PowerWeatherApp

Dependencies

- Using QT 5.15.2 (MSVC 2019), minGW 8.1.x
- QT: quick, widgets, network, quick charts (.pro file includes)
- OpenSSL 1.1.1d or newer

Design

- Using two different controllers for power (Fingrid) and weather (FMI) which are inherited from parent controller.
- We're using semi-active MVC (pull model).
- Using QML objects for viewing relevant data according to the user's preferences.
- Backend for requesting API information is done by using C++.
- Frontend is done using QML.

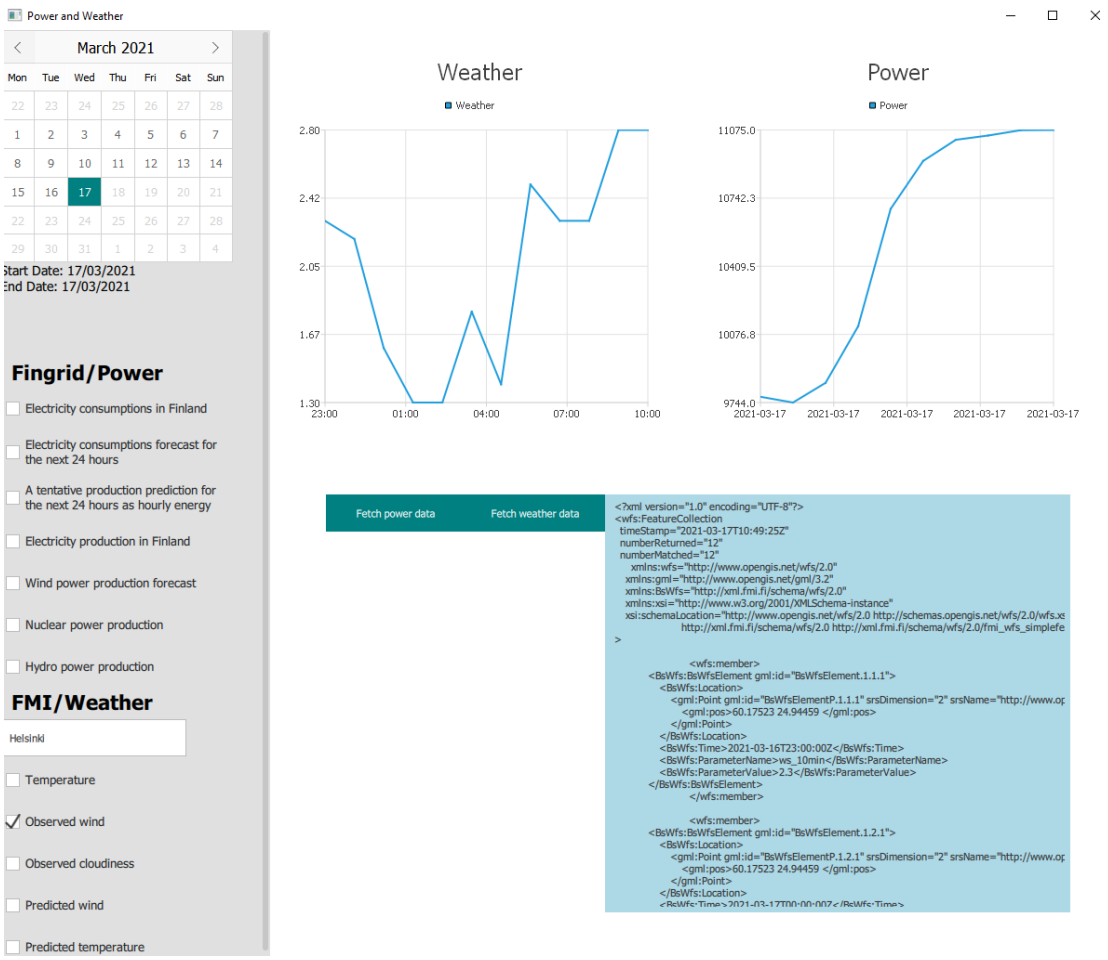

Photo 1: first design for the prototype phase

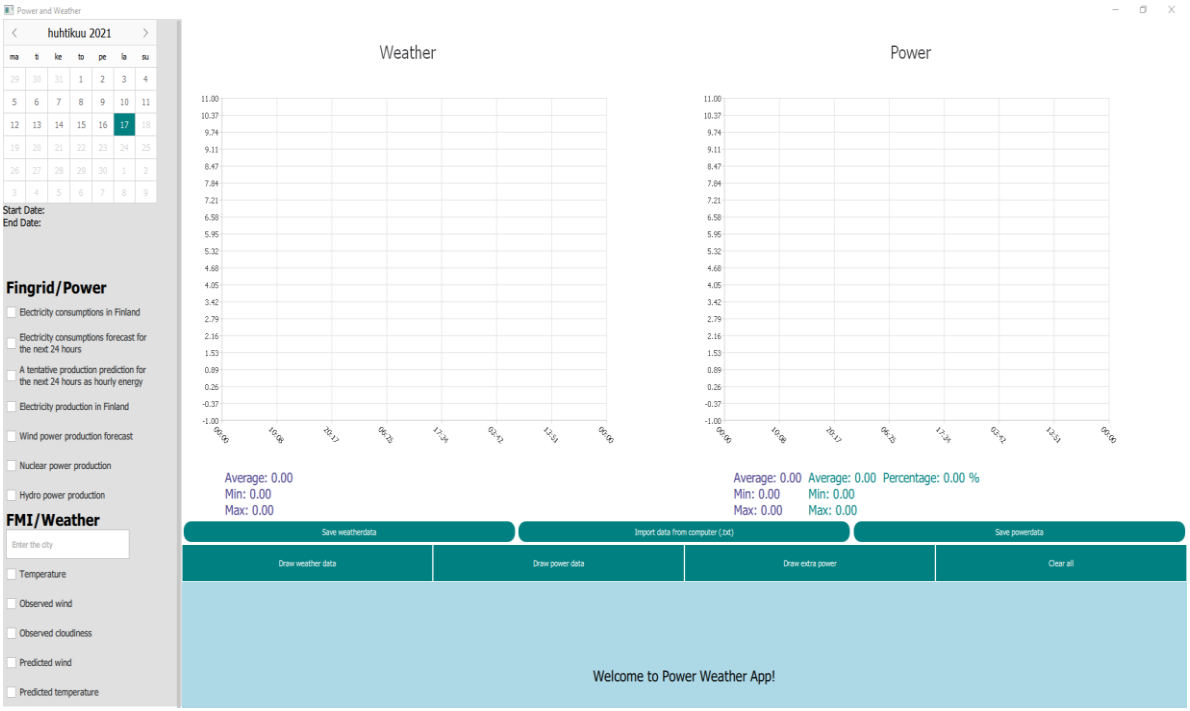

Photo 2: design for final submission

Instructions

- This app is best scaled for full hd screen (1920x1080)
- Open .pro file
- Change run directory to powerweatherapp/
- Compile the code
- To start using the app the user must choose start and end dates. Then depending on the datatype user can either type a city (for weather) or not (for power). User then proceeds to choose data type from checkboxes. User then fetches the data by clicking draw graph for desired data type. Multiplotting powerdata can be done by first plotting one data type and then proceeding to pick another checkbox and drawing the data by clicking draw extra power. User can clear all graphs by clicking on clear all graphs. User has the option to save the data by save buttons either as .txt or .png files. User can then import the data into the app whenever needed. Under the draw and clear button user can see instructions about the different stages the program is in or what the user needs to change in order to get the data plotted.

Functionalities

- Plotting data from FMI or Fingrid
- Zoom into the graph
- Save data as .txt or .png
- Import saved data (.txt)
- Plotting two different power datas to same graph for analysing purposes

Components

- User is able to choose what kind of data user wants to get in the view component.
- User's preferences go to the controllers and the URL is build based on those preferences. The preferences are then sent into the models.
- Models read data with APIReaders and parse that data with data parser. Models also hold the data.
- Parsed data is displayed with the charts in the view.
- User get instruction texts from a statistics class
- MVC model isn't strictly followed in the view part, we found it much simpler in the code to implement the view so that we control it by communicating with the model directly without going through a pretty obsolete controller directly for the view.

Class diagram

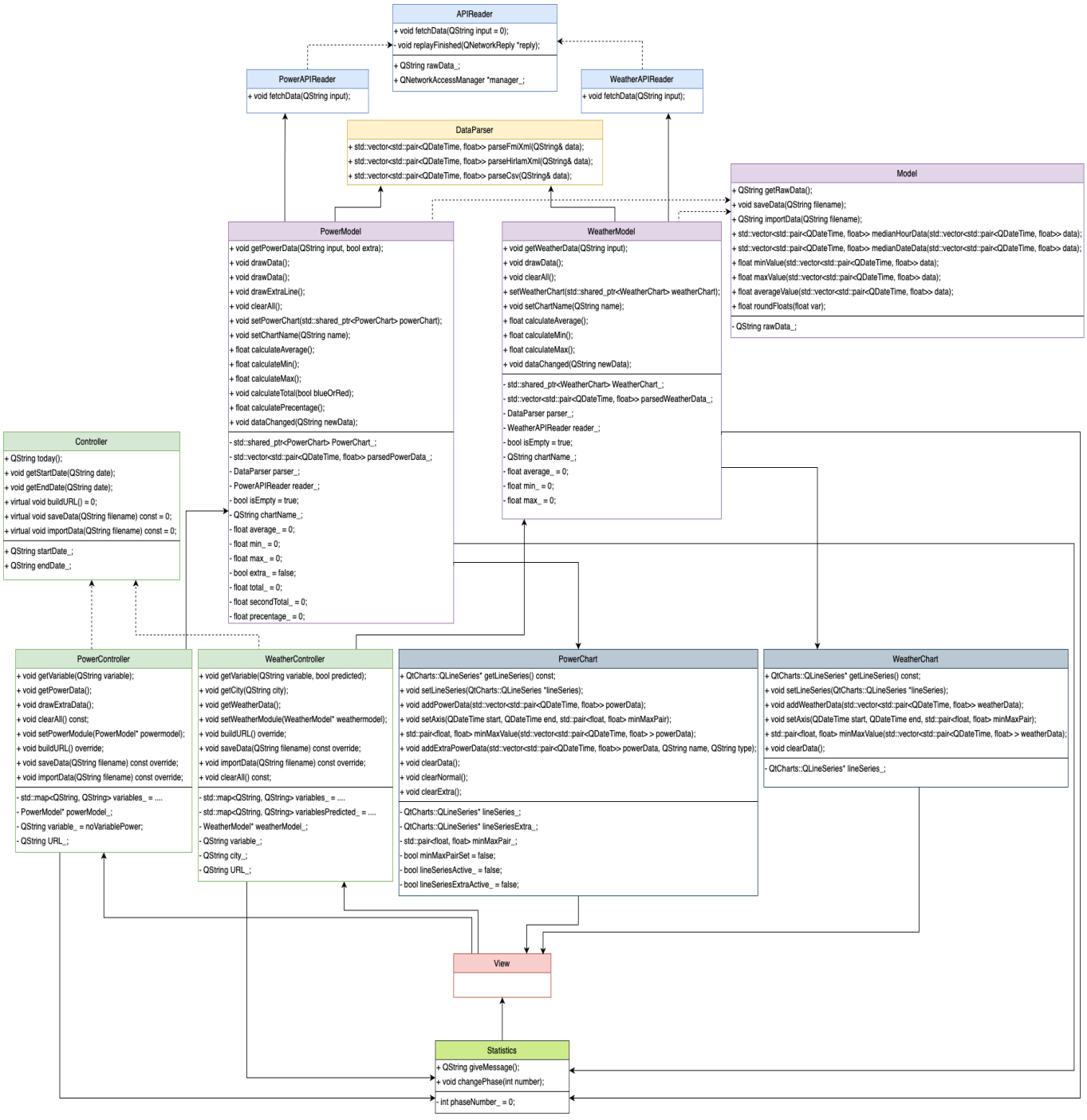

Diagram 1: class diagram for the app

Internal interfaces

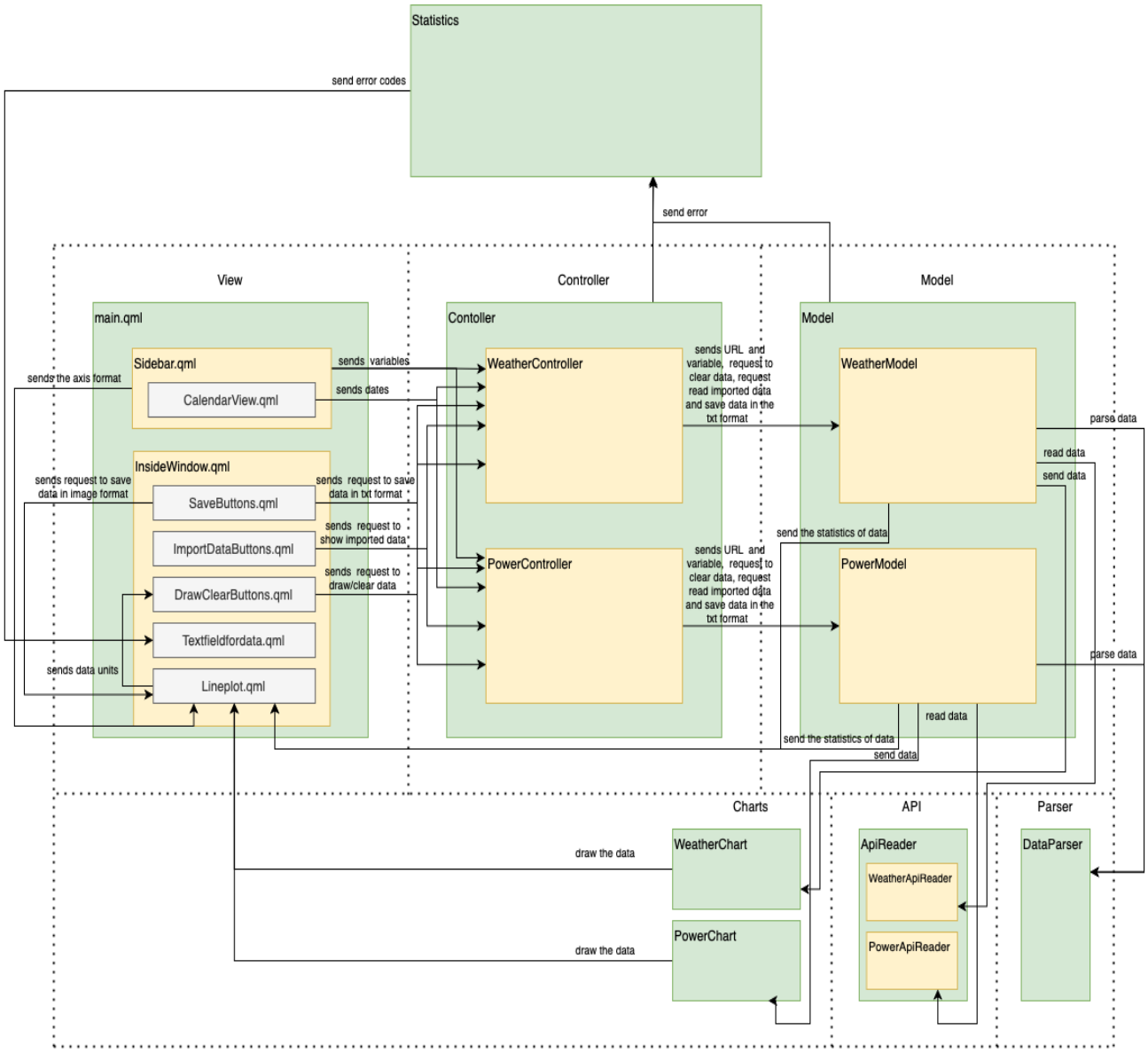

Diagram 2: internal interfaces of the app

Decision-making process

Prototype (DONE)

- Right now, we're thinking of making two different models for the different APIs because the data differs too much, and we couldn't find a proper relation between the two APIs. In the future, if necessary, we separate controller from the model classes. (DONE)
- Both APIs need to be parsed separately because we get power as csv format and weather as xml format. (DONE)

Midterm (DONE)

- Based on prototype phase meeting we decided to change our class diagram to more efficient solution (inheritance). We also changed the way how data flows between classes (through controllers).
- We inherited controllers from a parent controller and we're thinking of doing the same thing to models and charts in some point. (DONE)
- UI databox will probably be changed into an instruction box. Its functionality is telling the user some feedback of the choices made. (DONE)
- We didn't separate the parsers because they only have two functions. Their responsibilities are to only parse raw data.

Todo (after midterm)

- We still need to implement some functions into FMI/Weather (observed cloudiness, predicted wind, predicted temperature). (DONE)
- We will add a feature to make the view component able to give feedback to the user according to the preferences. (DONE)
- We are planning that we could save the chart in .png format and save the data in .txt file so you could load the data without online connection. (DONE)
- Commenting the code better and making it overall more readable. (DONE)

Final submission

- We added rest of the mandatory weather functions. Because the predicted data doesn't come from the same URL as all the non-predicted data, we had to implement a new parser for said URL. We need to keep track of which data type for weather is selected and based on that limit or allow the user to select different dates to be searched for. For example, is user chooses to search for predicted data then the dates automatically go forward 48 hours since FMI hirlam predicted data only goes into the future by 48 hours.
- From midterm submission we added the ability to do basic calculations based on the received data. Now when for example weather data is fetched you get the min, max and average values from the graph. Being a requirement in the guidelines, one can fetch 2 different power data's and get the total percentage the second fetched data is from the first. For example, how much wind power production is from the electricity production in Finland.
- We implemented the functionality for the user to save the graphs fetched as a .png picture. This wasn't a mandatory feature but since it gets more points and was pretty easily implemented, we added it.
- We implemented user feedback console which wasn't mandatory, but we thought it made sense to give the user some feedback about his actions.

Self-evaluation

- After prototype phase meeting our UI has not changed much. We are happy with the original UI. After midterm submission we scaled all the app components so that the user can resize the window. The UI still looks very similar to the very first UI design we thought of (photo 1, photo 2).
- We were able to stick to our own original design. During coding some new classes had to be added to the original plan since we didn't account for example statistics in the prototype phase.
- In the beginning we could have been planning bigger picture better. We should have been keeping an eye on the mandatory features better because now for example we had to do compromises on the power production percentage calculations. If we had paid attention to this problem better we could have made a ghost query of the production in the first place.
- Our approach was pretty much fix and go. We found an issue which we then fixed and moved on to the next issue. We didn't plan much ahead when fixing single errors. We probably could have fixed multiple issues at once if we had planned ahead.
- In the beginning we could have made better plans for the chart class. In the final submission we have 2 separate chart classes which could have probably been made into one class.
- We have split the workload pretty evenly and we have all agreed that everybody has participated in the making.
- Most of the productive coding has been done via Discord meeting where everybody in the group attended. Git commits might be a little uneven mostly because we have been coding together on one or two computers at a time. Other people in the call have been so called 'back seat gaming' and helping with the current problem in hand.
- Aliisa has done most of QML side and connecting C++ controller to QML. Arttu has been responsible for the charts and models. He has also been the middleman between C++ and QML. Lari and Joel did data parsers together. Lari did statistics for user to TextFieldData box and file saving. Joel has been mainly focusing on the weather sided C++ code and fetching weather data from FMI API.

Found bugs

- Power data in the dates 26.3.2021 - 29.3.2021 acts oddly. We anticipate that the issue comes from the api's data however we couldn't confirm this to be true.
- If user selects start date as today from calendar and the time is before 12AM then the end date won't follow. User needs to access today's values for date from calendar after 12AM. QDebug shows the correct Finnish date, but the calendar class from QT creator doesn't seem to apply this.