
# A Mobile Application for Fire Risk Notification based on Edge Computing

## System documentation

*Authors:*

Emilie Hinna Fisketjøn, Abu Tallaha Hussain, Thorbjørn Svendal

*Supervisors:*

Prof. Lars Michael Kristensen, Ph.D. candidate Ruben Dobler Strand

Department of Computer science, Electrical engineering, and Mathematical sciences

Western Norway University of Applied Sciences

May 23, 2022

# REVISION HISTORY

| Date | Version | Description | Author |
|---|---|---|---|
| 26.04.2022 | 1.0 | First iteration | Emilie, Tallah and Thorbjørn |
| 12.05.2022 | 2.0 | Second iteration | Emilie, Tallah and Thorbjørn |
| 21.05.2022 | 3.0 | Final iteration | Emilie, Tallah and Thorbjørn |

# Table of Contents

# Introduction

This document is supposed to give insight regarding the architecture and explain in-depth. It is supposed to function as aid for developers and others who wish to continue the work. This document contains a description of the project structure, architecture and database. Furthermore, a detailed step by step toolchain guide will be provided for setting up the project and so on.

Link to the private GitHub repository for the mobile application:

https://github.com/selabhvl/dynamic-mobileapp

Link to the private GitHub repository for the fire risk model:

https://github.com/selabhvl/fireriskmodel-csharp

Supervisor Lars Michael Kristensen can provide access to the repositories as required.

# Architecture

The application's core components (Figure 1) are represented in purple and follows the same naming convention as the MVVM pattern. The Views and View Models are tightly coupled, whereas Views will trigger commands in the View Models. The View Models component can access the local database by storing and modifying the content. The View Models uses the Services component, which sends API requests to the Weather Data Source component to retrieve weather data. The Services use the Models to format the weather data response to objects matching the .NET standards. After formatting the responses, it passes the data into the Fire Risk Model (FRM) and finally receives the fire risk calculations.

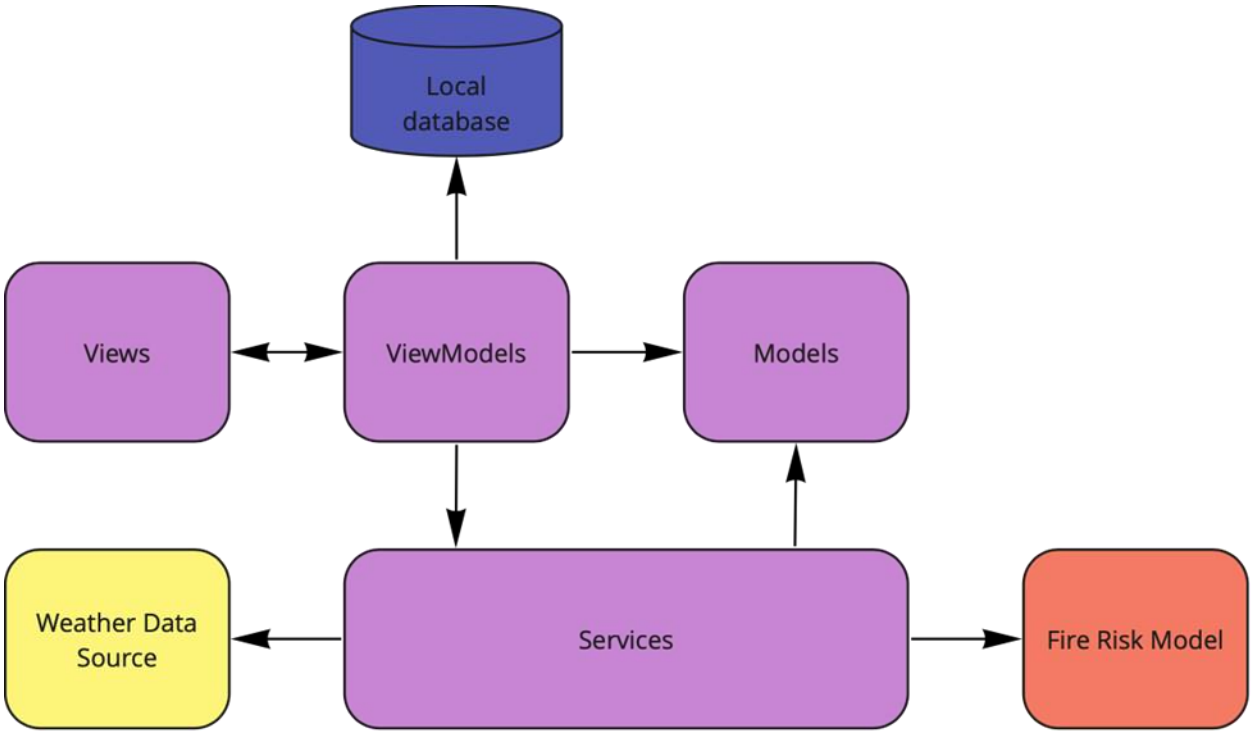


*Figure 1. High level software application architecture*

# Project Structure

Figure 2 illustrates the mobile application's project structure. The main project `FireGuard` consists of `Data`, `Models`, `Services`, `ViewModel`, and `View` folders. The `FireGuardTest` project uses the NUnit library to be able to perform Unit testing for all methods. The `Model` is the project consisting of the `Fire Risk Model (FRM)` and `FireGuard` has a dependency on this project, making it possible to access classes, interfaces, and methods.

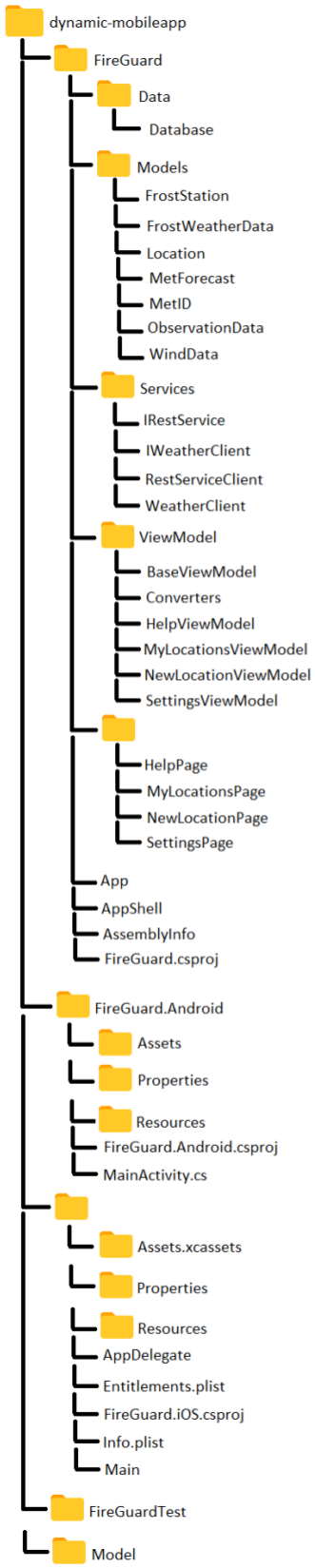


*Figure 2. Overview of the project structure*

If `FireGuard` does not have a dependency on the `Model` project:

1. Right-click and Add references (Figure 3)
2. Check the Model project (Figure 4) and Select

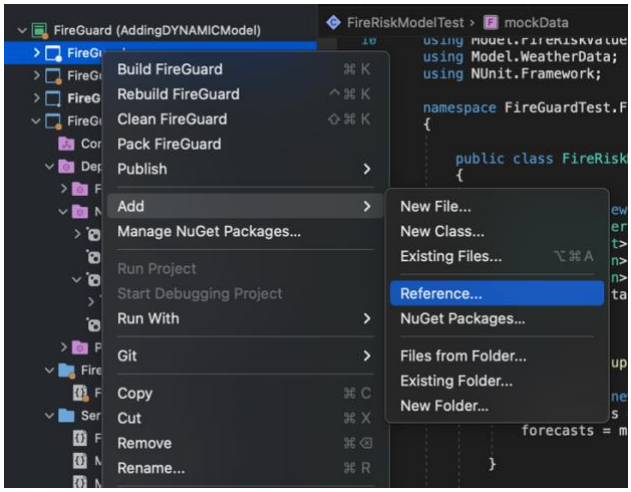


*Figure 3. How to add dependency (1/2)*

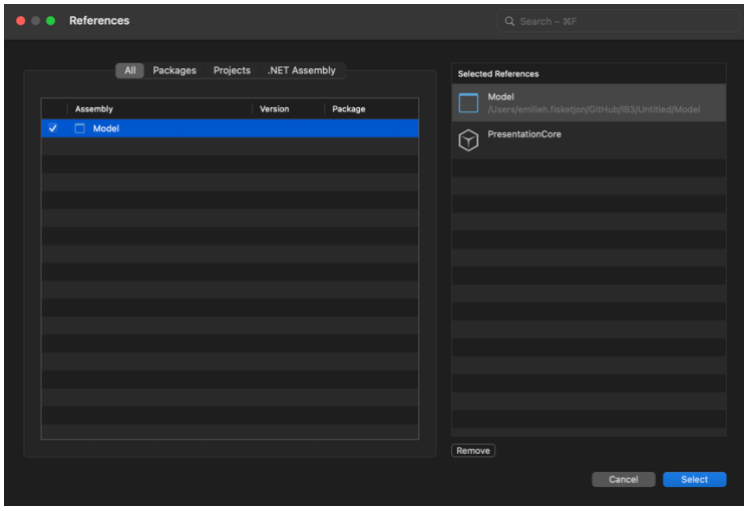


*Figure 4. How to add dependency (2/2)*

# Class Diagram

## 4.1 High-level class diagram

A high-level class diagram illustrating the different components in `FireGuard`, seen in Figure 5.

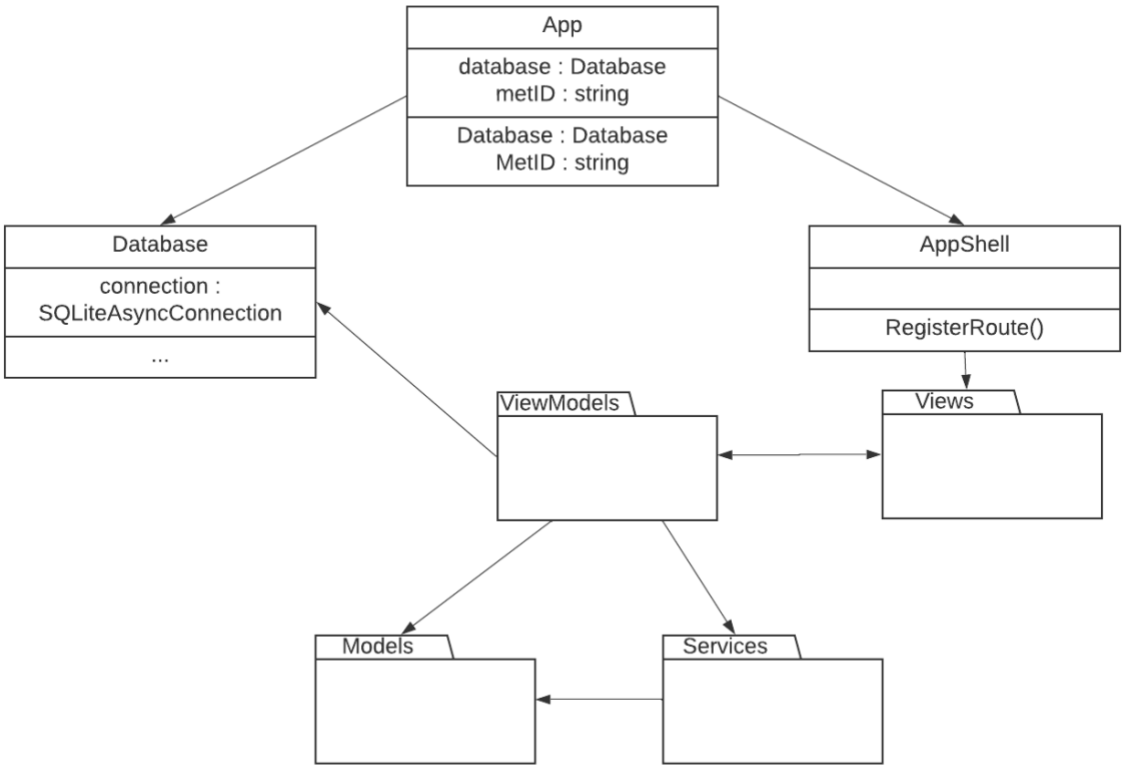


*Figure 5. High-level class diagram*

### 4.1.1 Services

The application requires an API layer for retrieving all weather data. The following chapter introduces the `Services` components that implement the protocols defined by the Business Layer and handle all HTTP requests to MET and Frost API.

`Services` consist of two interfaces, `IRestServiceClient` and `IWeatherClient` and classes `RestServiceClient` and `WeatherClient` seen in Figure 6, which implements these interfaces. The `View Models` classes heavily depends on `Services`.

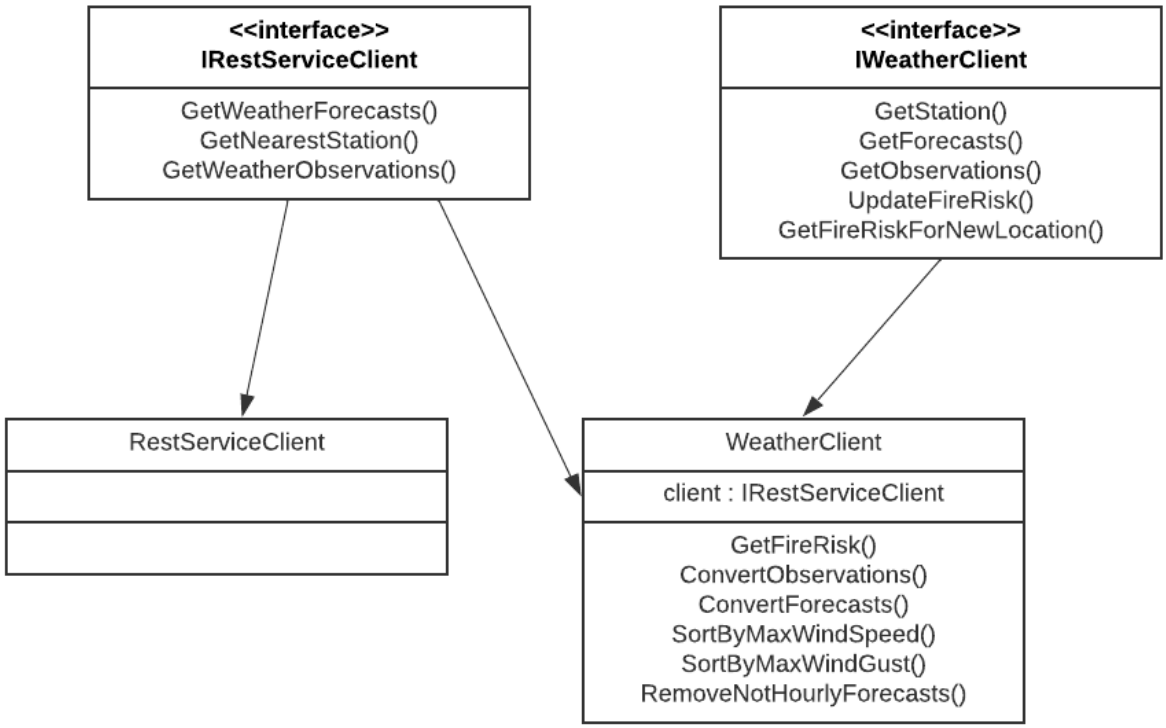


*Figure 6. Representation of the services components class diagram*

### 4.1.2 Models

The `Models` components act as non-visual classes that encapsulate the data and represent the app's domain, including business and validation logic. This chapter shows the different model objects. The `FrostStation`, `FrostWeatherData`, and `MetForecast` are auto generated classes by quicktype from JSON to the respective .NET classes. The `WindData` acts as a placeholder class when all wind data is retrieved from the `MetForecast` object. `Location` and `MetID` acts also as tables in the local database.

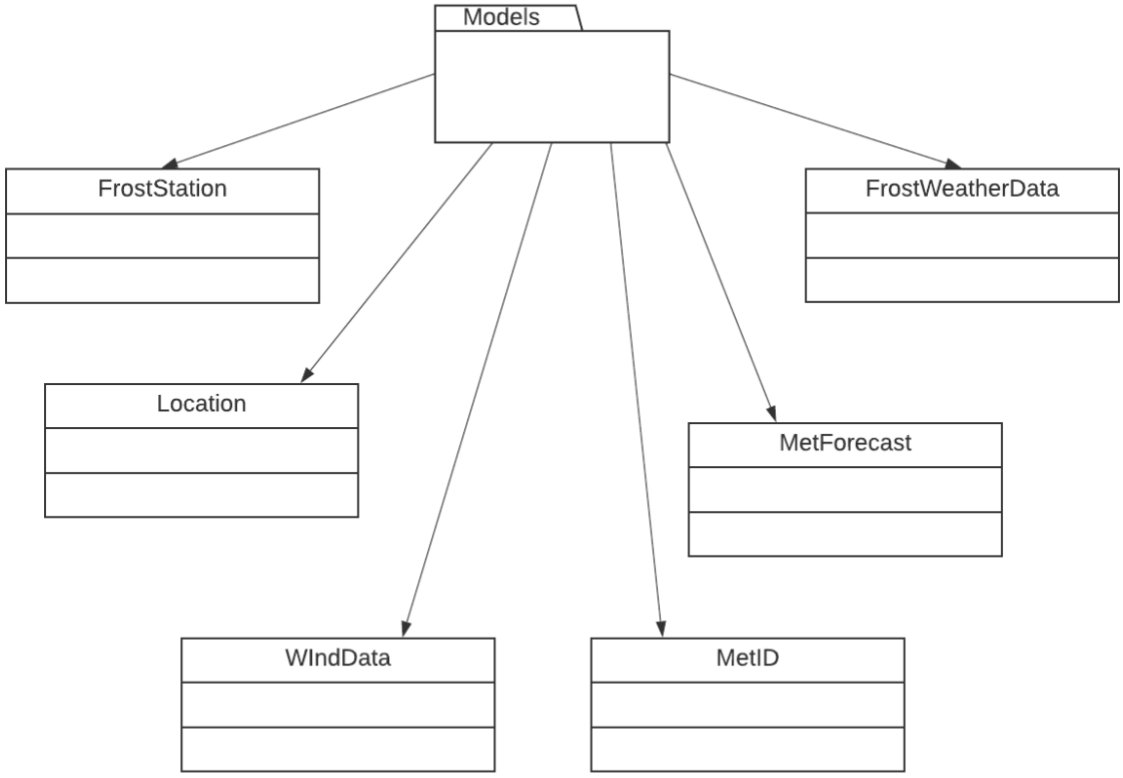


*Figure 7. Class diagram of the Models component*

### 4.1.3 Views and ViewModels

Figure 8 illustrates the relations between the `View` and `ViewModel`. The classes ending with Page represent a page in the app. Often, each `View` is tightly coupled to a `ViewModel` with data bindings, and these relations can be seen in the figure. `AppShell` is responsible of Initializing all the different pages, and each page will then initialize the `ViewModel`. The `BaseViewModel` is the base class that all the `ViewModel` classes implements.

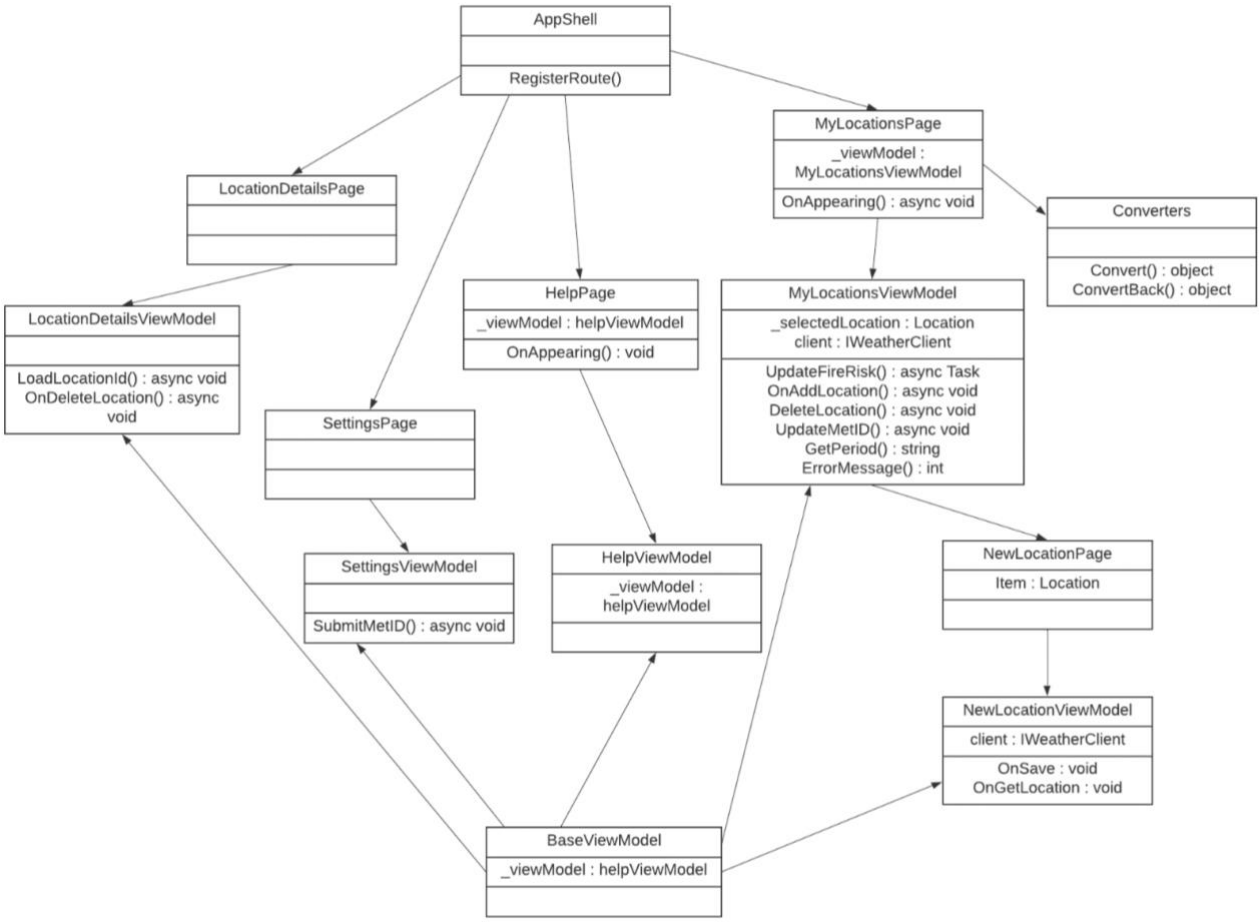


*Figure 8. Class diagram of the Views and ViewModels*

# Database Model

Figure 9 shows a representation of the two tables that is being stored in the local database. Figures 10 and 11 shows `Location` and `MetID` class, and how they are utilized to be stored in the database.

| location | |
|---|---|
| id | Guid [PK] |
| name | Varchar |
| longitude | Varchar |
| latitude | Varchar |
| weather station id | Varchar |
| wind speed | List<Double> |
| wind gust | List<Double> |
| wind direction | List<Double> |
| fire risk | List<Int> |
| peak risk | Int |

| met id | |
|---|---|
| id | Int [PK] |
| client_id | Varchar |

*Figure 9. Database tables; Location and Met ID*

```
1       public class MetID
2       {
3           [PrimaryKey]
4           public int id { get; set; }
5           public string client_id { get; set; }
6       }
```

*Figure 10. MET ID object*

```
1    public class Location
2    {
3        [PrimaryKey, AutoIncrement]
4        public Guid Id { get; set; }
5        public string Name { get; set; }
6        public string Latitude { get; set; }
7        public string Longitude { get; set; }
8        public string WeatherStationID { get; set; }
9        [TextBlob("WindSpeedsBlobbed")]
10       public List<double> WindSpeed { get; set; }
11       public string WindSpeedsBlobbed { get; set; }
12       [TextBlob("WindGustSpeedsBlobbed")]
13       public List<double> WindGustSpeed { get; set; }
14       public string WindGustSpeedsBlobbed { get; set; }
15       [TextBlob("WindDirectionsBlobbed")]
16       public List<double> WindDirection { get; set; }
17       public string WindDirectionsBlobbed { get; set; }
18       [TextBlob("FireRisksBlobbed")]
19       public List<int> FireRisk { get; set; }
20       public string FireRisksBlobbed { get; set; }
21       public int PeakRisk { get; set; }
22   }
```

*Figure 11. Location object*

# Server Services

## 6.1 The Norwegian Meteorological Institute

MET is Norway's national meteorological institute [1] and provides weather forecasts for civilians and conducts meteorology, oceanography, and climatology research. In collaboration with the Norwegian Broadcasting Corporation (NRK), they host the site Yr.no [2], an online weather service that offers detailed weather forecasting. In addition to Yr.no, they provide the service MET Norway Weather API [3]. The Weather API is an interface to a selection of data produced and made available by MET through the URL `https://api.met.no`.

The application will use the Location forecast service [4] for a specified area and Frost [5], a REST API for meteorological observation data.

### 6.1.1 The Weather API Location Forecast Service

The Weather API Location forecast service gives a weather forecast for the next nine days for the specified geographic position based on coordinates. The forecast is based on numerous data sources and is updated several times a day. The forecast provides hourly updates for the first two-three days, after the third day the update frequency is changed to an interval of six hours [6].

The data comes in three endpoints: *classic*, *complete*, and *compact*. *classic* is the old XML format and is now considered legacy, MET will not add any new parameters to this version. *complete* and *compact* are JSON objects, where *complete* consists of all values and *compact* is a shorter version consisting of the most used parameters. The endpoint *complete* included all the required data and was therefore used [4].

An example of an HTTP GET request to the *locationforecast complete* endpoint:

```
https://api.met.no/weatherapi/locationforecast/2.0/complete.json?
lat=60.3691&lon=5.3505
```

The *lat* and *lon* parameters represent the geocoordinate's latitude and longitude. It is not recommended to use more than four decimals by MET, the reason for this is to avoid blocking which will provide effective caching. The parameters that are typically included in an HTTP GET request are the following: altitude, latitude, and longitude. Longitude and latitude are the only parameters that must be included for every request whereas altitude is optional.

### 6.1.2 The Frost API

Frost [5] is a RESTful API that provides free access to MET Norway's historical weather and climate data archive. The data includes quality-controlled temperature, precipitation, and wind data measurements. It also offers information such as metadata about the weather stations. The API is primarily used by developers who wish to access MET's historical data archive.

The model of Log [7] uses historical weather data to calibrate, this is done so that model can achieve and provide more accurate predictions. To achieve the most accurate and relevant results, the application must retrieve data from the closest weather station that stores the relevant data. Frost has various API reference endpoints [8] which deliver different types of data. For this project, *sources* and *observations* were used.

The *sources* reference [8] provides the application with the closest station. There are several parameters to define when making a request. Relevant parameters for this application are *types*, *elements*, and *geometry*.

An example of an HTTP GET request to the *sources* endpoint:

```
https://frost.met.no/sources/v0.jsonld?types=SensorSystem&elements=air_temperature,relative_humidity,wind_speed&geometry=nearest(POINT(5.3327 60.383))
```

The *types* parameter specifies the station type, whereas *SensorSystems* is a station with measuring sensors. It is also possible to choose between *InterpolatedDataset* and *RegionDataset*. The API can

exclude stations that do not meet the requirements by specifying the elements parameter with: *air_temperature, relative_humidity,wind_speed.*

As for the geometry parameter, it specifies the geometry of a station. The geographical location is expressed in terms of either a single point or a polygon area [9]. The syntax `nearest(POINT (lon lat))` refers to the item closest to these coordinates. If the `nearest` function is used, the response will include the distance in kilometres from the reference point.

The *observations* reference [8] provides actual observations data from MET Norway's data storage system. It only handles restricted data by using query parameters. The required parameters to make a request are *sources, referencetime,* and *elements*.

An example of an HTTP GET request to the *observations* endpoint:

```
https://frost.met.no/observations/v0.jsonld?sources=SN50540&referencetime=2022-04-01/2022-04-02&elements=air_temperature,relative_humidity,wind_speed
```

*Sources* specify which station to get observations from, and *referencetime* implies for which time range. The time specification [9] is based on UTC and ISO-8601. Frost allows the distance between the starting point of consecutive intervals to be specified explicitly.

The format the application uses to cite dates is YYYY-MM-DD. For instance, one can define a time range by writing: *2022-04-01/2022-04-02* .

It will retrieve observations that are expressed according to the ISO-8601 format [10]. Starting from *2022-04-01T00:00:00:000Z* to *2022-04-02T23:00:00:000Z.*

# Security

To gain access to the service, the user needs to create an account [11]. This can easily be done by visiting `https://frost.met.no/auth/requestCredentials.html`. The user will provide an email address to register and will be provided with a new *client ID* and *client secret*. The email will be stored in a database managed by MET [12]. The *client ID* is required for free service and fast and stable performance [13]. The *client secret* is only necessary for access to data that is not available to the public.

It is optional to use Basic authentication [14] or OAuth2 [15]. When accessing confidential data, the OAuth2 must be used, but will also work for available data. The requests relevant for this application only need the *client id*, and will hereby be referred to as MET ID.

The authorization scheme for the application depends on Basic Authentication, seeing that requesting the *client's secret* was unnecessary. If this shifts, the application can easily be modified to meet those changes. The Basic Authentication scheme can be considered secure only when the web client and server connection are secure [14]. Thus, it is safe to assume that these rules apply for this service since MET encourages its users to utilize basic authentication. The credential is converted to base-64 encoding before passing it into an authentication header and the format of the header is Authorization: Basic <client ID>:<client secret>.

The application will store the user's MET ID in the local database in cleartext. When creating a MET account and receiving the MET id, the service only requires an email from the user. This implies that the client id is not seen as sensitive information, seeing that there is no need for a user password. If someone gets access to another user's email, the MET ID will be easily accessible. Therefore, the application will not secure the MET ID any further. We believe that this responsibility lies with MET.

# Installation and Execution

There are some distinctions between Visual Studio (windows) and Visual Studio for mac (iOS). The following chapters will give a further explanation of the main differences and a detailed guide for both operating systems.

## 8.1 Get started

### 8.1.1 Windows

Visual studio 2022

https://visualstudio.microsoft.com/vs/

1. Download Visual Studio and open
2. Press Modify (figure 12)

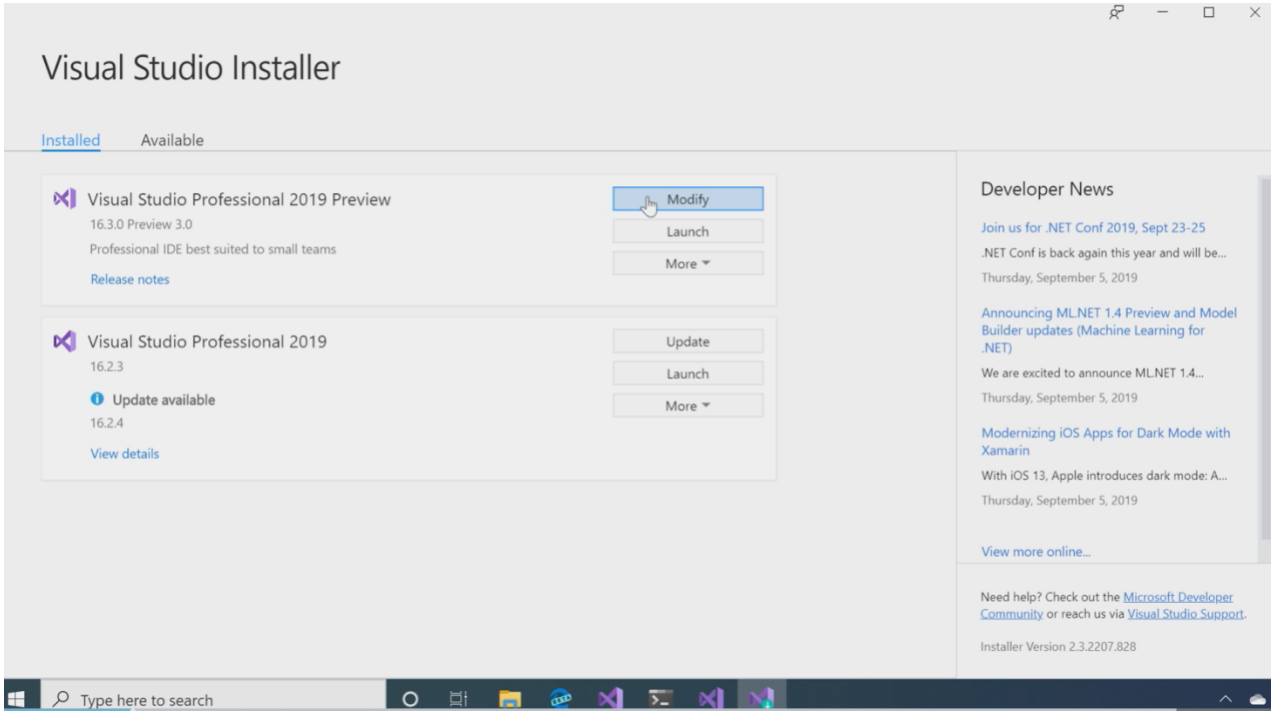


*Figure 12. How to download Visual Studio (1/2)*

3. Choose the Workloads tab folder and scroll down to Mobile & Gaming.
4. Check the Mobile development with .NET (Xamarin) (figure 13)

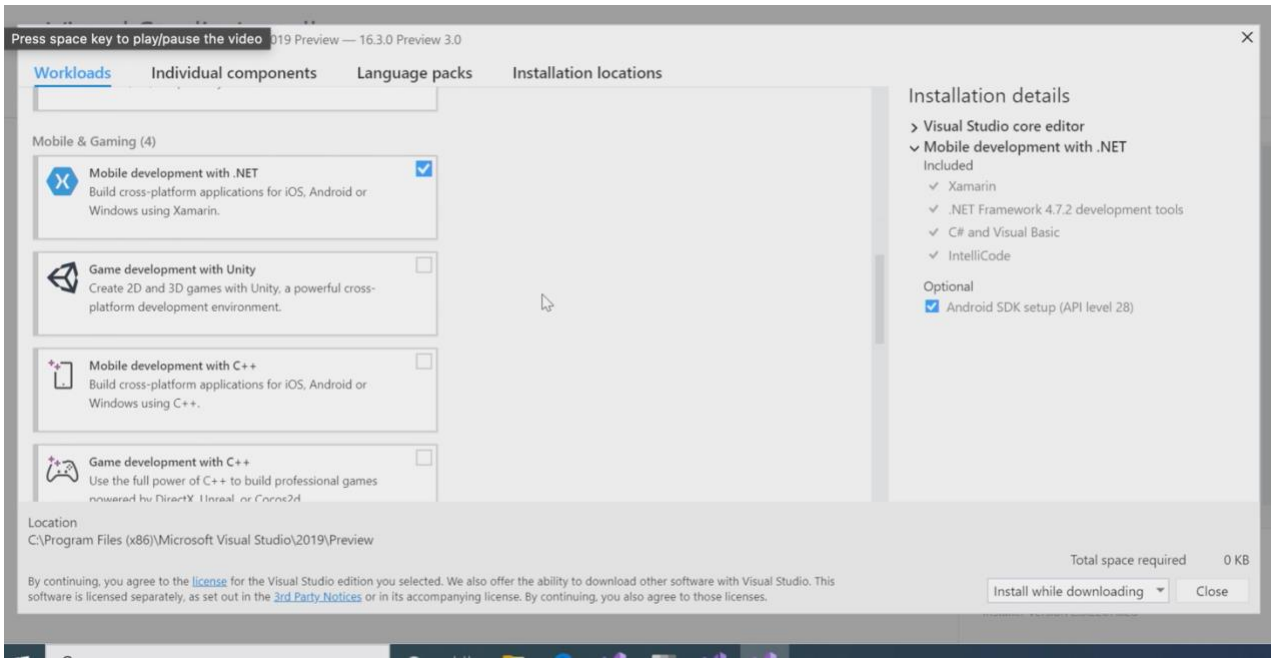


*Figure 13. How to download Visual Studio (2/2)*

### 8.1.2 Mac iOS

Choose between Visual studio for mac 2019

https://visualstudio.microsoft.com/vs/mac/

or 2022 Preview

https://visualstudio.microsoft.com/vs/mac/preview/

Download and open the Visual studio for mac file and check off .NET core, Android, and iOS. (Figure 14)

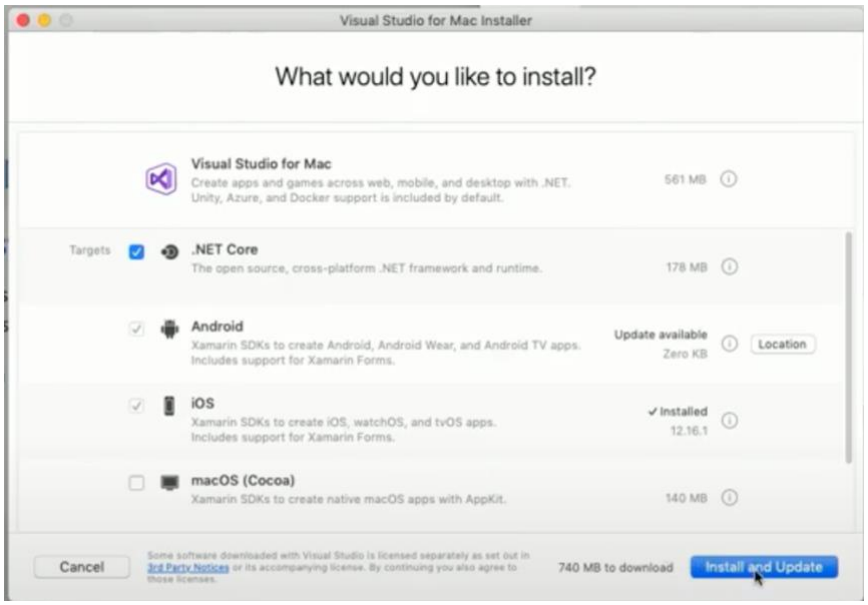


*Figure 14. How to download Visual Studio for mac*

Then, open App Store and download Xcode:
https://apps.apple.com/no/app/xcode/id497799835?mt=12

We also recommend following this tutorial for setting up Xamarin and getting started with developing:
https://www.youtube.com/watch?v=JH8ekYJrFHs&list=PLdo4fOcmZ0oU10SXt2W58pu2L0v2dOW-1&index=1

## 8.2 Emulator

To be able to run Xamarin apps it's required to install an emulator. Running an iPhone (iOS) emulator on a Windows machine is not supported. For Mac iOS, one can choose between Android and iPhone. The following chapter will give a brief tutorial on how to install the different emulators.

### 8.2.1 Windows

1. After installing Visual Studio, open a project
2. Tools -> Android -> Android SDK Manager… (Figure 15)

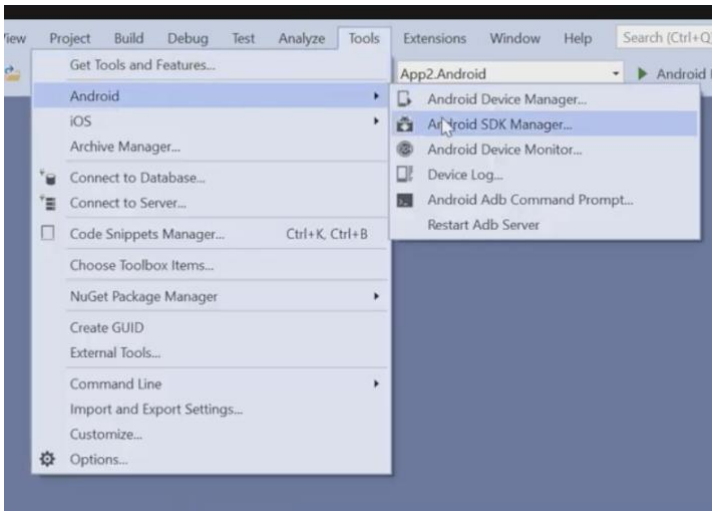


*Figure 15. How to download Android Emulator for Windows (1/6)*

3. Choose between many different Android SDK's. It is not required to have all installed. (Figure 16)

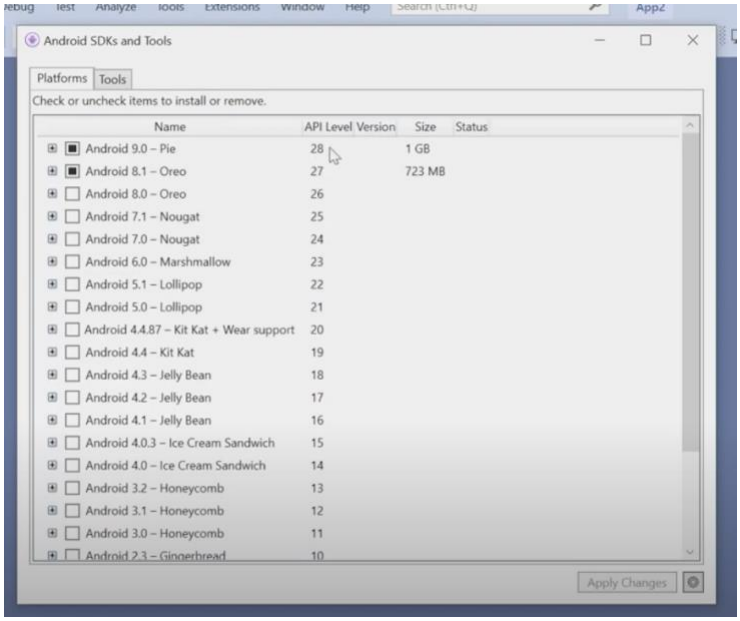


*Figure 16. How to download Android Emulator for Windows (2/6)*

4. Go to Tools and install the Android Emulator (Figure 17). Android Emulator is the tool set that makes it possible to run the app on the computer

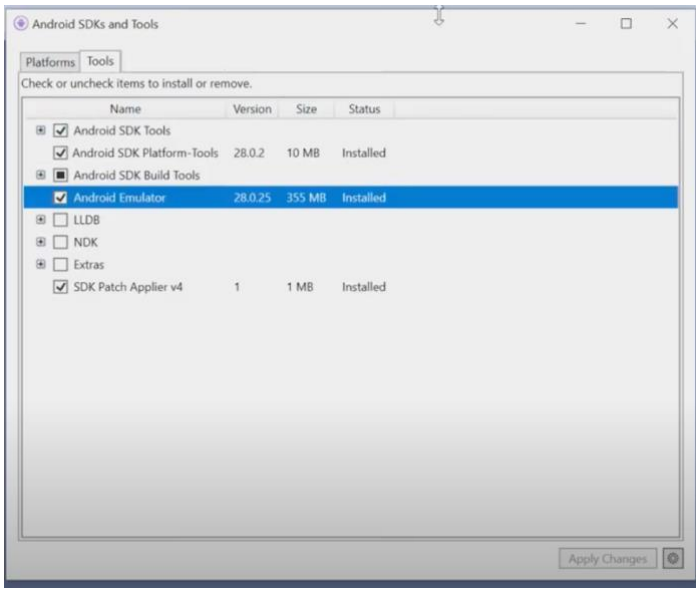


*Figure 17. How to download Android Emulator for Windows (3/6)*

5. Go back to the main menu -> Tools -> Android -> Android Device Manager (figure 18)

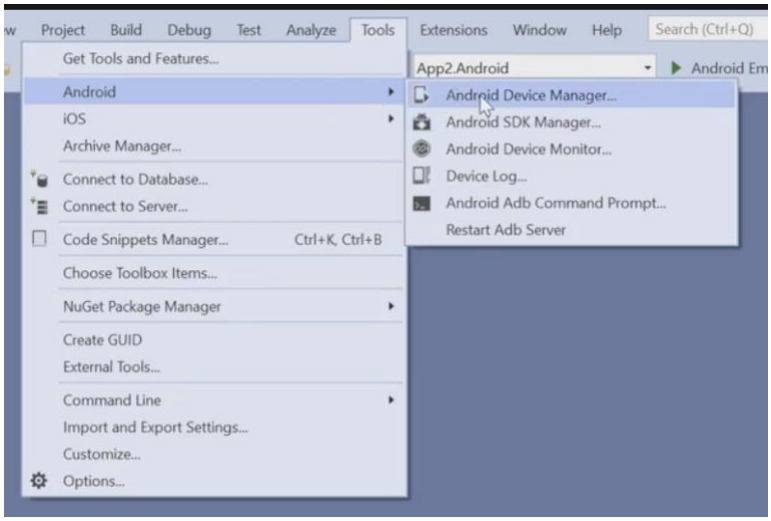


*Figure 18. How to download Android Emulator for Windows (4/6)*

6. Press +New and set up an Android simulator (Figure 19). Press Create, and the simulator will start downloading (Figure 20).

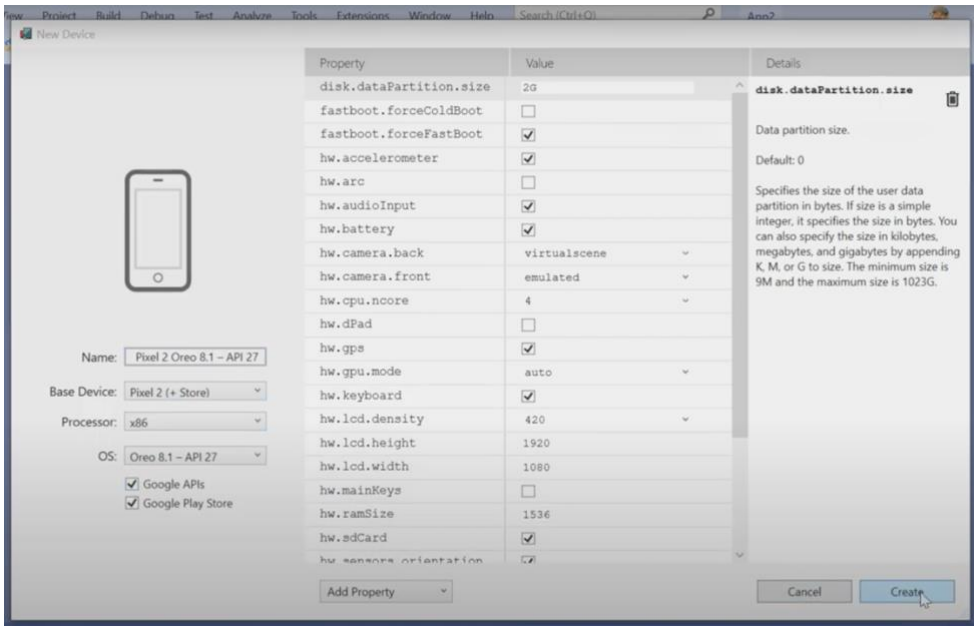


*Figure 19. How to download Android Emulator for Windows (5/6)*

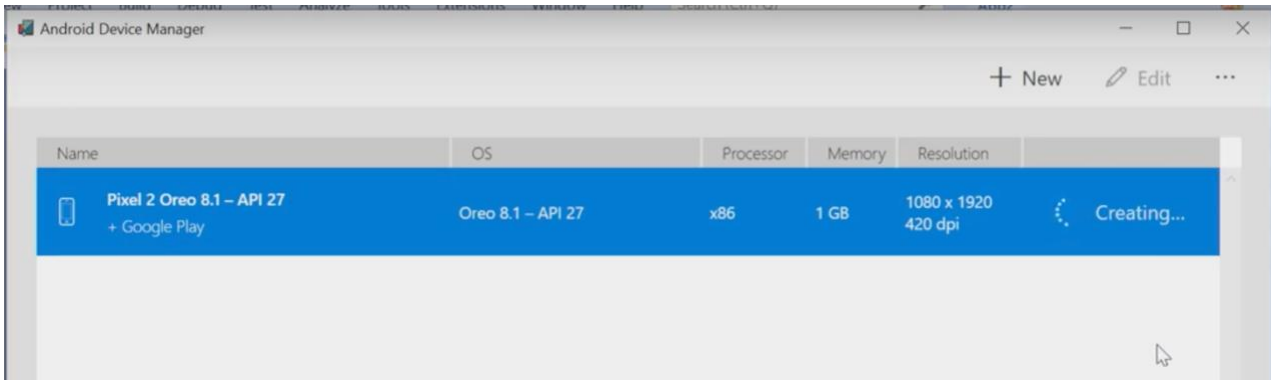


*Figure 20. How to download Android Emulator for Windows (6/6)*

### 8.2.2 Mac iOS

To set up an emulator for Visual Studio for Mac, it is required to have Xcode installed.

1. Open Xcode -> Create new project -> Document App (Figure 21) -> (Figure 22)

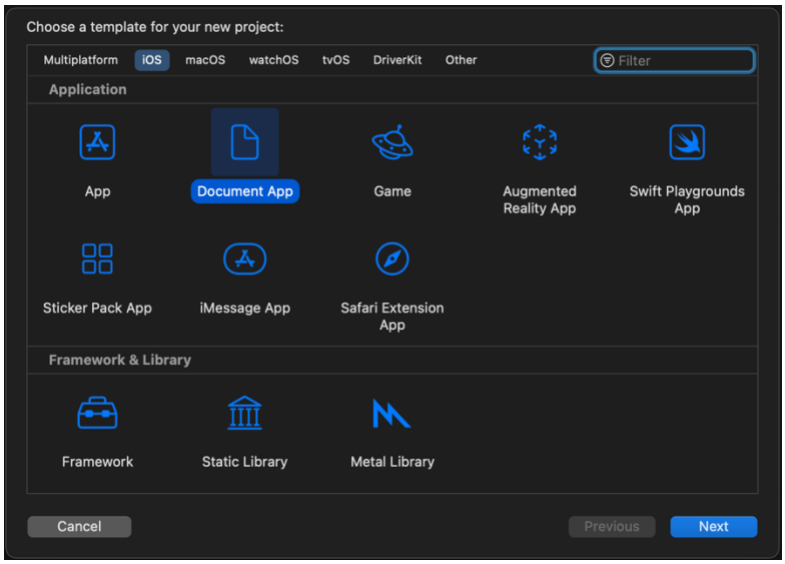


*Figure 21. How to set up iOS simulator for Mac (1/4)*

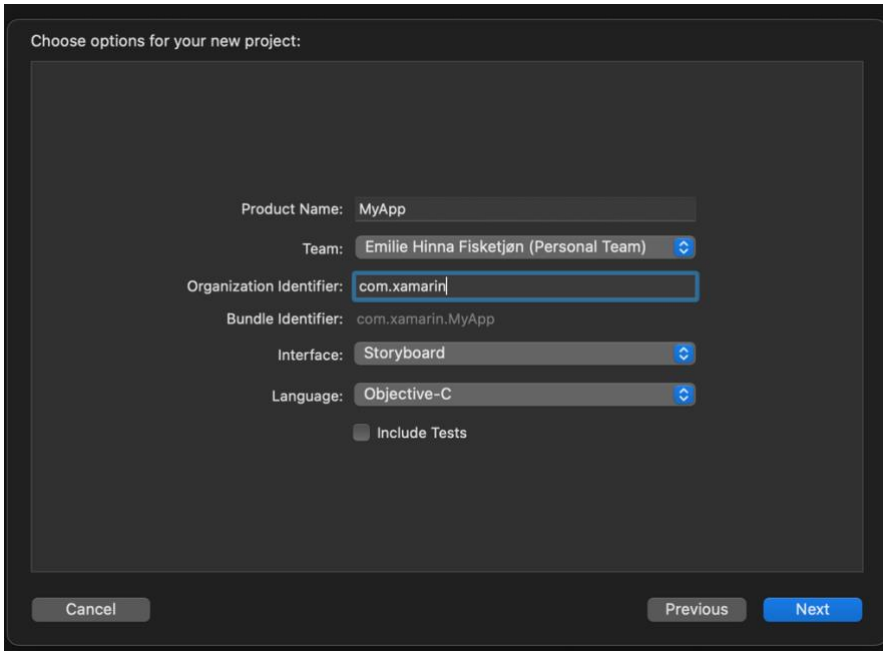


*Figure 22. How to set up iOS simulator for Mac (2/4)*

2. When Xcode is open -> Window -> Devices and Simulators On the left hence side, there should be various simulator options available, if not press the + button and create one (Figure 23)

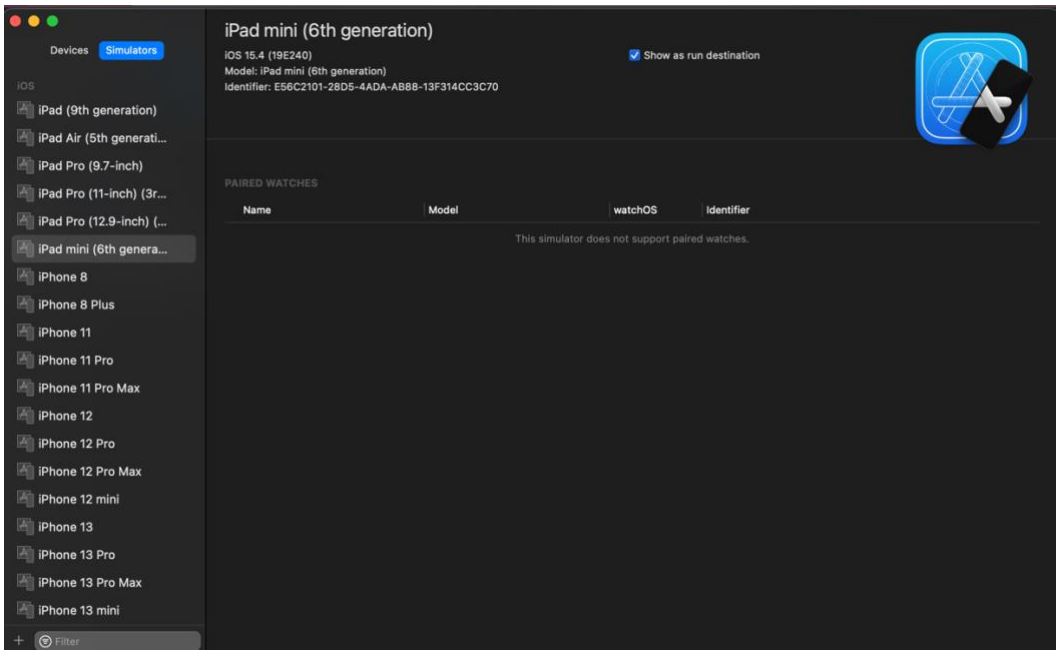


*Figure 23. How to set up iOS simulator for Mac (3/4)*

3. Restart Visual Studio for Mac, and all the various simulators should be available in the Debug menu (Figure 24)


*Figure 24. How to set up iOS simulator for Mac (4/4)*

## 8.3 NuGet packages

Both operating systems can follow the same step-by-step guide for downloading all NuGet packages.

### 8.3.1 FireGuard

**Frameworks**

- NETStandard.Library – 2.1.0

The project needed to be set to a lower framework version when coupling FireGuard and Models in the same project solution.

- Newtonsoft.Json
- RestSharp
- Sqlite-net-pcl
- SQLiteNetExtensions
- SqLteNetExtensions.Async
- Xamarin.CommunityToolkit
- Xamarin.Essentials
- Xamarin.Forms

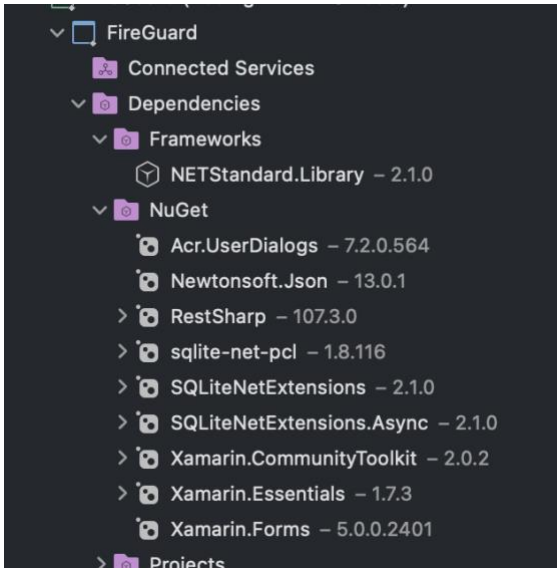


*Figure 25. NuGet packages in FireGuard*

## Installation Guide (example for RestSharp):

1. Right click on the project Solution **FireGuard**
2. Click on **Manage NuGet Packages…**
3. **Browse – Search** – "RestSharp" – Add Package (Figure 26)
4. Repeat for all NuGet packages

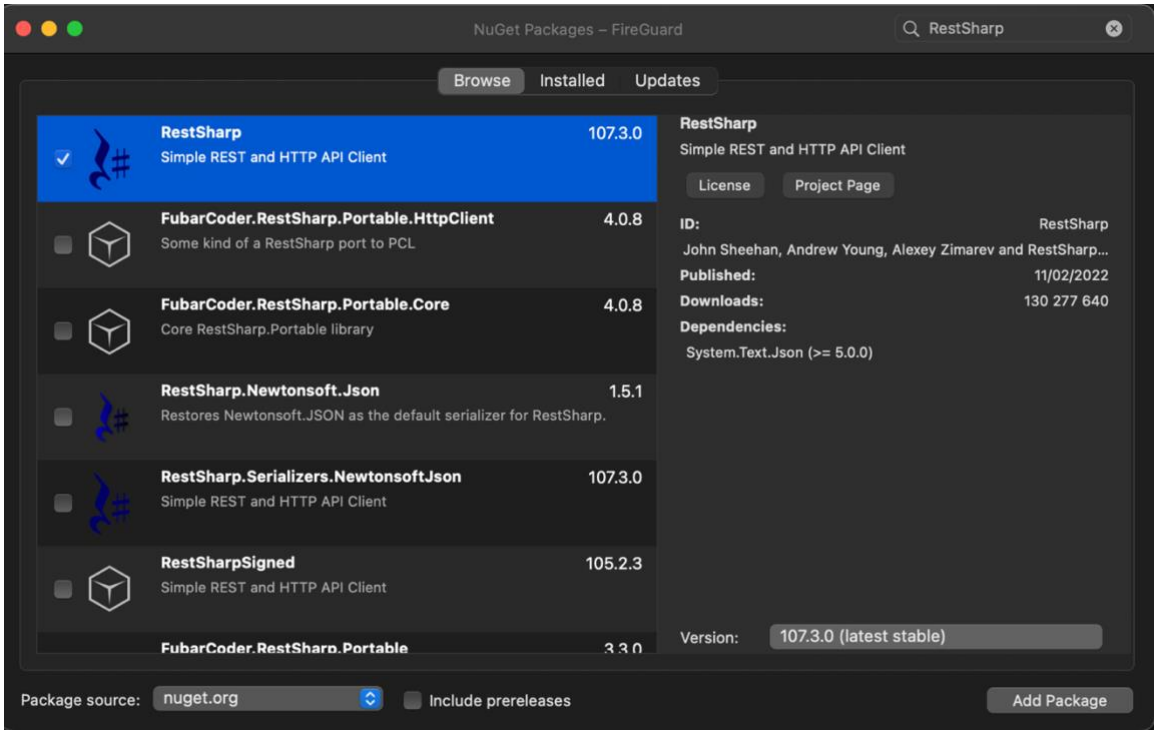


*Figure 26. How to download NuGet packages*

### 8.3.2 Model (FRM)

#### Frameworks

- NETStandard.Library – 2.1.0

#### NuGet Packages

- Newtonsoft.Json

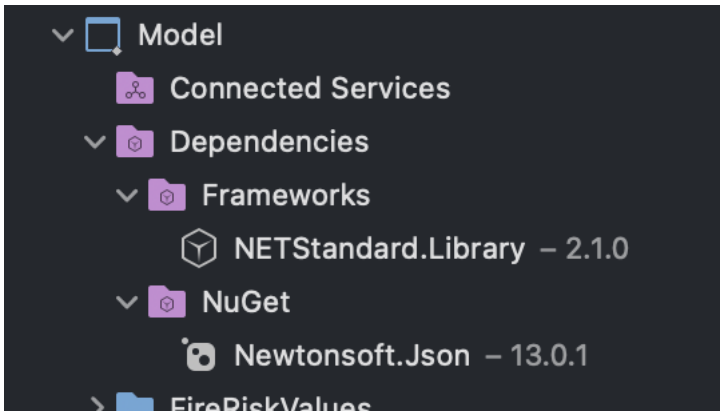


*Figure 27. NuGet packages in Model (FRM)*

# Documentation and Source Code

The following chapter will explain how to autogenerate all documentations to a single XML file from Visual studio [16].

## 9.1 Documentation: Windows

1. On the solution explorer window, right-click the project and click **Properties**.
2. Go to **Build** tab and **Output**.
3. Check of the "XML documentation file:" and choose the autogenerated files path (Figure 28).
4. Run the solution and view the documentation.

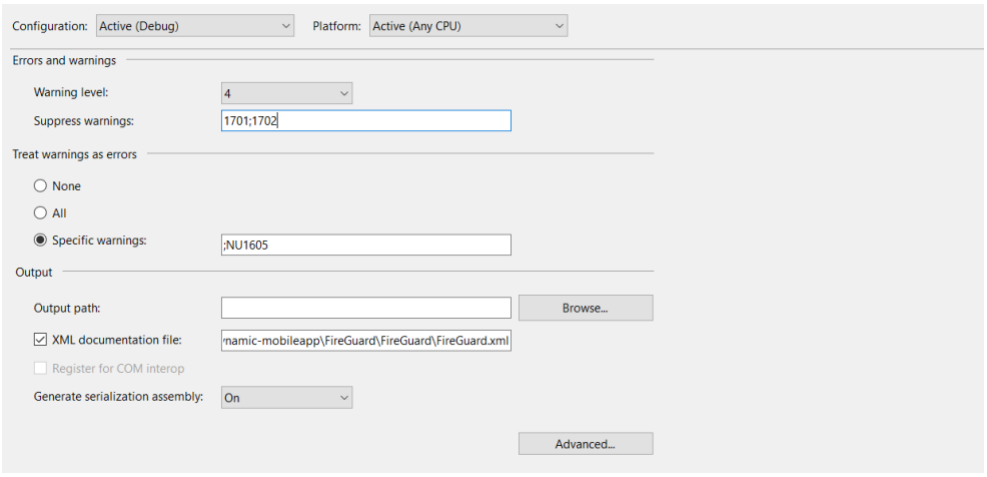


*Figure 28. How to generate the documentation XML file on Windows*

## 9.2 Documentation: iOS

1. Right click on the project and choose Options (Figure 29)

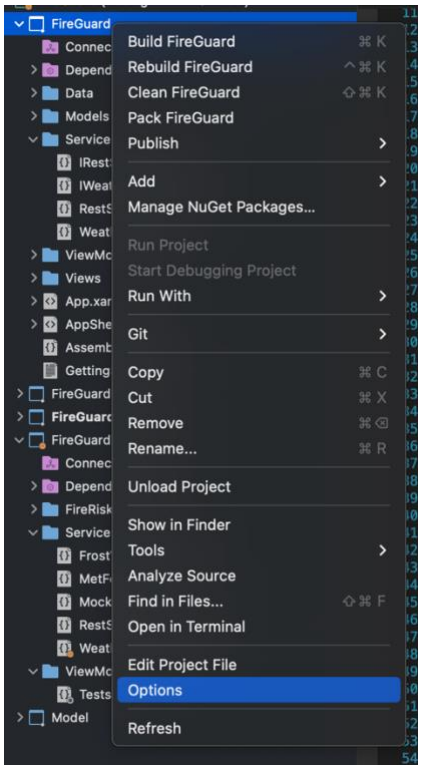


*Figure 29. How to generate the documentation XML file on Mac (1/3)*

2. Go to the **Build** drop down menu and **Compiler** (Figure 30).
3. Go down to **Errors and Warnings** and check of the "Generate XML documentation:"

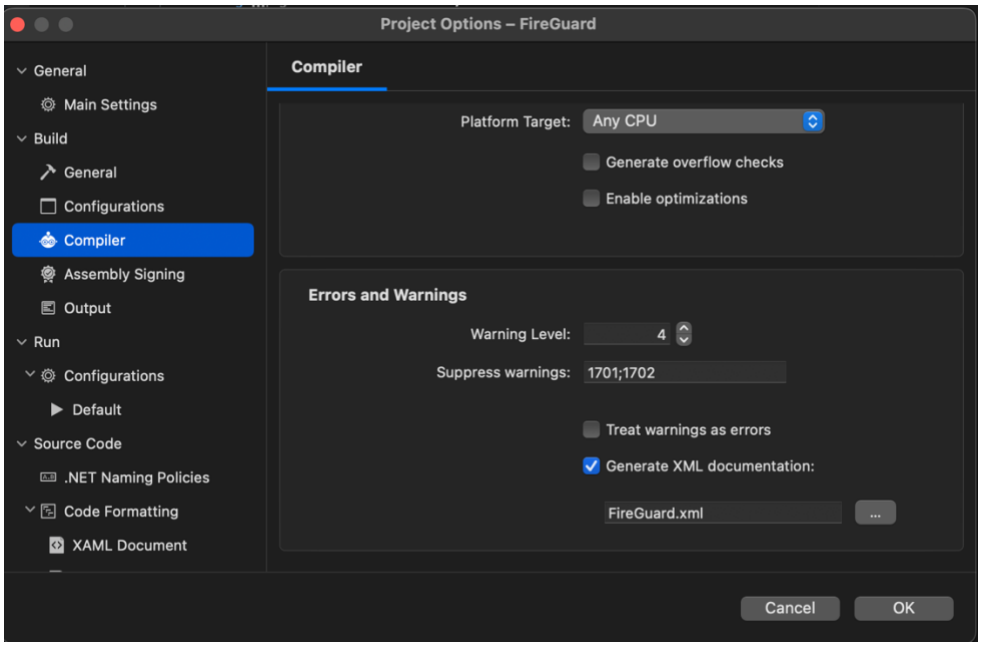


*Figure 30. How to generate the documentation XML file on Mac (2/3)*

4. Run the project and redirect to the file and view the XML documentation (Figure 31).

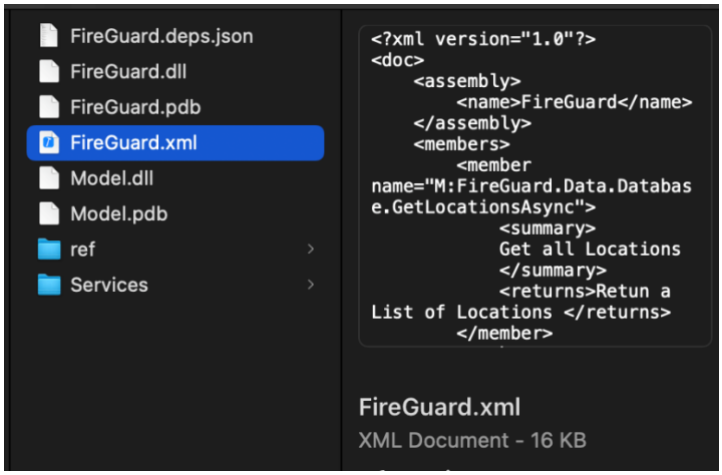


*Figure 31. How to generate the documentation XML file on Mac (3/3)*

## 9.3 Guide

Both operating systems can follow the same guide for creating documentations for new methods and classes.

All implemented methods should have written documentation to make it manageable for other developers to continue implementing any systems. To auto-generate documentation tags for a method, simply type "///", as seen In Figure 32 [17].


*Figure 32. How to auto-generate documentation tags*

When the method is hoovered over, the written documentation will be displayed (see Figure 33).

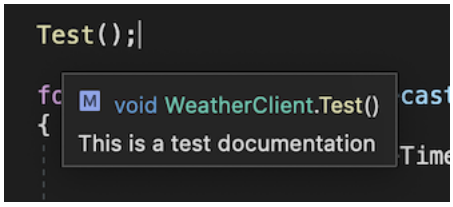


*Figure 33. The written documentation can be seen when method is hoovered over*

# Continuous Integration and Testing

Tests for both FireGuard and Fire risk model projects were created, seen in Figure 34. The library NUnit was utilized when creating the test project.

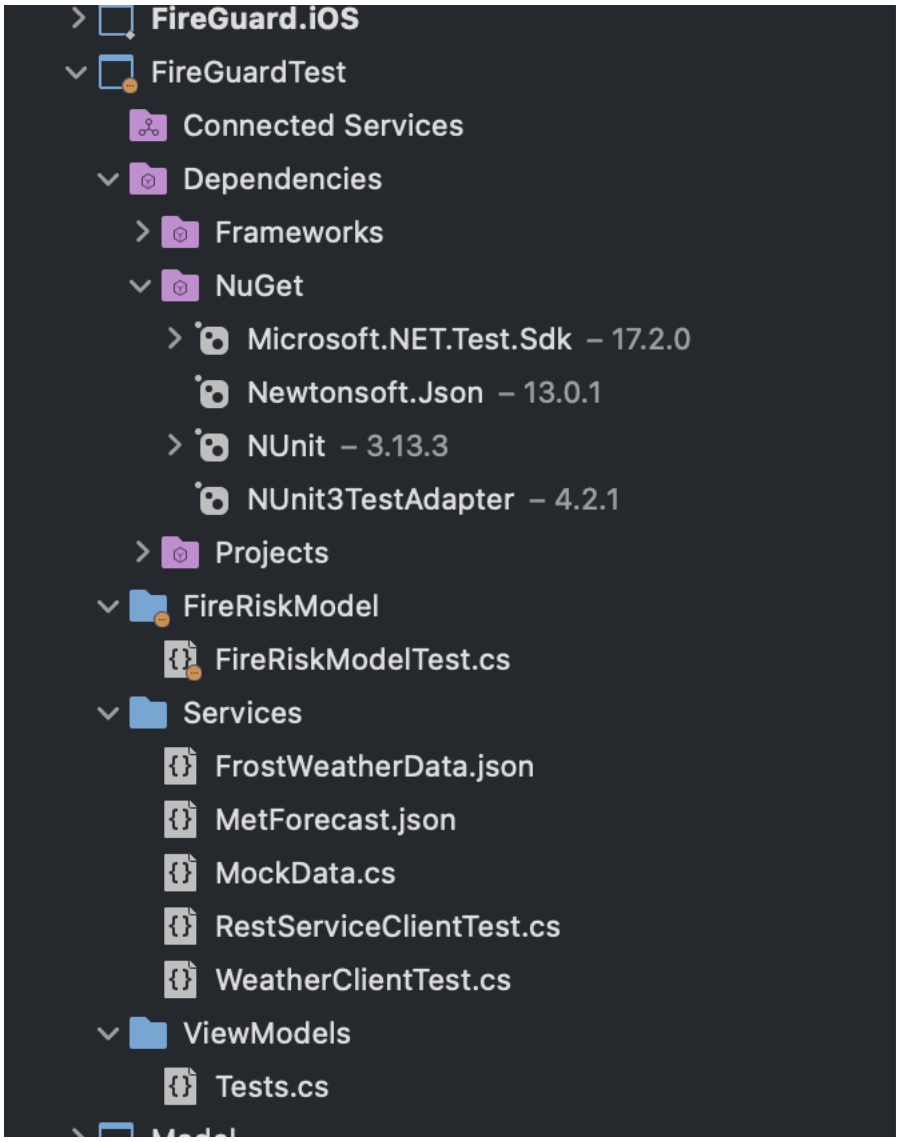


*Figure 34. Representation of the FireGuardTest project*

To test methods that depend on actual weather data, it was decided to set up mock data sets that could be used. `FrostWeatherData` and `MetForecasts` consist of JSON representations. `MockData` class consist of methods for retrieving data from the representation.

## 10.1 Test methods

`FireRiskModelTest` class consists of the following test methods:

- `MockDataTest` : test that the reading of the JSON file was successful
- `FireRiskTtfTest`: test that the fire risk model provides fire risk results
- `GetHourlyObservationstest`: test the `RemoveNotHourlyForecasts`, which determine the time jump from hourly to every sixth hour and removes all future forecasts.
- `SortByMinTtfTest` : test that the method `SortByMinTtf` groups the time to flashover by date, and retrieves the lowest ttf value and returns a list
- `FindFireRiskTest`: test if the method calculates the fire risk correctly based on the ttf benchmarking

`MockData` class consists of the following test methods:

- Classes to retrieve weather data from JSON files
- `Observations` and `ForecastObservations`: retrieves data from the JSON file and convert the data to `Observation` objects
- `ConvertObservations` and `ConvertForecasts`: converts the weather data to `Observations` data
- `RemoveNotHourlyForecasts`: determine where the forecasts time jump is, and remove all future forecasts from the list.

`RestServiceClientTest` class consists of the following test methods:

- `GetForecastTest`: test for retrieving forecasts from met
- `GetNearestStationTest`: test for retrieving the nearest station based on coordinates
- `GetWeatherDataTest`: test for retrieving observations from the nearest station for a given period

`WeatherClientTest` class consists of the following test methods:

- `SortByMaxWindSpeedTest`: the method groups the weather data by date, and determine the maximum wind speed value for each day

- `SortByMaxWindGustTest`: test if the method sorts the wind gust and direction correctly

`Tests` class consists of the following test methods:

- `GetPeriodTest`: test if the period string is generated correctly

## 10.2 Run test project

The following chapter will explain how to set up the test project and get everything up and running.

These are the required NuGet packages (Figure 35):

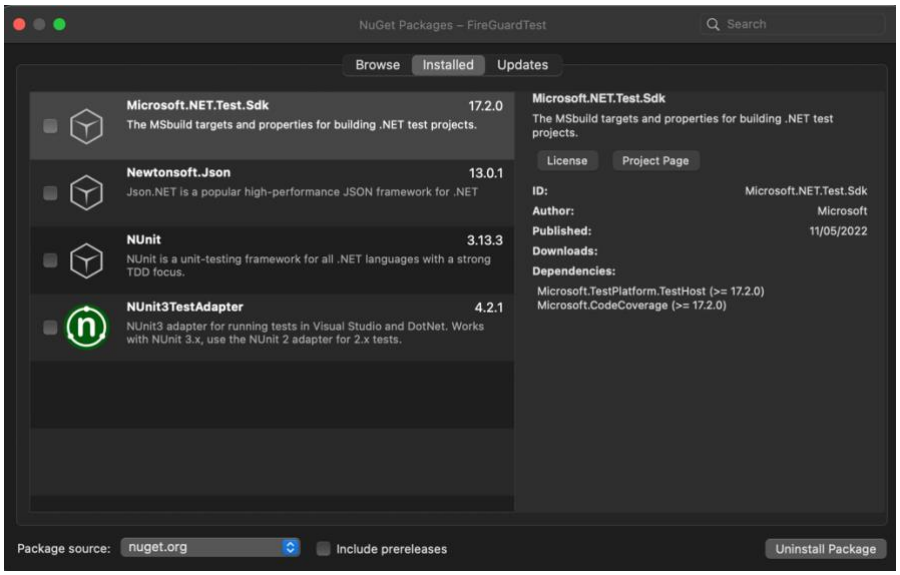


*Figure 35. The required NuGet packages for FireGuardTest project*

The `RestServiceClientTest` consists of tests to check if the request was successful.

Most of the methods depend on weather data, therefore it was decided to create a mock data set. This can be found in the `MockData` class that uses the two JSON files to retrieve data.

To use the mock data, the user needs to specify the file path, because it will vary (Figure 36).

A detailed description how to retrieve the full path is given below.

```
public class MockData
{
    public MetForecast MetForecastData()
    {
        //endre path her
        //du kan kopiere full path fra solution explorer om du høyreklikker på MetForecast.json
        Directory.SetCurrentDirectory(@"/Users/emilieh.fisketjon/GitHub/IB3/Untitled/FireGuardTe
        StreamReader reader = new StreamReader("MetForecast.json");
        string jsonString = reader.ReadToEnd();
```

*Figure 36. The MetForecastData method*

## To find the full path on Mac:

Figures 37, 38, and 39 provides a step-by-step guide for retrieving the files full path.

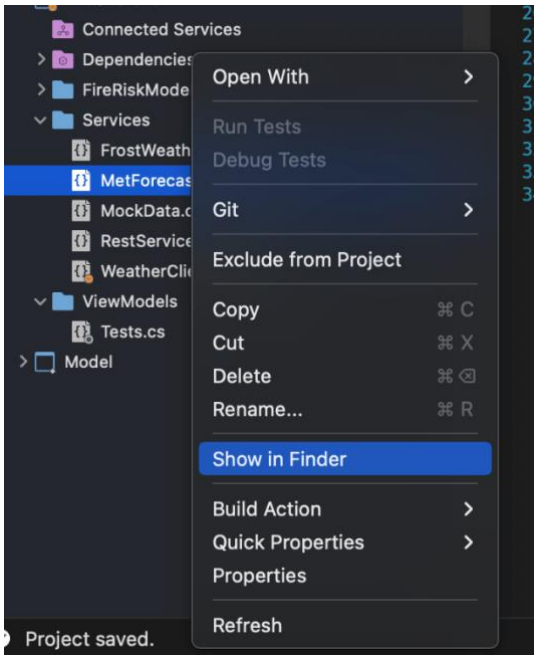


*Figure 37. How to retrieve the full path on Mac (1/3)*

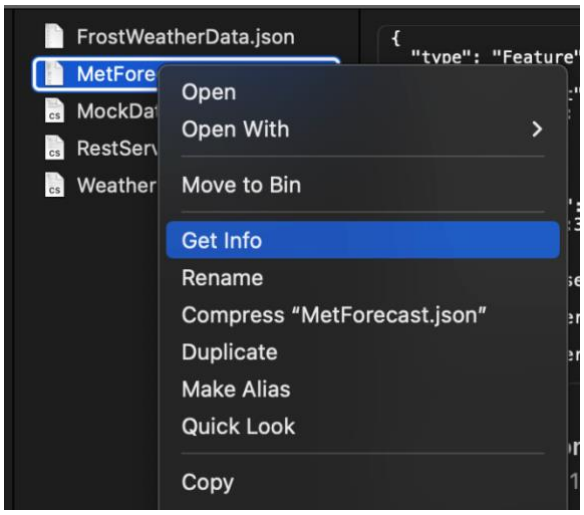


*Figure 38. How to retrieve the full path on Mac (2/3)*

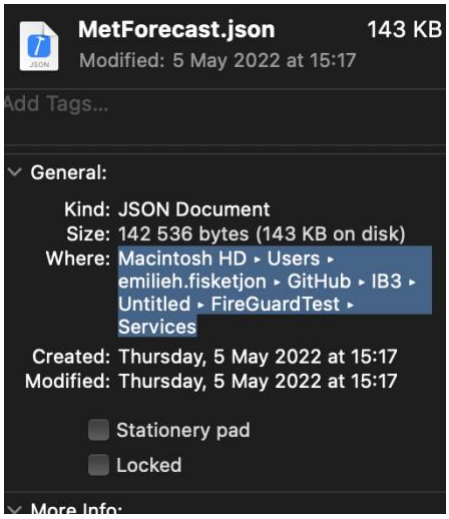


*Figure 39. How to retrieve the full path on Mac (3/3)*

Copy and paste the file path into the `MockData` class `Directory.SetCurrentDirectory(.....)`

If the developer wants to directly work with weather data from the API, methods in `IRestService` and `IWeatherClient` can be utilized.

## To find the full path on Windows:

Right-click on JSON file and choose Copy Full Path and paste into `MockData` class.

# Bibliography


1. Meterologisk-Institutt. *About the Norwegian Meteorological Institute*. 2017 2020; Available from: https://www.met.no/en/About-us/About-MET-Norway.
2. Yr.no. *Facts about Yr*. Available from: https://hjelp.yr.no/hc/en-us/sections/115001514149-About-us.
3. Meterologisk-Institutt. *WeatherAPI*. Available from: https://api.met.no.
4. Meterologisk-Institutt, *Locationforecast.*
5. Meterologisk-Institutt. *What is Frost?* . Available from: https://frost.met.no/index.html.
6. Yr.no. *Location Forecast*. Available from: https://developer.yr.no/featured-products/forecast/.
7. Log, T., *Modeling Indoor Relative Humidity and Wood Moisture Content as a Proxy for Wooden Home Fire Risk.* Sensors, 2019. **19**(22): p. 5050.
8. Meterologisk-Institutt. *API REFERENCE*. Available from: https://frost.met.no/api.html.
9. Meterologisk-Institutt. *API Concepts*. Available from: https://frost.met.no/concepts2.html.
10. Wikipedia. *ISO 8601*. 2022; Available from: https://en.wikipedia.org/wiki/ISO_8601.
11. Meterologisk-Institutt, *Request New Client Credentials.*
12. Meterologisk-Institutt. *MET's Privacy Policy Statement*. 2017 2021; Available from: https://www.met.no/en/About-us/privacy.
13. Meterologisk-Institutt. *Authentication*. Available from: https://frost.met.no/authentication.html.
14. IBM. *HTTP basic authentication*. 2022; Available from: https://www.ibm.com/docs/en/cics-ts/5.4?topic=concepts-http-basic-authentication.
15. auth0. *What is OAuth 2.0?* ; Available from: https://auth0.com/intro-to-iam/what-is-oauth-2/.
16. Stokkenes, S., et al., *Validation of a Predictive Fire Risk Indication Model using Cloud-based Weather Data Services.* Procedia Computer Science, 2021. **184**: p. 186-193.
17. *Recommended XML tags for C# documentation comments.* 2022.

# Table of Figures