# 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

# 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 in the same window, to show how one affects the other. The application is made with C++ and QML/Qt. The current state of the implemented GUI can be seen in the docs folder (GUI.png).

# 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 business logic behind the app and is not used by the user directly. The C++ view class and QML GUI make up the application's front-end. The model and the view are independent of each other. All data flow from view to model and model to view happens through the controller. A visual representation of the application's architecture can be found in the docs folder (Project-class-diagram.png).

The Backend class serves as an interface between the controller and the model. It relays data requests to a separate API_Caller class, which maps the implemented API classes to data types they provide. All API classes are derived from a parent API class, which contains the implementation of making API requests. The derived classes handle parsing of the API reply's XML or JSON format data into C++ containers and relay the parsed data back to the Backend class. XML data is parsed with QXmlStreamReader. Also JSON data is parsed with QT libraries. A new API is added by creating a class for it (derived from API class) which implements an API-specific parser.

Data is stored in the Datamodel class. All data implements a common data interface, which contains a list of data points, data type and a specified unit (for example GWh for electricity consumption).

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 and STL libraries.

# 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 modular components and clear 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.

# 5. Self-evaluation

Our original design has worked out very well in our implementation. The MVC design enabled us to edit front- and backend separately, because they are not directly connected. Instead they are connected via the Controller.

So far we have implemented features to create temperature charts from real data, multiple at a time. Our design has again supported the implementation well, and we haven't found a need for big changes in the program. Some single functions have been added and changed due to the syntax of the code being different than thought before, but the logic is still working almost in the same way as in our original design.

Things we have changed have been the addition of a backend-class, the functionality of which is to orchestrate the data flow of the model. It serves now as an interface between controller and the model. Another addition is the API-parent class, from which all the different API-classes are derived.

While we plan our future implementation and the features to be added (save, load, different datatypes etc.), for now we think that this design supports them as well. We anticipate that we might have to add a class which handles the saving of data, if the Datamodel class becomes too crowded.

Our design choices help us to keep the software quality high and reliable. Our classes follow the design principles we have decided to use in our program and help us to keep the program expandable. The design also makes sure that the program functions properly in every situation.