Unnamed

Design Document

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

Joonas Pelttari
Kalle-Henrik Raitanen
Jani Uolamo
Sipi Ylä-Nojonen
18.4.2021

High level description

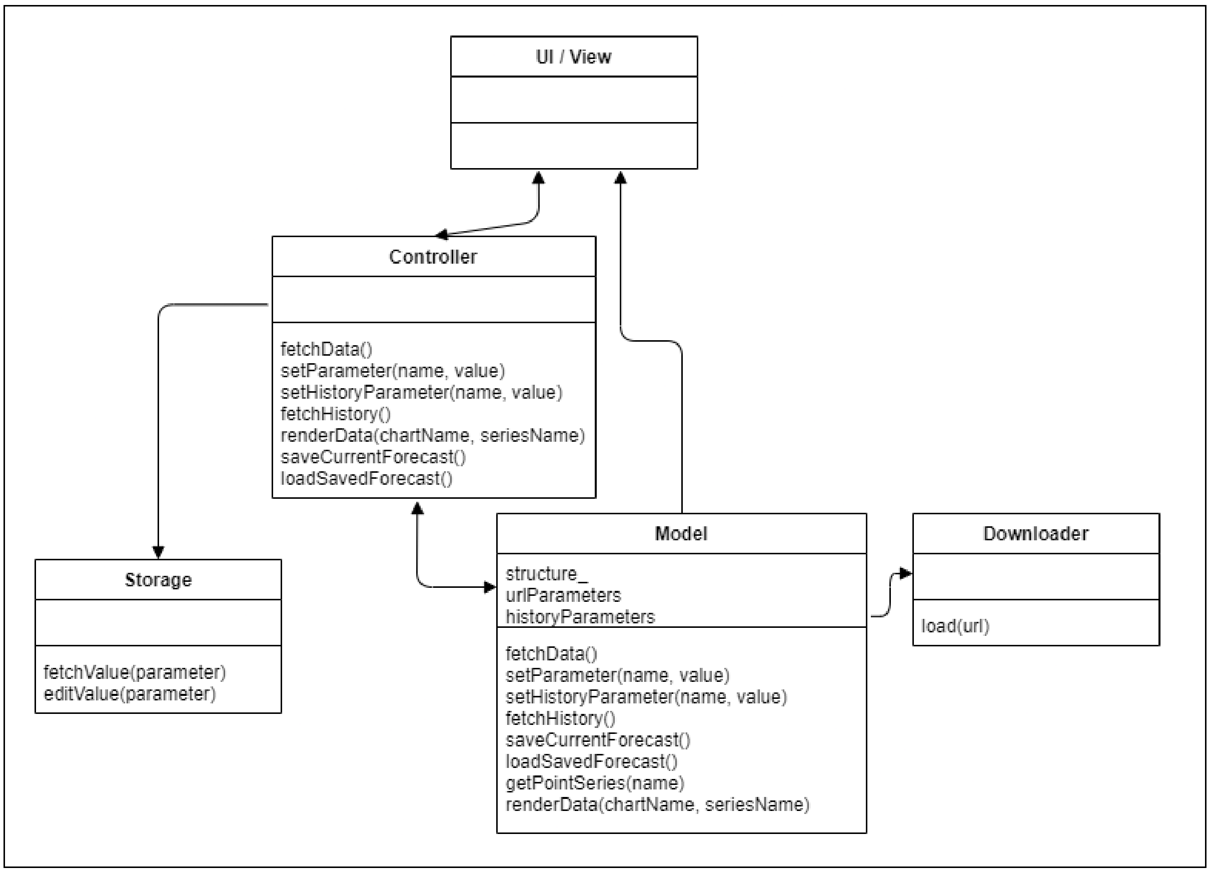
The applications GUI is implemented with QtQuick using QML. Other classes for application include Model, Controller and DownLoader which interfaces are further discussed in “Boundaries and Interfaces” section.

QtCharts is included as additional Qt-library to easily present data visualizations in a user-friendly way. From standard library included with QtCreator, multiple classes have been used, especially for implementing the GUI smoothly and flexibly as QML application as well as using QSettings class for easily saving user preferences by parsing them into Windows registry or on Linux into an .ini file.

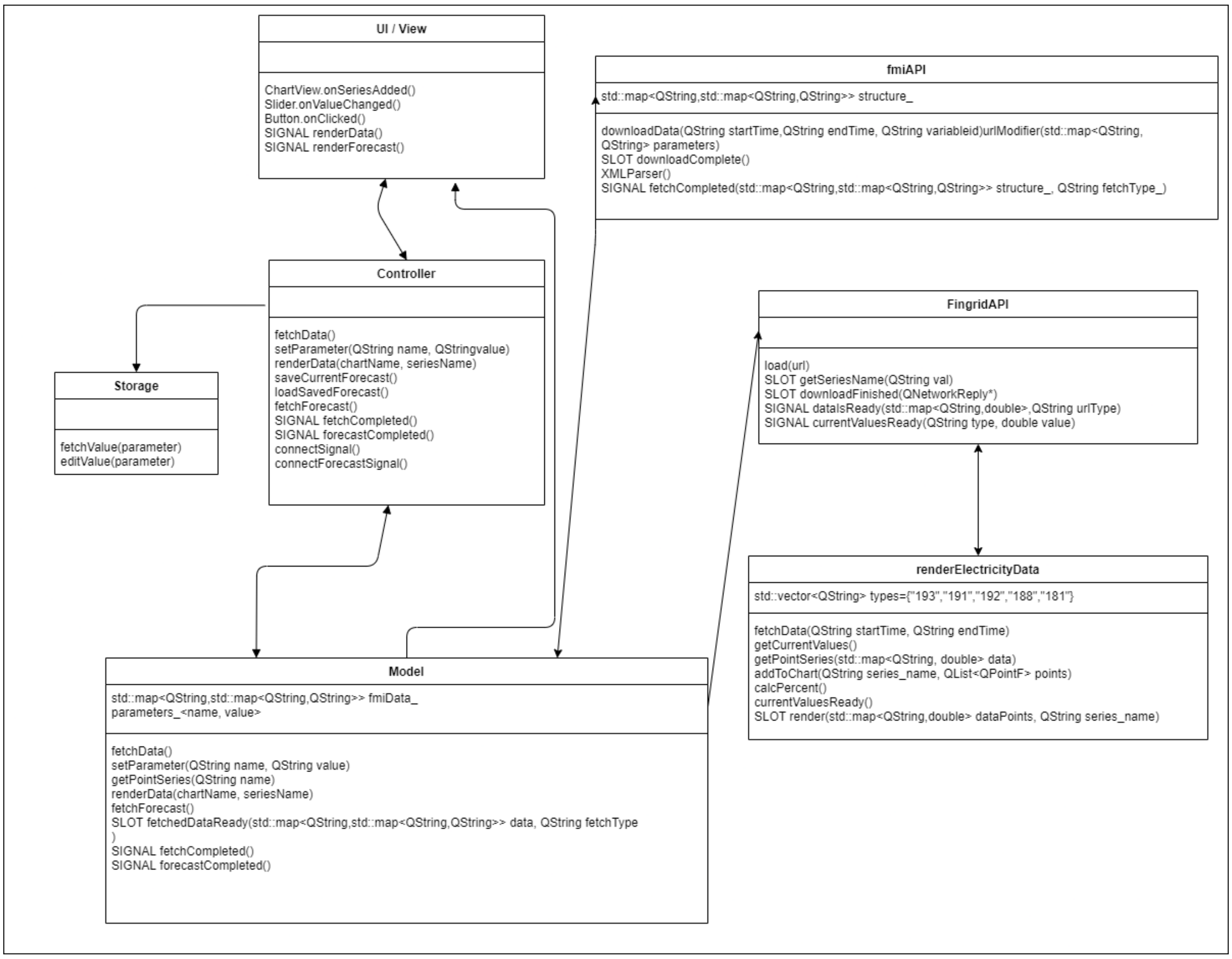
UI contains three views that separate displayed data into categories Weather/Power consumption/ Power production. Each view contains several graphs for separate predictions from each category (temperature, wind speed, visibility / total production, water production, wind production etc.) These can be individually shown or hidden based on user selection. In addition to this under the graphs are shown current values for weather and current total energy production + consumption.

All in all, utilizing the flexibility and simplicity of QML to great extended has guided a lot of the design as can be seen from the final View being implemented as part of UI on QML side of the program. This is also displayed by usage of Qt’s signal / slots system.

Class diagram - Midterm planning phase



Class diagram - Final implementation


UI / View
ChartView.onSeriesAdded()
Slider.onValueChanged()
Button.onClicked()
SIGNAL renderData()
SIGNAL renderForecast()
fmiAPI
std::map<QString,std::map<QString,QString>> structure_
downloadData(QString startTime,QString endTime, QString variableid)urlModifier(std::map<QString, QString> parameters)
SLOT downloadComplete()
XMLParser()
SIGNAL fetchCompleted(std::map<QString,std::map<QString,QString>> structure_, QString fetchType_)
Controller
fetchData()
setParameter(QString name, QStringvalue)
renderData(chartName, seriesName)
saveCurrentForecast()
loadSavedForecast()
fetchForecast()
SIGNAL fetchCompleted()
SIGNAL forecastCompleted()
connectSignal()
connectForecastSignal()
FingridAPI
load(url)
SLOT getSeriesName(QString val)
SLOT downloadFinished(QNetworkReply*)
SIGNAL dataIsReady(std::map<QString,double>,QString urlType)
SIGNAL currentValuesReady(QString type, double value)
Storage
fetchValue(parameter)
editValue(parameter)
renderElectricityData
std::vector<QString> types={"193","191","192","188","181"}
fetchData(QString startTime, QString endTime)
getCurrentValues()
getPointSeries(std::map<QString, double> data)
addToChart(QString series_name, QList<QPointF> points)
calcPercent()
currentValuesReady()
SLOT render(std::map<QString,double> dataPoints, QString series_name)
Model
std::map<QString,std::map<QString,QString>> fmiData_
parameters_<name, value>
fetchData()
setParameter(QString name, QString value)
getPointSeries(QString name)
renderData(chartName, seriesName)
fetchForecast()
SLOT fetchedDataReady(std::map<QString,std::map<QString,QString>> data, QString fetchType
)
SIGNAL fetchCompleted()
SIGNAL forecastCompleted()

Boundaries and Interfaces

Controller

Controller class is responsible for processing and delegating user requests from the user interface whereas Model stores downloaded data and updates user view according to orders from controller.

- std::shared_ptr<Model> model_, reference to Model object. **Definition added in final version.**
- std::shared_ptr<Storage> storage_, reference to Storage object. **Definition added in final version.**
- QQmlApplicationEngine* engine_, reference to QmlApplicationEngine, UI/View. **Definition added in final version.**

- fetchData(), calls for Model’s interface to initiate data download,
- setParameter(QString name, QString value), calls for Model to set single query url parameter (eg. City) by name, **Added since the prototype plan**
- renderData(chartName, seriesName), requests Model for rendition of changed data in view.

- loadSavedForecast(), loads saved forecast data based on currently selected timeline if found, **Added since the prototype plan**
- fetchForecast(), requests weather forecast for next 24 hours. **Added in final version.**
- SIGNAL, fetchCompleted(), signals to start rendering weather data in qml. **Added in final version.**
- SIGNAL, forecastCompleted(), signals to start rendering weather forecast in qml. **Added in final version.**
- ConnectSignal()
- fetchSettingsValue(QString key), get saved preference by key from Storage.
- editSettingsValue(QString key, QVariant value), modify saved preference to new value in Storage by given key.

UNIMPLEMENTED

- setHistoryParameter(name, value), Call for model to set starting or ending time for long-scale history fetch, **Added since the prototype plan**
- fetchHistory(), Call for Model to fetch long-time history based on saved starting and ending point, **Added since the prototype plan**
- saveCurrentForecast(), calls for Model to save currently displayed forecast data for later use, **Added since the prototype plan**

Model

Model class implements storaging of downloaded data as well as interface for controller to send user requests that are then displayed into the GUI through view. Model itself is mainly responsible for updating the shown data based on eg. City, but all categories of data are fetched every time. The selection between actual shown graphs is done in view in QML side.

- parameters_<name, value>, map that contains parameters for constructing query based on user preference, (City (weather specific), starting time, ending time, scale (energy specific)), **Added since the prototype plan. Renamed from urlParameters to parameters_.**
- std::map<QString,std::map<QString,QString>> fmiData_, Contains fetched weather data points for each time point separated into a map by parameter name, **replaces structure_ in final version.**
- fmiapi_, fmiApi object for fetching weather data.
- elecData_, electricityRenderData object for fetching and rendering electricity data.

- fetchData(), constructs url from parameters and calls for DownLoader to download data
- setParameter(name, value), Sets single query url parameters (eg City) by user setting, **Added since the prototype plan**
- getPointSeries(name), returns data points respective to series defined by name eg. temperature, power production
- renderData(chartName, seriesName), pushes request for View to update data in UI.
- fetchForecast(), initiates dowload for weather forecast for next 24 hours. **Added in final version.**
- SLOT fetchedDataReady(std::map<QString,std::map<QString,QString>> data, QString fetchType), receives parsed weather data and emits fetchCompleted or forecastCompleted to start rendering respectively based on parameter type. **Added in final version.**

- SIGNAL fetchCompleted(), signals for rendering weather data to start, connected to Controller::fetchCompleted(). **Added in final version.**
- SIGNAL forecastCompleted(), signals for rendering weather forecast to start, connected to Controller::forecastCompleted(). **Added in final version.**

UNIMPLEMENTED

- setHistoryParameter(name, value), Set starting or ending time for long-scale history fetch, **Added since the prototype plan**
- fetchHistory(), initialize data collection for large-scale history fetch, **Added since the prototype plan**
- saveCurrentForecast(), saves currently displayed forecast data for later use, **Added since the prototype plan**
- loadSavedForecast(), loads saved forecast data based on currently selected timeline if found, **Added since the prototype plan**

RenderElectricityData, Class separated from Model's rendering for weather data to a separate interface serving electricity data rendering

- std::vector<QString> types={"193","191","192","188","181"}, contains possible data types feched from Fingrid;
- FinGridAPI fingrid_, reference to FingridAPI object.
- QQmlApplicationEngine* engine, reference to QmlApplication object.

- fetchData(QString startTime, QString endTime), fetches data between set calls Fingrid data fetches for each data type listed in types, between set startTime and endTime. **Added in final version.**
- getCurrentValues(), Calls for Fingrid data of each listed data type for current point in time. **Added in final version.**
- SLOT render(std::map<QString,double> dataPoints,QString series_name), calls to plot all points from defined dataPoints, to defined LineSeries. **Added in final version.**
- getPointSeries(std::map<QString, double> data), maps data points from data into LineSeries compatible form (QPointF), similar to Model::getPointSeries. **Added in final version.**
- addToChart(QString series_name, QList<QPointF> points), adds defined points to LineSeries element by defined name. Similarly to Model::renderData(). **Added in final version.**
- calcPercent(), calculates water/wind/nuclear energy production as percentage of total produced. **Added in final version.**
- currentValuesReady(), Debug print for values fetch. **Added in final version.**

Downloader class was split into two separate interfaces for fmi and Fingrid both working similarly but adapted to respective source in their fetching process.

FmiAPI, **Added in final version**

DownLoader class is used to handle connections and fetching data from internet sources to be stored by Model. **FmiAPI handles downloads from FMI.**

- std::map<QString,std::map<QString,QString>> structure_, map for reading downloaded data into
- QString fetchType_, holds whether forecast or history data was fetched.
- QString urlModifier(std::map<QString, QString>, QString), map for presaving fetching url parameters.
- QStringurlFMI, full url for FMI data fetch
- QString urlFMIforeCast, full url for FMI forecast fetch
- QNetworkAccessManager *network, network manager for fetching data from FMI interface.

- load(std::map<QString, QString> parameters), downloads data from fmi by given map of parameters.
- urlModifier(std::map<QString, QString> parameters), modifies url to fetch from FMI to match user given parameters (City, start - end time and date).
- SLOT downloadComplete(), parses downloaded weather data readable for the Model. **Added in final version.**
- XMLParser(), reads data fetched from FMI to the structure_ map to be passed on to Model for rendering, **Added in final version**
- SIGNAL fetchCompleted(std::map<QString,std::map<QString,QString>> structure_, QString fetchType_), passes on the data to Model, when parsing is ready. **Added in final version.**
- loadforeCast(std::map<QString, QString> parameters), fetches weather forecast for next 24 hours. **Added in final version.**

FingridAPI, **Added in final version**

DownLoader class is used to handle connections and fetching data from internet sources to be stored by Model. **FingridAPI handles downloads from Fingrid.**

- QNetworkAccessManager* man, network manager for fetching data from Fingrid interface.

- downloadData(QString startTime,QString endTime, QString variableid) downloads data from fingrid by start-endTime and given parameter type.
- SLOT getSeriesName(QString val), matches fetched datatype with respective LineSeries element name. **Added in final version.**
- SLOT downloadFinished(QNetworkReply*), called when Fingrid download finishes. Maps fetched data for rendering and emits respective dataRenderSignal (current/start-endTime). **Added in final version.**
- SIGNAL dataIsReady(std::map<QString,double>,QString urlType), connected to renderElectricityData::render(). **Added in final version.**
- SIGNAL currentValuesReady(QString type, double value), connected to renderElectricityData::currentValuesReady(). **Added in final version.**

View (Changed from separate class to integrated in QML)

View is part of applications QML implementation and consists of slots that connect straight to signals caused by user from UI elements (buttons and checkboxes etc.). This is then used to define what graphs in each category is shown to user and how views are scaled. This was changed from separate C++ class to integrated, since it seemed separate class would become more complicated and QML has been designed to well support managing changes in UI.

- Integrated to QML structure
- ChartView.onSeriesAdded(), for each AbstractSeries shown adjust UI graph axis to show correct scale
- CheckBox.onCheckedChanged(), for each checkbox control change visibility of single AbstractSeries linked to said checkbox.
- Slider.onValueChanged(), change graph scaling between day/week scale in the view based on UI slider position
- Button.onClicked(), for initiating data fetching from FMI and Fingrid. **Definitions added in final version.**
- SIGNAL renderData(), emitted by Controller::fetchCompleted() signal and received in fetchButton to initiate data rendering request. **Added in final version.**
- SIGNAL renderForecast(), emitted by Controller::forecastCompleted() signal and received in fetchForecast Button to initiate forecast rendering request. **Added in final version.**

Storage

Storage class is used for storing user preferences regarding automatic datacollection etc. It provides interface for Controller to save single user preference parameters from UI as well fetch these values later-on. However, UI-side implementation only supports saving all parameters at once, so all settings will be overwritten each time preferences are saved. Storage uses QSettings object, for parsing values as simple <key, value> pairs on to disc, (in Windows to Windows registry, on Linux to a separate .ini file).

- _settings, QSettings object that is used to parse settings into and from Windows registry or platform equivalent.

- fetchValue(QString parameter), get currently saved value for setting.
- editValue(QString parameter, QVariant newValue), modify saved setting.

Self-evaluation

Mid-term version

Some things in original design have already been changed, for example the View class was originally only implemented as part of QML implementation but was planned to be changed to separate C++ class, however this idea was abandoned, and view implementation remains as it was: as part of QML side.

In general, the current plan is not too different from the original and mostly aims at expanding the original and defining plans more clearly. On the other hand, later implementation may need changes in some aspect such as how different things are shown to user in GUI and how user can interact with controls (for example how different graphs actually lay out on top of each other in practise / is the current categorization sensible). Changes to this might be required for the better user experience in fetching and displaying the data user wants.

On the other hand, many things have been implemented as they were planned in prototype phase and for example structure of separation into classes planned classes and responsibilities of these classes mostly remain unchanged (View class being exception).

Final version

In the final version the implementation still follows quite along what was planned earlier. However, some things have been changed from what was originally planned. There were also some aspects that turned out to have not properly been addressed in planning. For example, Qt signals for when download is ready, and rendering can be started were added in-order to make the rendering wait for current download and not render formerly fetched results. This was never addressed in detail in the original planning.

On the other hand, fetching and rendering was separated into two parts fitting fmiAPI and FingridAPI respectively, which did change some things in how the chain of fetching data and rendering it works.

As a whole the program follows the original plan and on higher level fits the Model-Controller-View as planned as well as implements lot through flexiblity of qml and Qt’s signal-slot system. Signals and slots themselves weren’t so much planned at first but implementation with Qt’s easy-to-use tools was a big part of the idea behind the project. The same was with usage of QSettings even if this was not exactly planned earlier on.