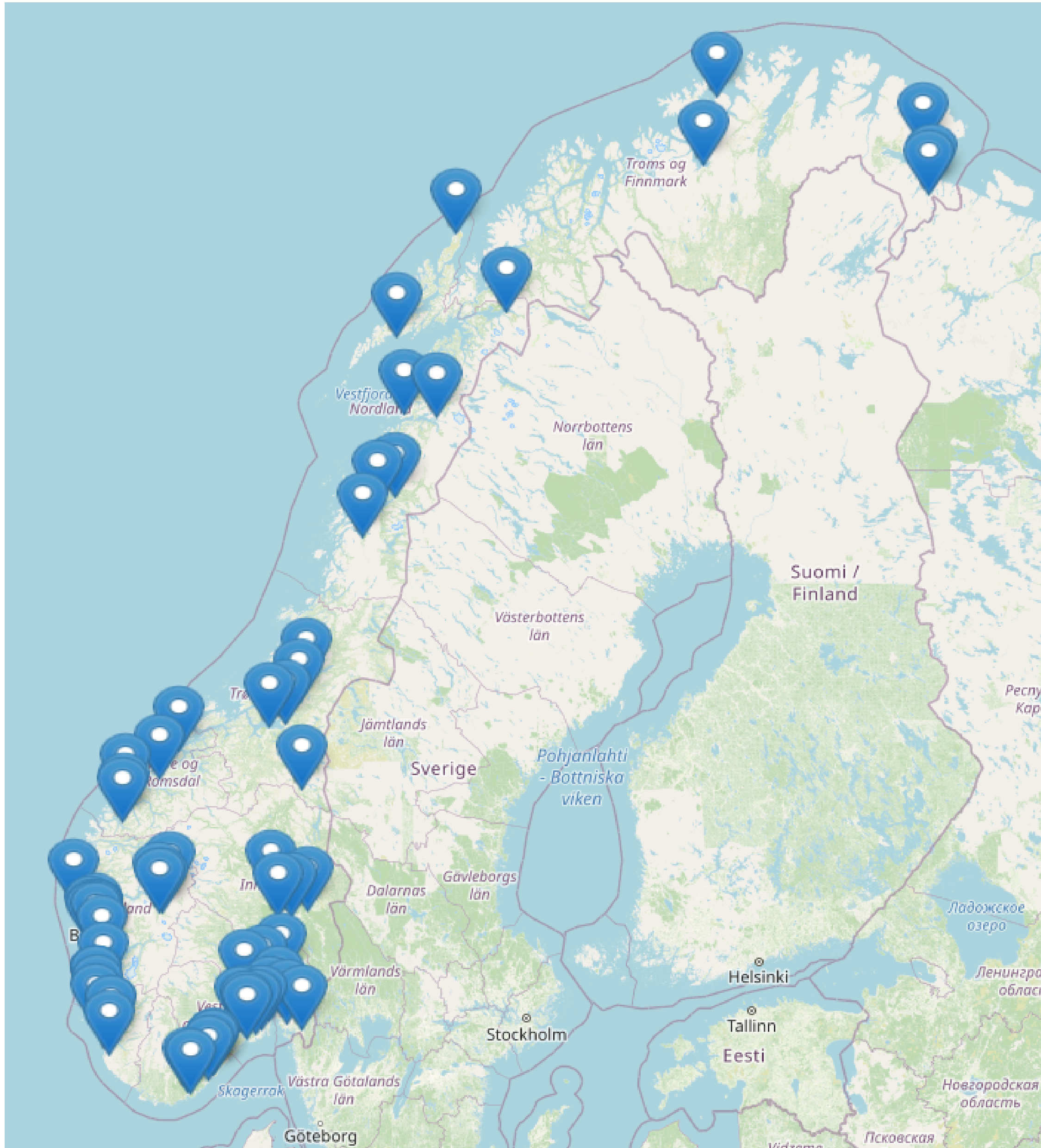

Figure 9.1: Distribution of the 74 locations used for the experiments.

9.2 Experiment 1: Predicting Time to Flashover

In this experiment, the predicted Time To Flash-over (TTF) based on weather forecasts were compared to TTFs based on weather measurements. Collecting fire risks and predictions for many different locations provided a large dataset for analysis.

9.2.1 Collecting Fire Risks

By using the Middleware, the FRS, and the DHS, fire risks for each location were fetched every hour. Using Scheduled Services, as described in section 7.3, the Middleware requested fire risks for each location from the Fire Risk Services. This was done twice every hour: once to calculate fire risks based on Netatmo stations, and once based on MET stations. The FRS fetched weather data from the DHS and returned the modelled risks computed from the weather data. The weather observations contain both weather forecasts and measurements. As such, for every hour, an array of predicted TTFs was included in the fire risks returned to the Middleware, with up to ten days of predictions.

The data collected is in the time range from 10th of February to 10th of March, 2021. For the 74 locations selected, the amount of data points can be calculated as:

$$n = (measurementFrequency * 24) * measurementSources * locations * days$$

With a measurement frequency of one (one measurement per hour), two measurement sources (Frost and Netatmo), 74 locations and 28 days, the number of computed fire risks are:

$$n = (1 * 24) * 2 * 74 * 28 = 99456$$

For each fire risk collected, an array of TTFs is stored. The first TTF in the array is the TTF for the time of the calculation, while the rest are predicted TTFs based on

weather forecasts. Each array contains up to 10 days worth of TTF predictions. With a frequency of 1 per hour, this yields an array of size 240. Each entry in this array consist of an object containing a timestamp and the modelled TTF value in minutes.

By multiplying the number of fire risks collected during the experiment with the number of TTF values for each fire risk, we obtain the number of TTF values that were collected for this experiment:

$$99456 * 240 = 23,869,440$$

These 23 million data points are not stored in the database as efficiently as they could have been. However, network latency and storage space is not a concern for this experiment. Experiment 5, described in section 9.6, goes into more details about resource management and other optimizations.

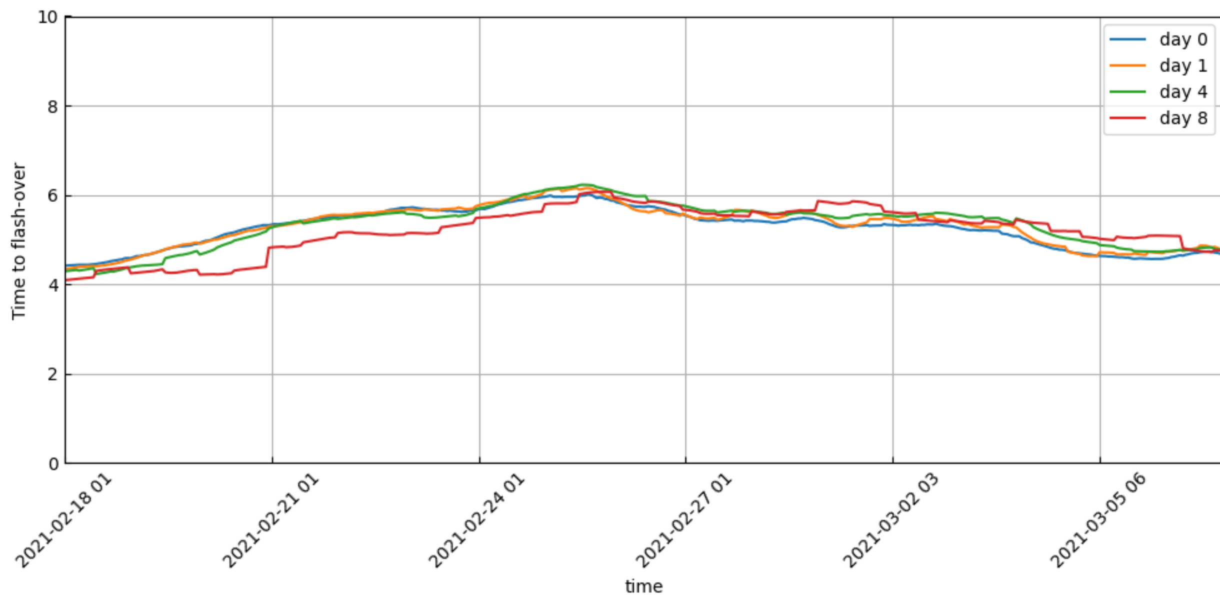

Figure 9.2: TTF based on measurements and forecasts for Oslo.

9.2.2 Analysing the Data

As mentioned, each fire risk has a list of TTF values and each value has a corresponding timestamp. The weather observations used to model it comes from measured

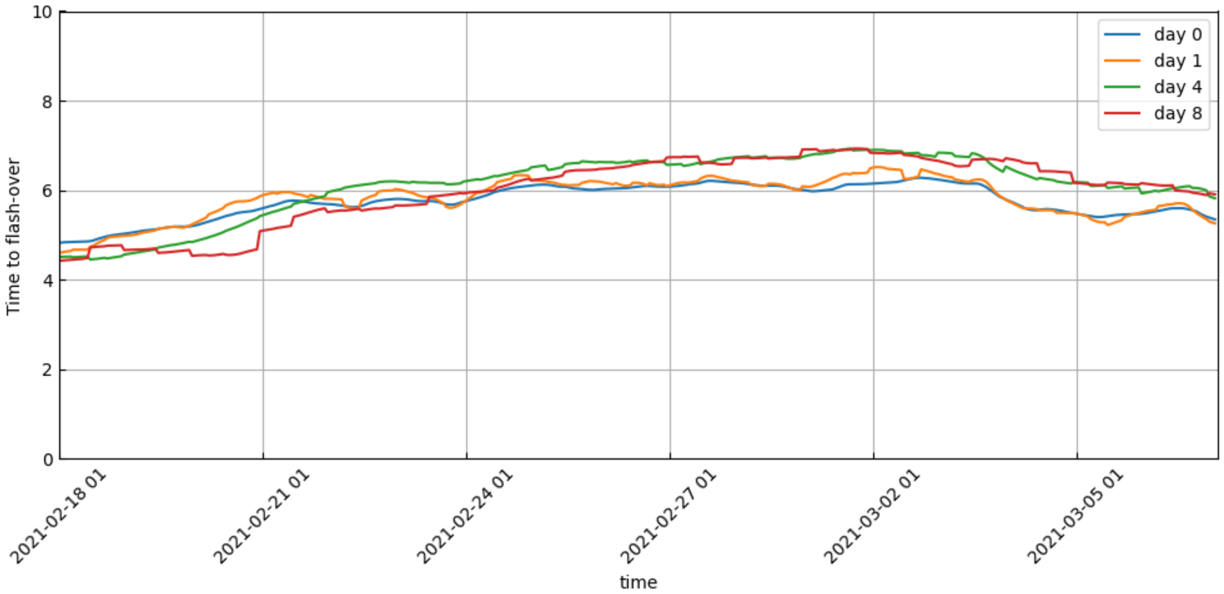

Figure 9.3: TTF based on measurements and forecasts for Volda.

observations and not from weather forecasts. The subsequent TTF values are predictions for how the TTF will change based on weather forecasts on an hourly basis.

Plotting the first TTF value for each fire risk and plot it, and for each hour similarly find what TTF value that was predicted for this timestamp 24 hours ago, 48 hours ago, all the way up to 8 days ago, we obtain graphs similar to Figure 9.3 and 9.2. The blue line is the TTF values only based on measurements, and are referenced as day 0. Moving from one day of predictions up to eight days, it can be see that the lines fluctuate more and differ from the fire risk based on measured values.

The deviation between TTF based on measurements and TTF based on forecasts is to be expected, as the TTF values uses uncertain weather forecasts to model its predictions. If different weather occurs instead of what was predicted, the fire risks will, as a consequence, have differences between those that are based on measured data and those that use an incorrect forecasts. The orange line, which is only predicting TTF values 24 hours into the future, will naturally stay closer to the blue line, as weather forecasts normally gets more accurate for shorter periods of time into the future. Even if the weather forecast is dramatically different from the actual weather, it takes time for moisture to be transported in and out of the wooden structures, which constraints the rate of change of fuel moisture content.

To figure out how much the predicted risks deviates from the risks computed only on measured weather data, their relative differences can be calculated using the following formula:

$$d_r = \frac{|x - y|}{(\frac{|x+y|}{2})}$$

where x and y are the TTF values computed on historic weather data and what it was predicted to be at that time. Multiplying the values by 100, we get the relative difference as percentages, shown in Figure 9.4. In the figure, the differences are calculated for predictions 1, 4 and 8 days into the future. The relative differences can be calculated for all of the other days as well, but that would render the figure unintelligible.

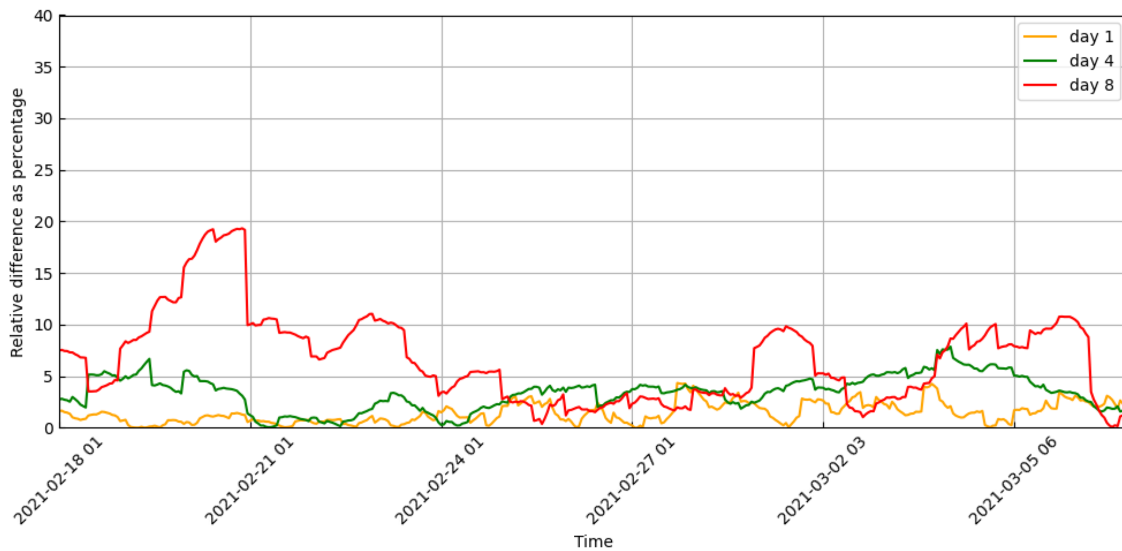

Figure 9.4: Relative differences between predicted and current TTFs.

As seen in Figure 9.4, the relative difference for the 1-day predictions never go above 5%, while the 8-days prediction goes above 15% relative difference multiple times. The figure is only for a single location, and the values are not necessarily representative for the other locations, but as we will see in the next sections, and as might not be a surprise, shorter times of predictions will in general yield lower difference to the risks based only on measured data.

9.2.3 Largest difference

In the experiment, the location with the largest difference between measured and predicted TTF values is Otternes in Aurland.

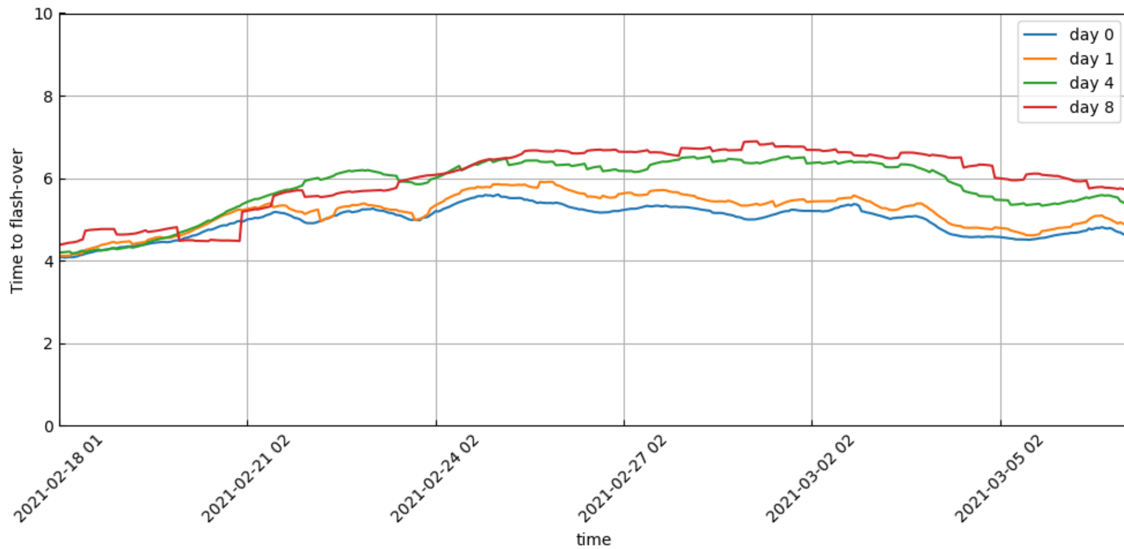

Figure 9.5: Predictions for Aurland; Location with largest relative difference.

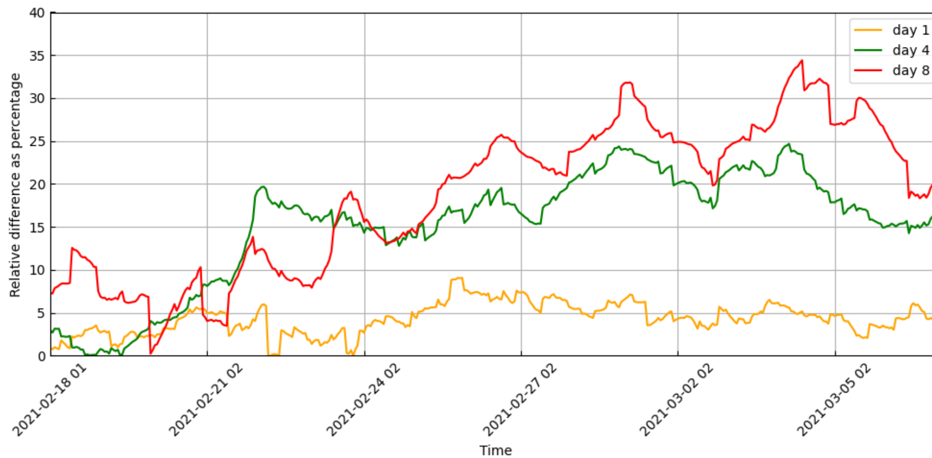

Figure 9.6: Relative difference of predictions for Aurland.

The *day 8* prediction is off by almost 40%. Figure 9.5 and 9.6 show the predictions, and how far off they are, relative to the *day 0* values. It is worth noting that the predicted value is higher than the actual value, thus it predicts that the risks are lower than they actually were.

9.2.4 Prediction deviation

Comparing the differences between the predicted TTF to the value computed only based on measured weather data, and group them by how many hours into the future they make predictions for, the standard deviation, average absolute difference and average relative difference can be plotted.

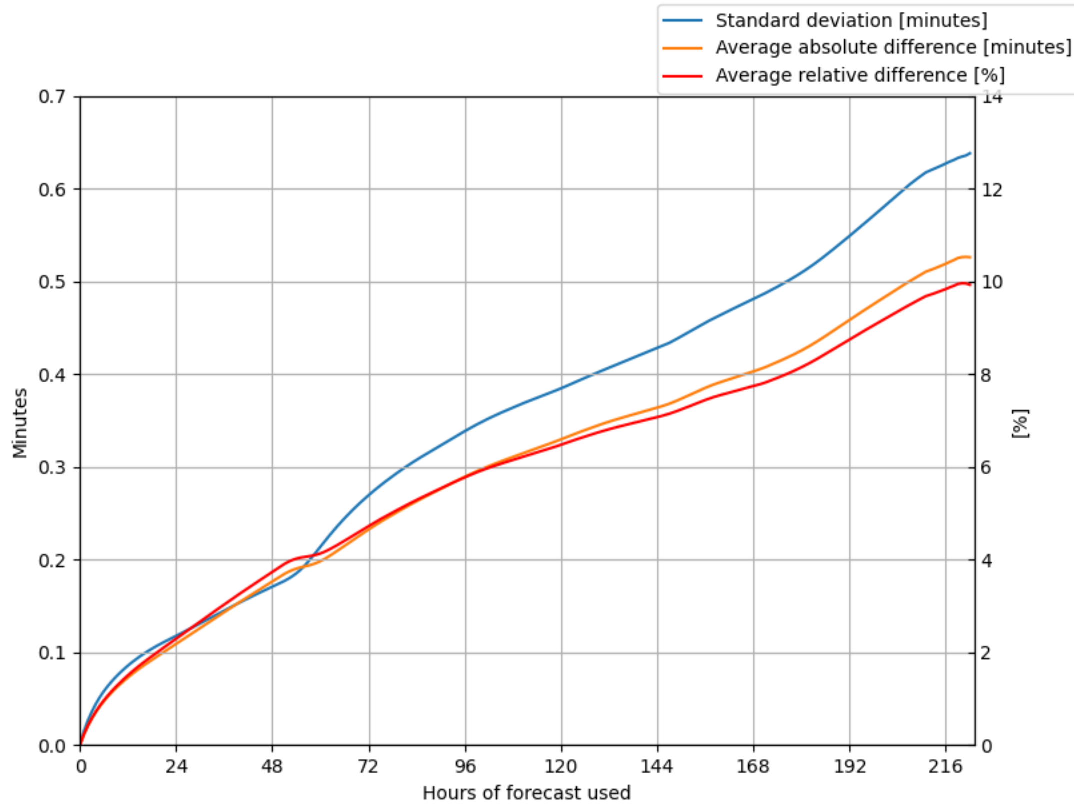

Figure 9.7: Standard deviation and average absolute and relative difference for predicted TTFs.

Figure 9.7 shows the results for this calculation on all 74 locations. From the graph, we can see that using predictions up to 72 hours yields an average absolute difference below 0.25 minutes.

9.3 Experiment 2: TTF Weather Data Sources

The purpose of this experiment is to investigate how different weather sources affect the modelled TTF values. Using the same dataset as in Experiment 1 (section 9.2), the fire risks can be sorted by measurement source and compared against each other per location. By comparing the modelled fire risks for each location with both sources, they follow similar trends, as expected.

9.3.1 Data

As detailed in section 6, there are two measurement sources implemented in the Data Harvesting Service, and the same sources are used for this experiment: Frost and Netatmo. They are both used to compute fire risks on an hourly basis, and both are used for the same locations. This means that they can be directly compared against each other. The application uses uncalibrated Netatmo stations as the application does not have access to specific calibrated stations. As the predicted fire risks use the same weather forecasts regardless of the selected measurement source, comparing the predictions are not of interest; given the same forecasts, they will converge towards the same predicted TTF values. Therefore, only the fire risks that use exclusively historical data – the first TTF value for each fire risk – are used. Figures 9.8 and 9.9 shows the two different sources for two of the locations used during the experiment.

If the computed TTF values for the two different sources are compared against each other for all of the locations, the difference between Frost and Netatmo becomes more apparent. Figure 9.10 shows the size of the differences as a histogram. Here, the difference are found by taking $TTFFROST - TTFNETATMO$. A negative value can be interpreted as the value from Frost being lower than the value from Netatmo.

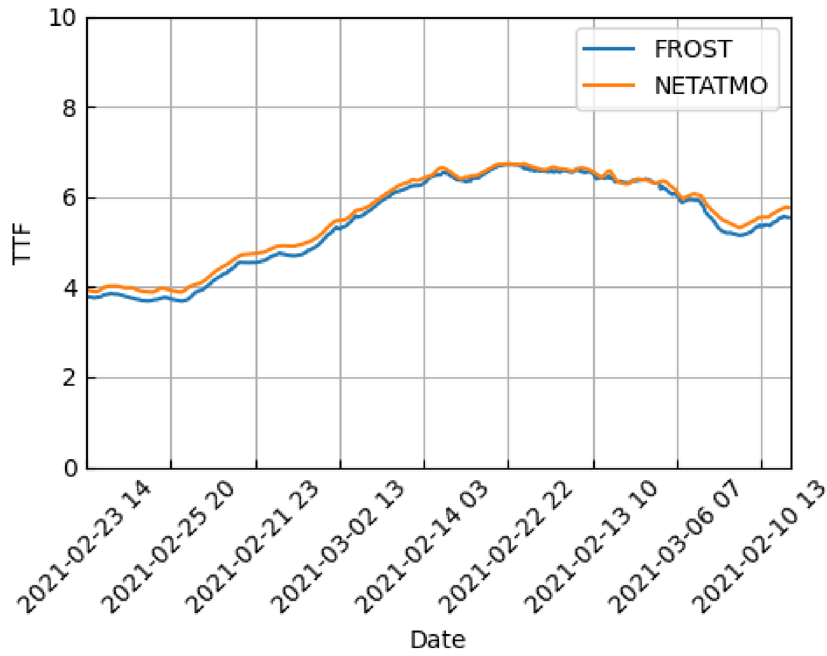

Figure 9.8: Frost and Netatmo compared with each other at Bryne.

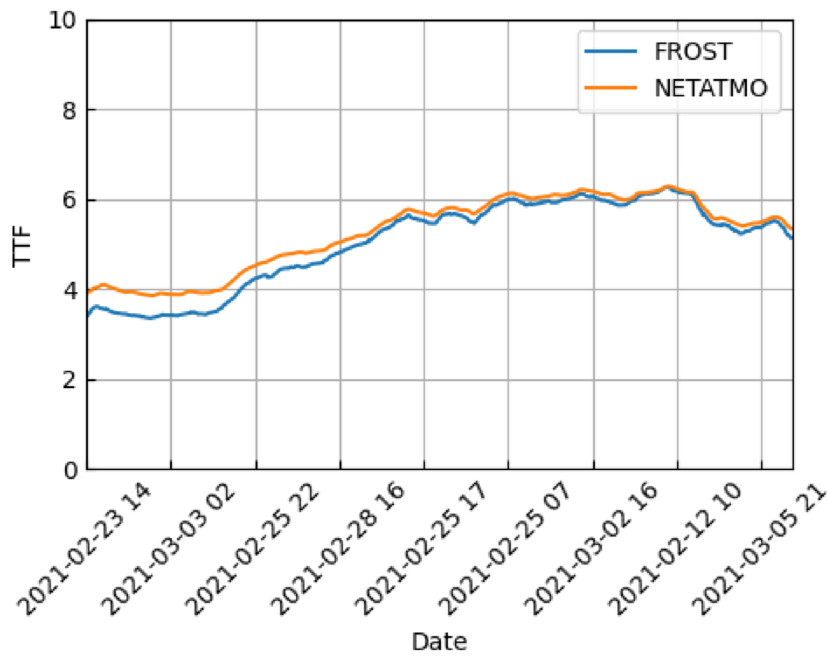

Figure 9.9: Frost and Netatmo compared with each other at Volda.

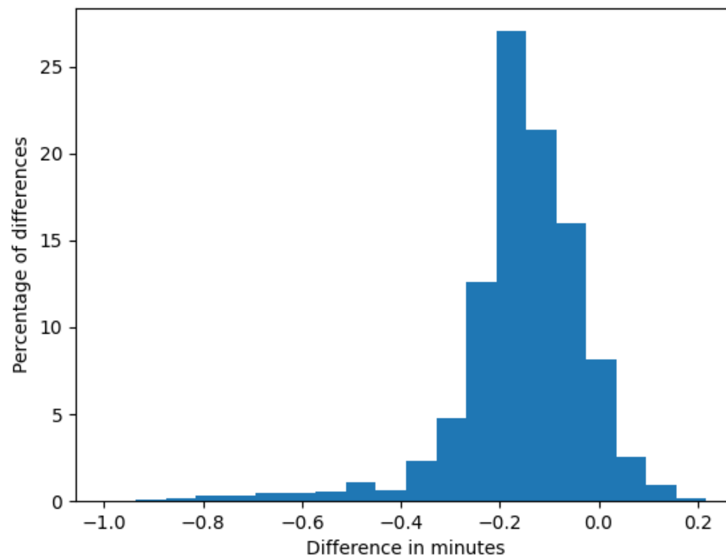

Figure 9.10: Differences between weather data sources.

9.3.2 Analysing the Data

It is interesting to note that the Frost data consistently reports lower TTF values than the Netatmo data, even though the differences are negligible. For most of the computations during the experiment, TTFs computed using weather data from Frost lead to values that are less than 0.3 minutes lower than TTFs computed using weather data from Netatmo.

9.4 Experiment 3: Historical Fire Risk 2013 - 2021

Running the system on historical weather data is a way to validate the implementation of the Fire Risk Model, as one can check if it outputs reasonable values when running for an extended time. By calculating Time To Flash-over for three locations from 1st of October 2013 to 1st of April 2021, the FRM is run every time step all year, across multiple seasons, to check for model stability and seasonal variations.

9.4.1 Data

As detailed in Section 2.2.1, The Norwegian Meteorological Institute's Frost REST API can be used to request several years worth of weather data. Hourly measurement were requested and given as input to the model. Three locations of the 74 were chosen for this experiment: Lærdal, Røros and Flatanger. Lærdal and Flatanger have had large historical conflagrations, both in January of 2014, while the town of Røros is a UNESCO World Heritage site [39], known for old wooden houses dating back to the 17th century. October of 2013 was chosen as start date to capture the whole winter of 2013/2014, which includes the fires of Lærdal and Flatanger. Eight years should additionally capture yearly variations.

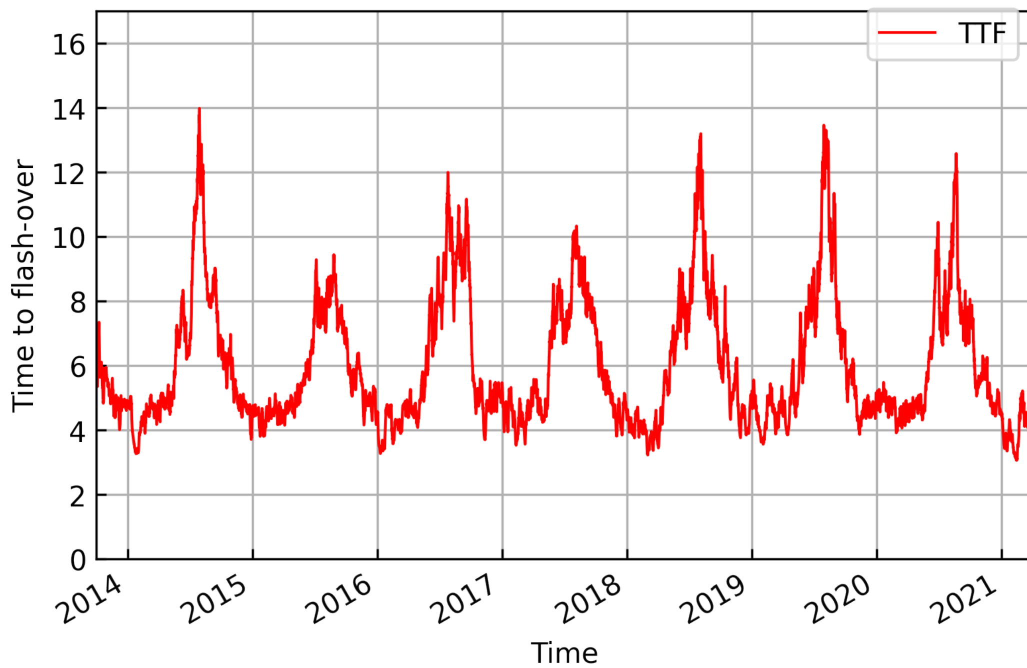

Figure 9.11: Time To Flash-over at Lærdal from 2014 to 2021.

[h]

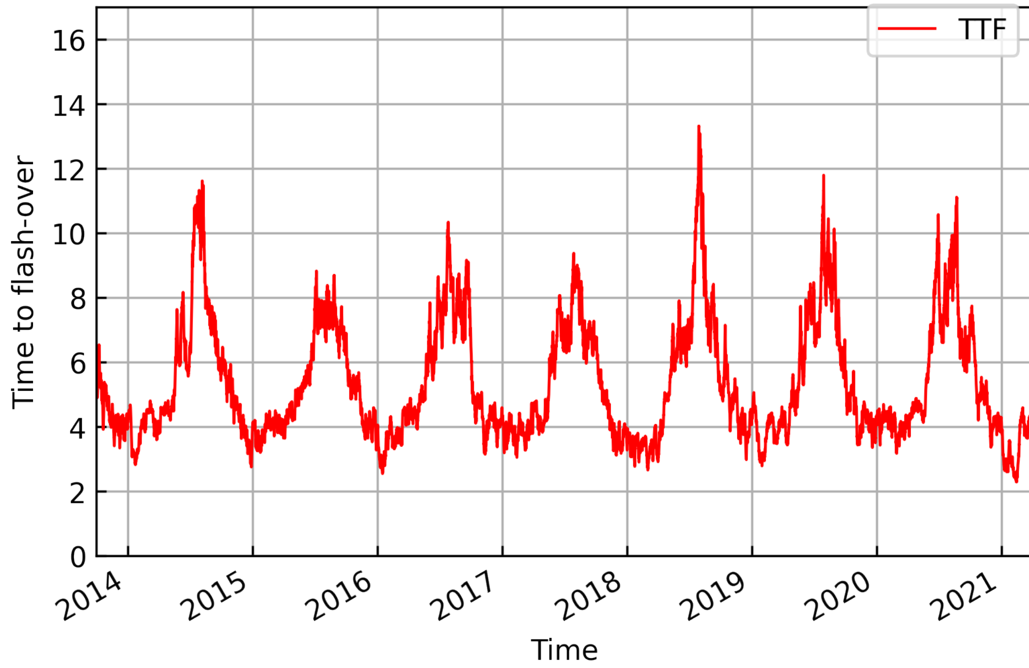

Figure 9.12: Time To Flash-over at Røros from 2014 to 2021.

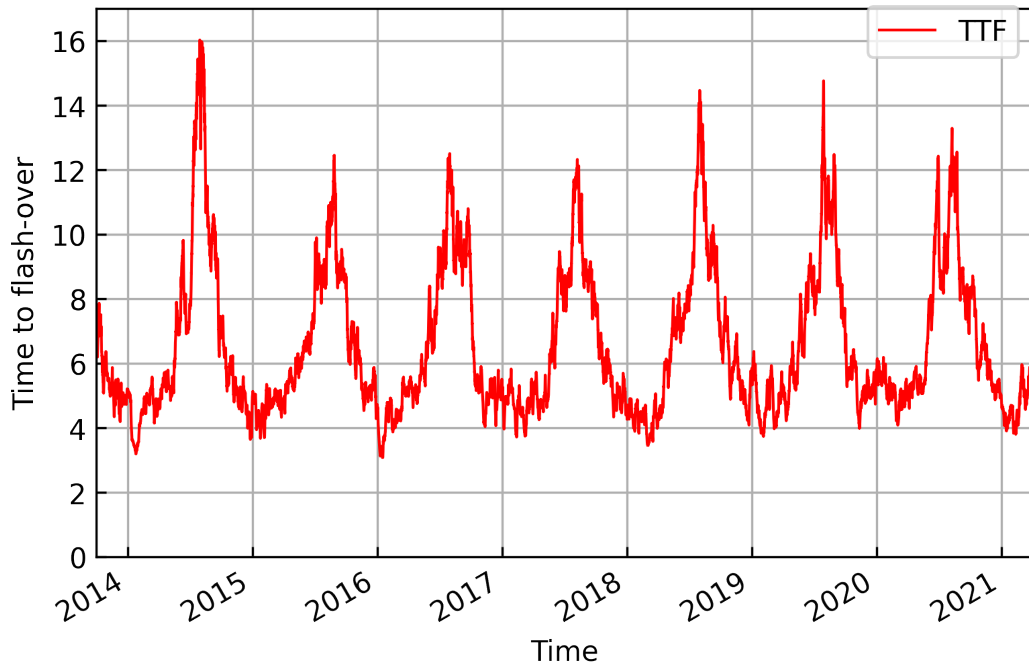

Figure 9.13: Time To Flash-over at Flatanger from 2014 to 2021.

9.4.2 Analysing the Data

In Figure 9.11, 9.12, and 9.13, Time To Flash-over is plotted to visualize it during the period. Not included is five days in September, starting 26th of September at 12 AM to 12 AM at 1st of October, to help initialize the model. Looking at the figure, there are distinct differences spikes of high TTF during the summer, while the winter months are fairly stable low, around 4-5 minutes TTF.

After running, the results were analysed to figure out how many model iterations had Time To Flash-over under four minutes for each winter, to figure out each location's most dangerous winters. The Fire Risk Model has 720 seconds, or 12 minutes, between each iteration, and as such, we count every instance of below four minutes TTF as 12 minutes. Number of iterations per day $= (60\ minutes/12\ minutes) * 24\ hours = 120$. Flatanger had at most 23 days during the winter of 2015/2016, Lærdal had 50 days during the winter of 2017/2018, while Røros had a total of 130 days out of a total of 182 days during the winter of 2017/2018. This analysis shows that at Røros more than $2/3$s of the winter of 2017/2018 were below four minutes TTF. From these results, there does not seem to be any specific about the winter of 2013/2014, when the fires in Lærdal and Flatanger happened, based on Time To Flash-over alone. However, including wind could change the picture.

Some statistics of these three location can be seen in Table 9.14, 9.15, and 9.16 based on quarterly fire risks. Winter is defined as December to February, spring is March to May, summer is June to August and autumn is September to November. The Time To Flash-over varied the most during the summer and autumn with a standard deviation of over 1 for all locations. However, the average TTF is higher with over 7 minutes during the summer and 5-6 minutes during the autumn.

9.5 Experiment 4: Winter of 2020/2021

Experiment 4 looks at the 74 locations used in Experiment 1 and Experiment 2 for an analysis of the winter of 2020/2021, 1st of October to 31st of March. It uses MET's Frost API and hourly measurements over the period for all 74 locations as input to the model, similar to Experiment 9.4.

Season	Max	Min	Average	Std
Winter (Dec. - Feb.)	5.56	3.07	4.43	0.52
Spring (Mar. - May)	8.53	3.23	4.92	0.82
Summer (Jun. - Aug.)	13.99	5.41	8.65	1.67
Autumn (Oct. - Nov.)	11.17	3.71	6.16	1.36

Figure 9.14: Seasonal statistics from Lærdal 2013 to 2021.

Season	Max	Min	Average	Std
Winter (Dec. - Feb.)	6.38	3.09	4.73	0.60
Spring (Mar. - May)	8.30	3.46	5.32	0.83
Summer (Jun. - Aug.)	16.03	5.86	9.64	2.04
Autumn (Oct. - Nov.)	12.14	4.00	6.77	1.60

Figure 9.15: Seasonal statistics from Flatanger 2013 to 2021.

Season	Max	Min	Average	Std
Winter (Dec. - Feb.)	5.06	2.29	3.78	0.50
Spring (Mar. - May)	7.74	2.66	4.51	0.80
Summer (Jun. - Aug.)	13.32	5.20	7.86	1.44
Autumn (Oct. - Nov.)	9.72	3.12	5.46	1.20

Figure 9.16: Seasonal statistics from Røros 2013 to 2021.

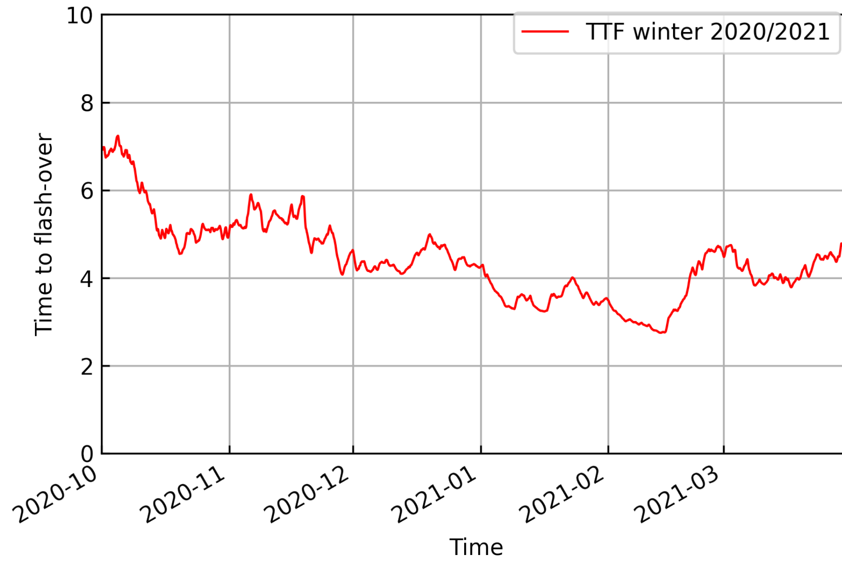

Figure 9.17: Time To Flash-over in Aurland during the winter of 2020/2021.

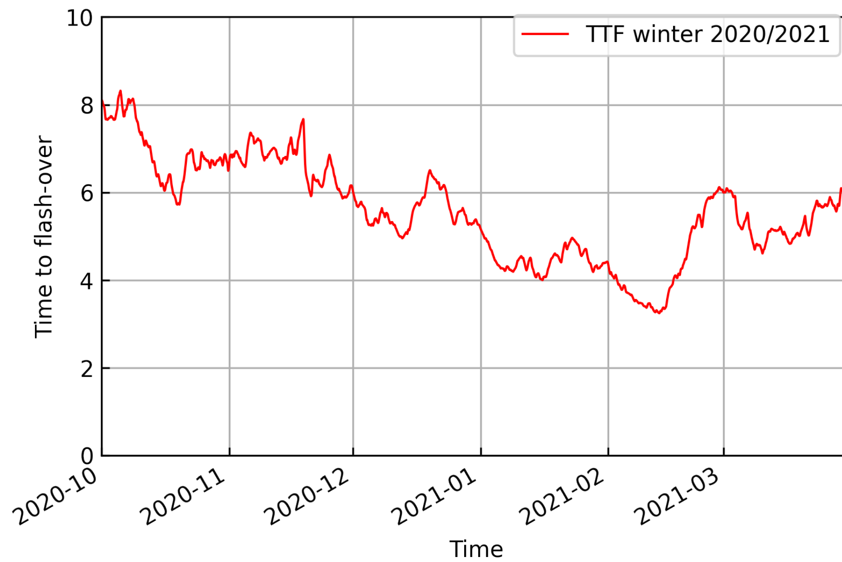

Figure 9.18: Time To Flash-over at Bryggen in Bergen during the winter of 2020/2021.

9.5.1 Analysing the Data

The results from three selected locations are visualized in Figure 9.17, 9.18, and 9.19. To analyse the data, periods of consecutive low TTF were identified. The majority of locations, 43 out of 74, had at least a week of consecutive TTF at less than 3.5 minutes.

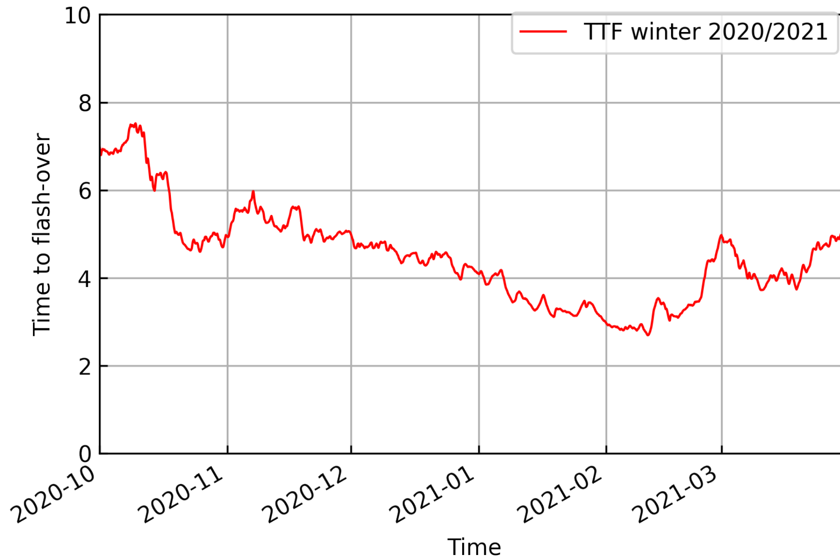

Figure 9.19: Time To Flash-over at Mo i Rana during the winter of 2020/2021.

The most prominent was Mo i Rana, just south of the Arctic circle, with 27 days at less than 3.5 minutes, and even 11 days less than 3 minutes.

Month	Max	Min	Average	Std
Oct. 2020	10.37	4.21	6.98	1.19
Nov. 2020	8.83	3.37	6.30	0.97
Dec. 2020	7.22	3.11	5.31	0.72
Jan. 2021	5.82	2.61	4.20	0.55
Feb. 2021	6.38	2.29	4.05	0.85
Mar. 2021	6.92	3.19	4.95	0.69

Figure 9.20: Monthly statistics from all 74 locations during the winter of 2020/2021 combined.

In Figure 9.20 statistics from the months of October 2020 to March 2021 based on all 74 location is shown. Time To Flash-over averages at about 4 minutes during the months of January and February and locations with under 3 minutes, compared to October and November at 6-7 minutes on average.

9.6 Experiment 5: Data Storage and Computation Time

Given that the system is built as cloud-hosted applications, some considerations to the network latency and usage, CPU usage, and storage requirements should be made. By analysing the resource management required to conduct Experiment 1 and 2, estimates on how the systems will scale can be made. Bottlenecks may not be detected during the experiment, such as third-party API rate limits, but general metrics of the resource management can still be useful for an informed decision on hosting and future improvements.

9.6.1 Processing utilization

When a fire risk is requested from the Middleware to the FRS, the observation data is fetched from the Data Harvesting Service and a stopwatch object is used to to measure how long it takes for the model to calculate fire risks for the observations. Since all the data from the database is fetched before the stopwatch starts, and it stops before the data is returned to the Middleware, this can give a good indication on the processing requirements of the Fire Risk Model. All the data is loaded in memory, so there are no I/O operations to disk or database during the modelling phase.

The stopwatch measures time in *nanoseconds*, because when using *milliseconds*, the result took more often than not *0ms*. Stopwatch timings are visualised in Figure 9.21. Given that the calculations are so fast, there is no need to focus further on this metric at this time.

9.6.2 System test

One of the operations that the Middleware performs is a scheduled service that requests fire risks from the FRS for all of the locations, and store it in the database. In order to complete this service, a sequence of actions has to take place:

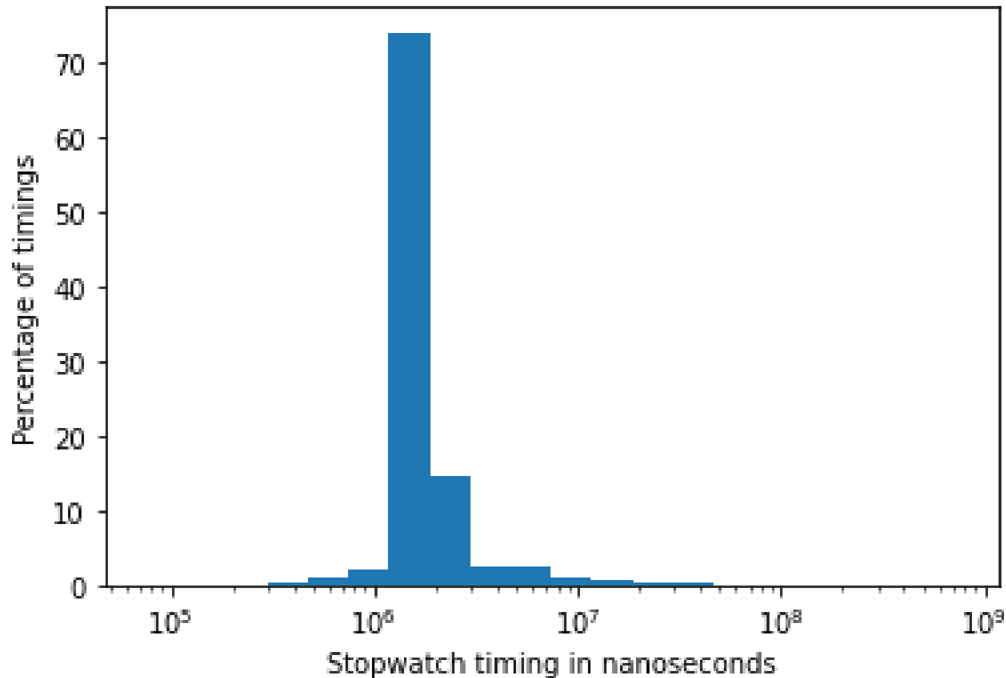

Figure 9.21: Distribution of CPU time used for fire risk calculations.

1. All locations are fetched from the database
2. For each location, a fire risk is requested from the FRS
3. For each request the FRS requests weather data from DHS, then it calculates and returns a firerisk
4. The firerisk is stored in the database

Three consecutive runs of this service yielded a runtime of *245433ms*, *243457ms* and *242573ms*, with an average of *243821ms*, or roughly four minutes. The runtime of this service relies on multiple factors: response time from the database, response time from the Data Harvesting Service, and the time it takes to calculate the results and transport the requests and responses across the services. Since this is not a service that the end-user has to wait for, time is not critical. However, as the number of locations in the database increase, the service will become slower.

The main bottleneck is waiting for the external weather data services. This can be mitigated by pre-fetching and storing the weather data in the database, but that

would come with the cost of added storage space, so there is a trade-off that needs to be considered. For the time being, a runtime of around 4 minutes are well within acceptable limits for a service to only runs every hour (or less frequent than that).

When an end-user accessing the web application requests a fire risk for a specific location, the number of actions required to complete the action is a bit simpler, as described by diagram 9.22.

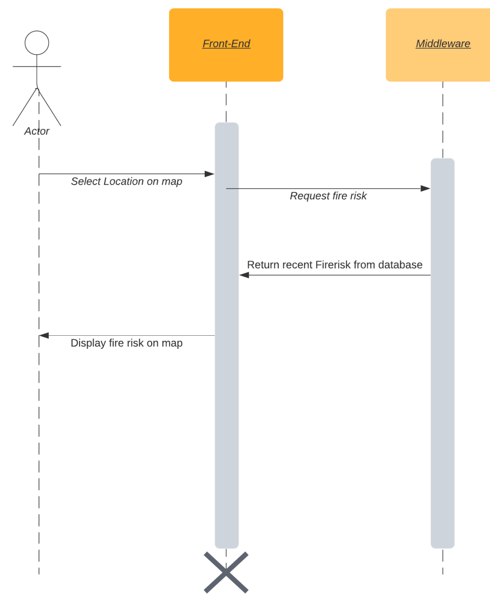

Figure 9.22: Sequence of actions for fetching stored fire risks.

Some testing on this scenario reveals that the request is usually completed within 2-3 seconds. This is a fairly high waiting time for a web service that an end-user waits for, but it is not unreasonably high. This can easily be mitigated in the Middleware by simply caching the results and keeping a map of fire risks for the different locations in memory.

9.7 Conclusion

After conducting these series of experiments, some conclusions can be made. These experiments are summarised in Table 9.1.

n	name	description
1	TTF Development	Development of TTF at day x, with a decreasing window of forecast weather.
2	TTF - Weather sources	Development of TTF at day x, as described above, but utilizing different weather sources.
3	Historical Fire Risk 2014 - 2021	Validate the Fire Risk Services based on historical weather data from three locations to validate the Fire Risk Model and analyse the outputted Time To Flash-over over time.
4	Historical Fire Risk - Winter of 2020/2021	Validate the Fire Risk Services based on historical weather data based on all 74 locations to validate the Fire Risk Model and analyse the outputted Time To Flash-over over the winter of 2020/2021.
5	Data Storage and Computation Time	Consideration of data storage and computation time.

Table 9.1: List of experiments.

By using the DHS, FRS and the Middleware, it was possible to fetch weather data, model risks on measured data and predictions on forecasts, and store the risks in a database in a way that they could later be used for analysis.

By comparing early predictions to later risks based only on measured weather data, it was shown that the predicted values were sufficiently accurate to be used for predicting fire risks. For a prediction 72 hour earlier compared to the risk based on

measured weather data, the standard deviation was below 0.3 minutes. This seems to fit the resolution of the forecast data, which is hourly the first three days, and then every sixth hours for the next seven days. As the model linearly interpolates between each observation, a resolution of six hours might not reflect the changes during the day.

Using Frost data, the modelled TTF values are consistently lower, compared to results modelled on Netatmo data. Overall, both sources predicts similar fire risks. As mentioned in section 6, measurements from Frost are collected from the closest MET station, with no guarantee to how close it to the desired location. Netatmo however, finds the stations within a polygon of 10 km by 10 km with the requested location in the middle. As a result, comparing a MET more than 5 km away will not necessarily be representative, due to variations in local weather.

The model seem to run stable when run over several years. The Fire Risk Model outputs higher TTF during the summer, while a stable low during the winters. Low TTF is not a huge problem in itself. However, if the low Time To Flash-over persists over a longer period, then the likelihood of an ignition happening increases.

Based on TTF alone, January of 2014 does not seem extreme and periods of low TTF occur every winter. Nevertheless, some winters have longer periods of low TTF, such as the winter of 2020/2021.

The Fire Risk Model (FRM) is sufficiently fast, to the point where the modelling itself is negligible compared to the preparation of the data. The services that fetches fire risks at scheduled frequencies, uses around 4 minutes to fetch fire risks for 74 locations. The most time-consuming part is waiting for the external weather data and reading/writing from the database. The requests for fire risks at a single location only reads data from the database, and therefore only takes a couple of seconds without any form of caching.

Chapter 10

Conclusion and Future Work

In this thesis, the fire risk model developed by Log [23] has been implemented, and services required for subscribing to locations and fetching weather data has been developed. The cloud-hosted microservices can be used from a front-end application, and during the month of February, the services were used to compute fire risks and predictions. Computed data was then analysed in accordance with predefined experiments in order to further investigate proposed research questions.

Five applications have been created in this thesis: Data Harvesting Service, Fire Risk Services, Fire Risk Model, Middleware and the Front-end web application. Users are able to use the system through the web application, which request fire risk from the Middleware. If the fire risk for the location is stored in the database, it is returned, if not, the Middleware request a fire risk from the Fire Risk Services. The FRS use the Fire Risk Model combined with weather data from the Data Harvesting Service to compute a fire risk and return it to the Middleware.

10.1 Conclusion on experiments

To validate the application, five quantitative experiments were conducted; one to compare weather forecast to measurements, one to compare the two sources of measurements, Netatmo and The Norwegian Meteorological Institute's Frost API, two different experiments to validate the Fire Risk Model on historical data over several years

and the winter of 2020/2021, and the final experiment looked at different aspects of the system itself, such as data storage and computation time.

Experiment 1 compared fire risks based on weather measurement with fire risks based on weather forecasts. By sampling a wide distribution of locations, it was demonstrated that the standard deviation of time to flashover predicted three days into the future were below 0.3 minutes. The deviation increases for prediction further into the future, partly because weather forecasts gets less accurate further into the future, as well as the resolution of forecasts going from 1 hour windows to 6 hour windows somewhere around 72 hours (3 days). Furthermore, the decreasing temporal resolution results in inaccurate modelling as the linear interpolation of the model excludes natural daytime variations.

For fire risks predicted all the way up to eight days into the future, a specific prediction with relative differences of 35% compared to the fire risks computed from measured weather data were found. This turns out to more than a full minute of difference in predicted TTF and 'actual' TTF. By restricting the predictions to 72 hours, sufficiently accurate predictions are achieved. This in turn, should allow for fire departments to be notified prior to upcoming risk peaks, allowing proactive measures to be initiated.

Experiment 2 compared the fire risks computed from the two different weather sources implemented by the Data Harvesting Service. The differences between the two sources were small enough that using one or the other have little impact on the actual values computed. By having multiple weather sources that yields similar results, the DHS can use the best suited source to completing requests, based on availability of data from the sources for a given location without the Fire Risk Services needing to know (or care) about what sources have been used. These can be chosen based on proximity to the location.

In Experiment 3 and 4, the FRS and DHS are used to compute fire risks over many years for specific locations. A repeating pattern emerges from the graphs: lower TTF, in the winter months, and higher in the summer months. During the winter of 2014/2015, the TTF values in Lærdalsøyri were low, but appears in accordance with the seasonal variations seen during the considered period, winter 2013/2014 until winter 2021.

And finally, in Experiment 5, resource usage of the microservices were investigated. The CPU usage needed to calculate fire risks were measured and demonstrated to be negligible. Fetching new fire risks for 74 locations takes a couple of minutes, and the time taken will increase linearly if more locations are added. Most of the time is spent waiting for external weather services to provide the requested data.

10.2 Research Questions Revisited

In this section, we return to the research questions proposed in the introduction of this thesis, and reflect if the artefacts developed in combination with the experiments conducted, can be used to answer the questions.

RQ1: How can different weather services be combined to obtain one unified data service, that can provide weather measurements and forecasts for the fire risk model?
The developed applications combines weather data from Netatmo and MET via their respective REST APIs. These services have their own format, which is parsed and mapped to Data Harvesting Service's internal representation. The Fire Risk Services can then request this data to compute a fire risk using the Fire Risk Model. These are requested periodically to save time when requested.

RQ2: How can a software architecture that implements a complex fire risk notification system composed of multiple independent cloud-hosted services be designed?
The developed application uses a microservices architecture design. The system is split into multiple separate applications, each with its own responsibilities. The applications are hosted on the cloud service Heroku, but with individual cloud hosting configurations.

RQ3: Can the fire risk model be combined with weather forecast data to predict accurate fire risk indications?
Stokkenes [44] has already demonstrated that weather data can be used for historic fire risk computations, and by using the services developed in this thesis, the model can be extended with weather forecasts to predict fire risk in the near-future. Experiment 1 concludes that the predicted fire risks are accurate for up to 72 hours.

10.3 Threats to validity

The microservices developed in this thesis does fairly complex tasks. Numeric computations are done on weather data using a mathematical model, and location data and other data is parsed from one service to the other. In this section, we discuss potential threats to the validity of the experiments we have conducted and the conclusions we have drawn from them.

The fire risk model developed in [23] is based on a generic living space, and makes some assumptions about air volume, water produced, thickness of the wooden panels, and air exchanged with the outside. All these factors affect how fast humidity can be transported into and out of the wooden panels, which in turn determines how the outside weather affects the Time To Flash-over. The assumptions are necessary to compute the risks, but it also means that if all the houses in an area have qualities that leads to higher or lower indoor relative humidity, the computed risks will not be accurate. Hence, a generic approach to "houses" might be unsuited, but for stakeholders with good knowledge of fire and risk analysis, the model can still be an important tool.

The model was translated from a set of equations in an Excel spreadsheet to a Java library. During that process, bugs and errors may have been introduced. The risks of that happening have been mitigated by using the data output from the spreadsheet as a basis for unit-tests. The output from the model implemented in Java has also produced reasonable outputs during the experiments.

Stokkenes [44] demonstrated that weather data could be used to accurately model indoor humidity. This is fundamental to the implemented model, in order to develop a system that can provide fire risks for any location in Norway, without having on-site measurements.

Another threat to the validity of the fire risks, is that for any given location, the exact weather is not known. The measurement sources provide open APIs to collect weather data from the nearest devices, and some clever combination of multiple devices can be used to interpolate the weather for a location. For instance, if location B does not have a measurement device, but is mid-way between location A and C, which have devices, it can seem reasonable to take the average between them to get the weather at location B. Nevertheless, given the geographic differences in Norway, location A, B, and C can have fundamentally different weather patterns.

10.4 Discussion

The microservices and front-end application developed in this thesis has been designed with two conflicting interests; To collect as much current data as possible during an extended period of time for the experiments, and to work as a self-contained system that cleans up old data automatically, such that the data stored in the database does not need to be manually removed every time too much data is stored. For the limited time of the experiments, it was possible to store all the fire risks and request new weather data from the external sources every hour, but this is not a very scalable solution. When the experiments are not running, a scheduled service deletes all fire risks older than a month.

The Middleware can also cache recent fire risks fetched for the front-end application. If the same data is requested multiple times from the same (or different) front-end users, the cached data can be returned instead of requesting new fire risks from the Fire Risk Services.

Based on the results from Experiment 1, section 9.2, forecast for up to 3 days ahead can reliably be used. Consequently, DHS does not need to collect new measurements every hour. With the Frost API, it is possible to request previously measured weather observations, and, as such, the DHS could request measured data every 6 hours, 12 hours or even once a day and use the forecast for the hours in-between. If the measurements are requested every 6 hours: 00:00, 06:00, 12:00, and 18:00, a fire risk computed at 14:00 could use measurements up to 12:00, and for 13:00 and later use the forecast. This will not work when using Netatmo however, as the API only lets one request the current measurement. By requesting weather data less frequently, the DHS saves resource usage by fewer API requests and less time processing the response. In the current version, the Data Harvesting Service collect data every hour, and the Middleware request an updated Fire Risk Model iteration afterwards. If the DHS collects less frequently, the FRM iterations would not need to be recomputed as often too. By strategically choosing the intervals of the data collection and at which times it runs, up-to-date measurements and forecasts can still be available at critical moments, while saving computational time and resource usage.

An example of this, is for the usage of the system by a fire department. If the data is harvested every 6 hours, at hours 00, 06, 12 and 18, new data will be available each morning, and also updated data at the middle of the work-day.

10.5 Future Work

One aspect of the system, specifically the Data Harvesting Service, that could improve weather data quality is how to choose the weather data source. The Norwegian Meteorological Institute's Frost API is a quality controlled source, but the stations are spread out. One option is to do interpolation on multiple weather stations based on proximity.

With the Fire Risk Services current usage of the Fire Risk Model, it requests five days of weather data measurements to initialize the model in addition to weather forecast, and then runs the model on all the data with every fire risk request. An alternative is to store the state of the model after it is initialized on weather measurements, which can be used to start the model from when the next request happens. This way, the FRS only model the five days once, and whenever it receives a request, the FRM only need to be run on data collected since previous request and the forecast. This reduces the amount of stored measurements significantly.

For usage of Netatmo as a data source, the application has a set request for stations within a 10-by-10 km polygon with four corners and does not widen this if the are few or none stations.

All endpoints are currently open to all of the Internet. Authorization should be added to avoid malicious users. Particularly the endpoints to add or remove locations are sensitive and ought-to only be available to specific users with admin permission.

An experiment to validate that the modelled RH (and the TTF that is calculated on RH) matches the actual values, can be conducted. Similar to Stokkenes' [44] work. By using Netatmo stations, a small scale experiment can be performed. This experiment is not a proper validation of the model itself, but it can be a good indication of whether the weather data accurately predicts indoor RH.

For locations that are not close to a measurement device, it may be more accurate to use the near-future weather forecast for that location as the actual historical data input to the model for calculating fire risks instead of attempting to uses the nearest measurement device. The closest devices might be a long distance away, or one could average, interpolate, or extrapolate the measured data between multiple devices. One of the major challenges is determining if weather forecasts are actually correct, and/or by which margin they are incorrect.

List of Figures

2.1 Requesting a Netatmo token in Postman 21

2.2 Refreshing a Netatmo token in Postman. 22

3.1 A birds eye view of the software architecture of the system. 29

3.2 Data flow for fetching a fire risk for a location. 30

5.1 Class diagram for the FRS. 47

6.1 File structure of Data Harvesting Service 52

6.2 Class diagram for the DHS. 52

6.3 Usage of relevant weather data 54

7.1 File structure of the Middleware project 61

7.2 Class diagram for the Middleware. 62

8.1 Directory for the web application 71

8.2 Map with a superimposed grid of fire risk factors. 75

8.3 Screenshots of adding and removing locations. 75

9.1 Distribution of the 74 locations used for the experiments. 80
9.2 TTF based on measurements and forecasts for Oslo. 82
9.3 TTF based on measurements and forecasts for Volda. 83
9.4 Relative differences between predicted and current TTFs. 84
9.5 Predictions for Aurland; Location with largest relative difference. 85
9.6 Relative difference of predictions for Aurland. 85
9.7 Standard deviation and average absolute and relative difference for predicted TTFs. 86
9.8 Frost and Netatmo compared with each other at Bryne. 88
9.9 Frost and Netatmo compared with each other at Volda. 88
9.10 Differences between weather data sources. 89
9.11 Time To Flash-over at Lærdal from 2014 to 2021. 90
9.12 Time To Flash-over at Røros from 2014 to 2021. 91
9.13 Time To Flash-over at Flatanger from 2014 to 2021. 91
9.14 Seasonal statistics from Lærdal 2013 to 2021. 93
9.15 Seasonal statistics from Flatanger 2013 to 2021. 93
9.16 Seasonal statistics from Røros 2013 to 2021. 93
9.17 Time To Flash-over in Aurland during the winter of 2020/2021. 94
9.18 Time To Flash-over at Bryggen in Bergen during the winter of 2020/2021. 94
9.19 Time To Flash-over at Mo i Rana during the winter of 2020/2021. 95
9.20 Monthly statistics from all 74 locations during the winter of 2020/2021 combined. 95
9.21 Distribution of CPU time used for fire risk calculations. 97
9.22 Sequence of actions for fetching stored fire risks. 98

List of Tables

4.1 Fire risk categories 44

7.1 Data stored for a single fire risk request. The full request includes more rows than displayed. 64

7.2 Subscribed Locations SQL table 67

7.3 Firerisks SQL table 67

9.1 List of experiments. 99

A.1 List of locations. 122

Listings

2.1 Example of a JSON response body from the MET forecast REST API 14
2.2 Example of a JSON response from the Frost nearby stations endpoint . . . 16
2.3 Example of a JSON response from the Frost available timeseries endpoint 18
2.4 Example of a JSON response from the Frost observations endpoint 20
2.5 Response from requesting a Netatmo access token. 22
2.6 Measurements from Netatmo's public data endpoint 24
4.1 Implementation of setting the initial values. 35
4.2 Method for setting initial water concetration. 36
4.3 Air exchange beta 36
4.4 Calculation of $\dot{c}_{wall}$ 37
4.5 Calculation of $\dot{m}_{wall}$ 37
4.6 Calculation of $\dot{c}_{ac}$ 38
4.7 Model current step. 39
4.8 Updating the water concentration in the differnent layers 40
4.9 Updating the water concentration inside the house 41
4.10 Updating relative humidity inside the house 41
4.11 Calculation of Time To Flash-over. 42
4.12 Calculation of Time To Flash-over based on Fuel Moisture Content. 42
4.13 Calculation of Fire Risk Factor. 43
4.14 GitHub Action for deploying new package to GitHub Packages. 45
5.1 Usage of the Fire Risk Model in the Fire Risk Services 49
6.1 Requesting measurements and forecast from the database. 54
6.2 An example JSON body of a request to add a location. 55
6.3 Collecting historical measurements from Frost. 56
6.4 A scheduled service as a cron job collecting Netatmo measurement. 57
6.5 Measurement class stored in the database 58

6.6 Measurement class id 59
7.1 An example JSON body of a request to get all locations 63
7.2 The GitHub Action used for building and testing in CI 66
7.3 Unit testing with mocks 69
8.1 A React component used for navigation 72
8.2 React component for displaying map. 74
8.3 Calculate a position offset from another. 76
8.4 Testing fetch with mocked responses. 78

Bibliography

[1] Vardis Anezakis et al. “A Hybrid Soft Computing Approach Producing Robust Forest Fire Risk Indices”. In: *A Hybrid Soft Computing Approach Producing Robust Forest Fire Risk Indicies*. Vol. 475. Sept. 2016. doi: 10.1007/978-3-319-44944-9_17.

[2] L. Bodrozic, D. Stipanicev, and M. Stula. “Agent based data collecting in a forest fire monitoring system”. In: (2006), pp. 326–330.

[3] N. Cheney, James Gould, and Wendy Anderson. “Prediction of Fire Spread in Grasslands”. In: *International Journal of Wildland Fire - INT J WILDLAND FIRE* 8 (Mar. 1998). doi: 10.1071/WF9980001.

[4] Paulo Cortez and Aníbal de Jesus Raimundo Morais. *A data mining approach to predict forest fires using meteorological data*. 2007.

[5] Daegil. “Personal email to Netatmo regarding the filter option in their API”. unpublished. Feb. 2021.

[6] *Data and products available from MET*. 2017 (accessed October 23, 2020). url: https://www.met.no/en/free-meteorological-data/Data-and-products-available-from-MET.

[7] D. P. Dee et al. “The ERA-Interim reanalysis: configuration and performance of the data assimilation system”. In: *Quarterly Journal of the Royal Meteorological Society* 137.656 (2011), pp. 553–597. doi: 10.1002/qj.828. eprint: https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1002/qj.828. url: https://rmets.onlinelibrary.wiley.com/doi/abs/10.1002/qj.828.

[8] Khac Phong Do et al. “Spatial Interpolation and Assimilation Methods for Satellite and Ground Meteorological Data in Vietnam”. In: (2015).

[9] Dougal Drysdale. *An Introduction to Fire Dynamics: Third Edition*. John Wiley & Sons, Ltd, 2011.

[10] *ECMWF Who We Are*. accessed April 23, 2021. url: `https://www.ecmwf.int/en/about/who-we-are`.

[11] Martin Fowler and James Lewis. *Microservices, a definition of this new architectural term*. url: `https://martinfowler.com/articles/microservices.html`.

[12] *GitHub Data Harvesting Service*. accessed May 31, 2021. url: `https://github.com/selabhvl/data-harvester`.

[13] *GitHub Fire Risk Front-End*. accessed May 31, 2021. url: `https://github.com/Evjenth/fireriskfrontend`.

[14] *GitHub Fire Risk Middleware*. accessed May 31, 2021. url: `https://github.com/selabhvl/middleware`.

[15] *GitHub Fire Risk Model*. accessed May 31, 2021. url: `https://github.com/selabhvl/fireriskmodel`.

[16] *GitHub Fire Risk Service*. accessed May 31, 2021. url: `https://github.com/selabhvl/fireriskservices`.

[17] *GraphQL*. accessed May 3, 2021. url: `https://graphql.org/`.

[18] *Heroku CLI*. accessed May 22, 2021. url: `https://devcenter.heroku.com/articles/heroku-cli`.

[19] Alan Hevner et al. *Design Science in Information Systems Research*. Sept. 2013. doi: `10.2307/25148625`.

[20] *jest testing framework*. accessed May 10, 2021. url: `https://jestjs.io/docs/api`.

[21] Arjen Kraaijeveld et al. “Burning Rate and Time to Flashover in Wooden ¼ scale Compartments as a Function of Fuel Moisture Content”. In: July 2016.

[22] *Leaflet*. accessed May 13, 2021. url: `https://leafletjs.com/`.

[23] Torgrim Log. “Modeling Indoor Relative Humidity and Wood Moisture Content as a Proxy for Wooden Home Fire Risk”. In: *Sensors* 19.22 (Nov. 2019), p. 5050. doi: `10.3390/s19225050`. url: `https://doi.org/10.3390/s19225050`.

[24] *Maven*. accessed December 3, 2020. url: `https://maven.apache.org/`.

[25] *MET Frost API secret*. accessed April 22, 2021. url: https://frost.met.no/howto.htmlhttps://frost.met.no/howto.html.

[26] *Netatmo Smart Weather Station*. https://www.netatmo.com/en-eu/weather/weatherstation. Accessed: 06-05-2020.

[27] *Netatmo weather station specification*. accessed April 23, 2021. url: https://www.netatmo.com/en-eu/weather/weatherstation/specifications.

[28] *Netatmo Weathermap*. https://weathermap.netatmo.com/. Accessed: 06-05-2020.

[29] *Netatmo's weather REST API*. accessed March 17, 2021. url: https://dev.netatmo.com/apidocumentation/weather.

[30] *Norwegian Meteorological Institute 's Frost REST API*. accessed March 17, 2021. url: https://frost.met.no/index.html.

[31] *Oauth*. accessed March 17, 2021. url: https://oauth.net/.

[32] *Open Weather Pricing*. accessed April 23, 2021. url: https://openweathermap.org/price.

[33] *OpenStreetmap*. accessed May 4, 2021. url: https://www.openstreetmap.org.

[34] *React*. accessed May 10, 2021. url: https://reactjs.org/.

[35] *React Leaflet*. accessed May 4, 2021. url: https://react-leaflet.js.org/docs/api-map.

[36] *React Redux*. accessed May 13, 2021. url: https://react-redux.js.org/.

[37] *Reducing fire disaster risk through dynamic risk assessment and management (DYNAMIC)*. 2019.

[38] *Redux thunks*. accessed May 13, 2021. url: https://github.com/reduxjs/redux-thunk.

[39] *Røros Mining Town and the Circumference*. Accessed April 27, 2021. url: https://whc.unesco.org/en/list/55/.

[40] Laura Rusu et al. "Data-Driven Prediction and Visualisation of Dynamic Bushfire Risks". In: *Databases Theory and Applications*. Ed. by Muhammad Aamir Cheema, Wenjie Zhang, and Lijun Chang. Cham: Springer International Publishing, 2016, pp. 457–461. isbn: 978-3-319-46922-5.

[41] Jesús San-Miguel-Ayanz et al. “Comprehensive Monitoring of Wildfires in Europe: The European Forest Fire Information System (EFFIS)”. In: *Approaches to Managing Disaster*. Ed. by John Tiefenbacher. Rijeka: IntechOpen, 2012. Chap. 5. doi: `10.5772/28441`. url: `https://doi.org/10.5772/28441`.

[42] *Spring Initializr*. Accessed April 26, 2021. url: `https://start.spring.io/`.

[43] *Stillwater Weather*. `https://stillwaterweather.com/chandlerburningindex.php`. Accessed: 2021-24-03.

[44] Sindre Stokkenes. “Implementation and Evaluation of a Fire Risk Indication Model”. MA thesis. Department of Computing, Western Norway University of Applied Sciences, 2019.

[45] *StormGeo Research and Development*. accessed April 23, 2021. url: `https://www.stormgeo.com/company/research-and-development/`.

[46] *StormGeo REST API*. accessed April 23, 2021. url: `https://webapi.stormgeo.com/`.

[47] *StormGeo Who We Are*. accessed April 23, 2021. url: `https://www.stormgeo.com/company/who-we-are/`.

[48] *TITAN (auTomatIc daTa quAlity coNtrol)*. accessed February 25, 2021. url: `https://github.com/metno/TITAN`.

[49] Athanasios Tsipis et al. “An Alertness-Adjustable Cloud/Fog IoT Solution for Timely Environmental Monitoring Based on Wildfire Risk Forecasting”. In: *Energies* 13.14 (2020). issn: 1996-1073. doi: `10.3390/en13143693`. url: `https://www.mdpi.com/1996-1073/13/14/3693`.

[50] *TypeScript Programming Language*. accessed May 11, 2021. url: `https://www.typescriptlang.org/`.

[51] Michelle Ward et al. “Impact of 2019–2020 mega-fires on Australian fauna habitat”. In: *Nature Ecology & Evolution* 4 (Oct. 2020), pp. 1–6. doi: `10.1038/s41559-020-1251-1`.

[52] Roel Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Jan. 2014, pp. 1–332. isbn: 978-3-662-43838-1. doi: `10.1007/978-3-662-43839-8`.

[53] *Yarn*. accessed May 13, 2021. url: `https://yarnpkg.com/`.

[54] *Your packages, at home with their code*. accessed December 3, 2020. url: https://github.com/features/packages.

Appendices

Appendix A

Locations used for Experiments

Place	**Coordinates**
Oslo	59.9139° N, 10.7522° E
Stavanger	58.9700° N, 5.7331° E
Haugesund	59.4136° N, 5.2680° E
Bryne	58.7358° N, 5.6477° E
Kopervik	59.2837° N, 5.3069° E
Ålesund	62.4722° N, 6.1495° E
Molde	62.7372° N, 7.1607° E
Kristiansund	63.1103° N, 7.7281° E
Ørsta	62.1981° N, 6.1276° E
Volda	62.1453° N, 6.0748° E
Bodø	67.2829° N, 14.4151° E
Mo (Mo I Rana)	66.3137° N, 14.1420° E
Narvik	68.4385° N, 17.4273° E
Mosjøen	65.8369° N, 13.1934° E
Fauske	67.2592° N, 15.3916° E
Drammen	59.7441° N, 10.2045° E
Fredrikstad	59.2205° N, 10.9347° E
Moss	59.4260° N, 10.6985° E
Halden	59.1330° N, 11.3875° E
Kongsberg	59.6689° N, 9.6502° E
Hamar	60.7945° N, 11.0680° E
Lillehammer	61.1153° N, 10.4662° E
Gjøvik	60.7954° N, 10.6916° E
Elverum	60.8821° N, 11.5625° E
Porsgrunn	59.1386° N, 9.6555° E
Tønsberg	59.2676° N, 10.4076° E
Sandefjord	59.1313° N, 10.2166° E
Larvik	59.0538° N, 10.0295° E
Kristiansand	58.1599° N, 8.0182° E
Arendal	58.4618° N, 8.7724° E
Korsvik	58.1447° N, 8.0744° E
Grimstad	58.3447° N, 8.5949° E
Bergen	60.3913° N, 5.3221° E
Askøy	60.4619° N, 5.0893° E
Leirvik	59.7798° N, 5.5005° E
Osøyro	60.1839° N, 5.4638° E
Knarrevik	60.3738° N, 5.1578° E
Trondheim	63.4305° N, 10.3951° E
Stjørdalshalsen	63.4712° N, 10.9189° E
Steinkjer	64.0150° N, 11.4953° E
Levanger	63.7464° N, 11.3005° E
Alta	69.9689° N, 23.2716° E
Hammerfest	70.6634° N, 23.6820° E
Vadsø	70.1583° N, 29.8117° E
Kirkenes	69.7269° N, 30.0450° E
Bjørnevatn	69.6675° N, 29.9872° E
Haugesund Sentrum	59.4138° N, 5.2679° E
Haugesund Hasseløy	59.4171° N, 5.2535° E
Skudeneshamn	59.1470° N, 5.2656° E
Stavanger Gamlebyen	58.9722° N, 5.7259° E
Røros	62.5867° N, 11.3870° E
Henningsvær	68.1545° N, 14.2070° E
Hemnesberg	66.2246° N, 13.6243° E
Lærdalsøyri	61.0992° N, 7.4816° E
Bryggen I Bergen	60.3977° N, 5.3246° E
Bleik	69.2719° N, 15.9559° E
Arendal Kolbjørnsvik	58.4502° N, 8.7614° E
Arendal Tyholmen	58.4583° N, 8.7654° E
Askøy Gamle Strusshamn	60.4053° N, 5.1922° E
Aurland Låvi	60.8770° N, 7.2521° E
Aurland Otternes	60.8756° N, 7.1489° E
Aurland Undredal	60.9497° N, 7.1030° E
Bamble Herre	59.1044° N, 9.5579° E
Bamble Langesund	59.0005° N, 9.7447° E
Bamble Stathelle	59.0452° N, 9.6982° E
Bergen Laksevåg	60.3810° N, 5.2593° E
Bergen Marken	60.3906° N, 5.3322° E
Bergen Nordnes	60.3988° N, 5.3076° E
Bergen Nøstet	60.3938° N, 5.3148° E
Bergen Salhus	60.5048° N, 5.2663° E
Bergen Sandviken	60.4120° N, 5.3253° E
Bergen Skuteviken	60.4027° N, 5.3215° E
Bergen Sydnes	60.3894° N, 5.3172° E
Bergen Vågsbunnen	60.3940° N, 5.3282° E

Table A.1: List of locations.

Appendix B

URL to the Source Code of implementations

URLs for each implemented application is presented below.

- **Fire Risk Model**: `https://github.com/selabhvl/fireriskmodel`
- **Fire Risk Services**: `https://github.com/selabhvl/fireriskservices`
- **Data Harvesting Service**: `https://github.com/selabhvl/data-harvester`
- **Middleware**: `https://github.com/selabhvl/middleware`
- **Front-End**: `https://github.com/Evjenth/fireriskfrontend`

URLs to the files containing the data used for experiments are found here:

- `https://drive.google.com/file/d/1GOR_B3qbaF2eY-ppSfEtlh-sLSBkc2Ql/view?usp=sharing`