# Design document

COMP.SE.110-2020-2021-1 Software Design

Henri Seppä H262947
Juho Keto-Tokoi H283635
Anniina Honkasaari H283117
Jimi Riihiluoma H252785

# Table of contents

**Table of contents** 2

**1. Introduction** 3

**2. Architecture** 3

**3. Boundaries and Interfaces** 3

**4. Reasoning** 4

# 1. Introduction

This document depicts the software we are producing for the course COMP.SE.110-2020-2021-1 Software Design. Our software uses two different APIs, one from FinGrid and one from FMI. We are going to use these APIs in our program to view weather data, power consumption data and power production data both separately and together, to show how one affects the other. The application uses C++ and QML.

# 2. Architecture

Our application follows the MVC pattern. It contains a model, view and a controller. The model can be thought of as a backend, since it does the logic behind the app and is not used by the user directly. The view, controller and GUI make up the application's front-end. Only the controller has dependencies to both model and the view. The model and the view are independent of each other. A visual representation of the application's architecture can be found in the docs folder (Project-class-diagram.png). A sketch of the look for the GUI can also be found in the same folder (gui.png).

The model class stores different kinds of data, for example weather data and electricity consumption data. These all implement a common data interface, which has a list of data points and a specified unit (for example GWh for electricity consumptions). The responsibility to make API calls has been set to a separate API caller class. The API caller is then used by the model to fetch new data and parse it from XML to C++ containers.This is done with QXmlStreamReader, which can be used to parse XML-files.

In a nutshell, the user interacts with the controller to modify the data in the model and to modify the GUI. All of the functionality is done using Qt's components.

# 3. Boundaries and Interfaces

The user interacts with the controller class through the GUI. The controller makes function calls to the model and modifies the model according to the parameters that the user has set in the GUI. The model then returns data to the controller, and the controller forwards the data to the view. The view stores the data from the model in a way that it can be visualized by the GUI. In other words, the controller acts as a middle man between the view and the model.

The Model:

Contains the data and logic to update it by using the API caller. Contains multiple types of data, which all implement a common data interface.

The View:

Contains data from the model that is relevant to the GUI, and is stored in a way that can be displayed by the GUI.

The Controller:

Is triggered by the actions of the user. Acts as the middleman between model and view, passing data between them.

# 4. Reasoning

We have created the architecture in this way, so we have clear components and interfaces. Each part has a distinct job and an area of responsibilities, and they don't overlap. We decided to use this MVC pattern, because there was plenty of information and guides to use it and in our opinion it makes the functionality a lot clearer. With the View and Model separated, the application is modular and data can be easily passed between the front- and backend. QNetworkAccessManager and QXmlStreamReader are used in the back end, because they come with QT and don't require any external libraries. To make https-requests, we have included OpenSSL in our project, which is easy to set up for the project. OpenSSL is used by QNetworkAccessManager.

The application architecture is planned to comply with SOLID principles, so that the application would be easily maintainable and expandable during the development process.