PWAnalyzer documentation

Eero Lehtinen, Samuel Helén, Roope Korkee, Saku Haikio

Brief

PWAnalyzer is an application for monitoring power consumption and weather data in a certain area. This application can be used to visually analyze past power usage and maybe see a correlation with the weather and seasons. The power data is fetched from Fingrid and the weather data from Finnish Meteorological Institute (FMI). The data can be combined and visualized with graphs.

Design process

Our group has had weekly meetings on Discord to discussing features, work responsibilities and division, and design methods. At other times we have kept in touch with Signal. We have also made a Trello board to divide the work evenly and to show the current progress of the project. To make the code look uniform for all participants, we use the clang-format. Clang-format definitions can be found at the root of the repository in the *.clang-format* file. We tried to employ the MVP design pattern for the GUI and charts. We tried to keep UI and logic separate.

Structure

MainWindow holds an instance of ChartWidget.

ChartWidget initializes its UI elements and creates a ChartPresenter for handling the chart and UI controls. Pointers to UI elements are passed to ChartPresenter in a helper struct ChartControls.

ChartPresenter listens to UI changes and based on them, and calls functions of DataLineSaver, Preference, and DataLinesModel.

DataLinesModel has setter functions that update its internal settings. Based on these settings it will call the fetch function of WebAPI if it is needed. When web responses finally arrive, it updates its internal data structures containing the resulting DataLines. Then it reports these changes back to ChartPresenter through signals.

CalcsController class can be opened from ChartWidget, and CalcsController opens CalcsWidget that visualizes calculations with a graph.

WebAPI holds a QNetworkAccessManager, that is used to connect to the internet. IProvider handle the actual request generation and response parsing.

DataLineSaver and Preference have a JsonManager for writing/reading JSON. DataLineSaver and Preference are associated with ChartPresenter: ChartPresenter uses them in its implementation for exporting/importing data/preferences, i.e., they only live in the scope of a method / some methods.

The program has a namespace called Utils. Utils contain many useful miscellaneous functions for formatting the data to different calculations of that data. This namespace's functions are used by multiple different classes.

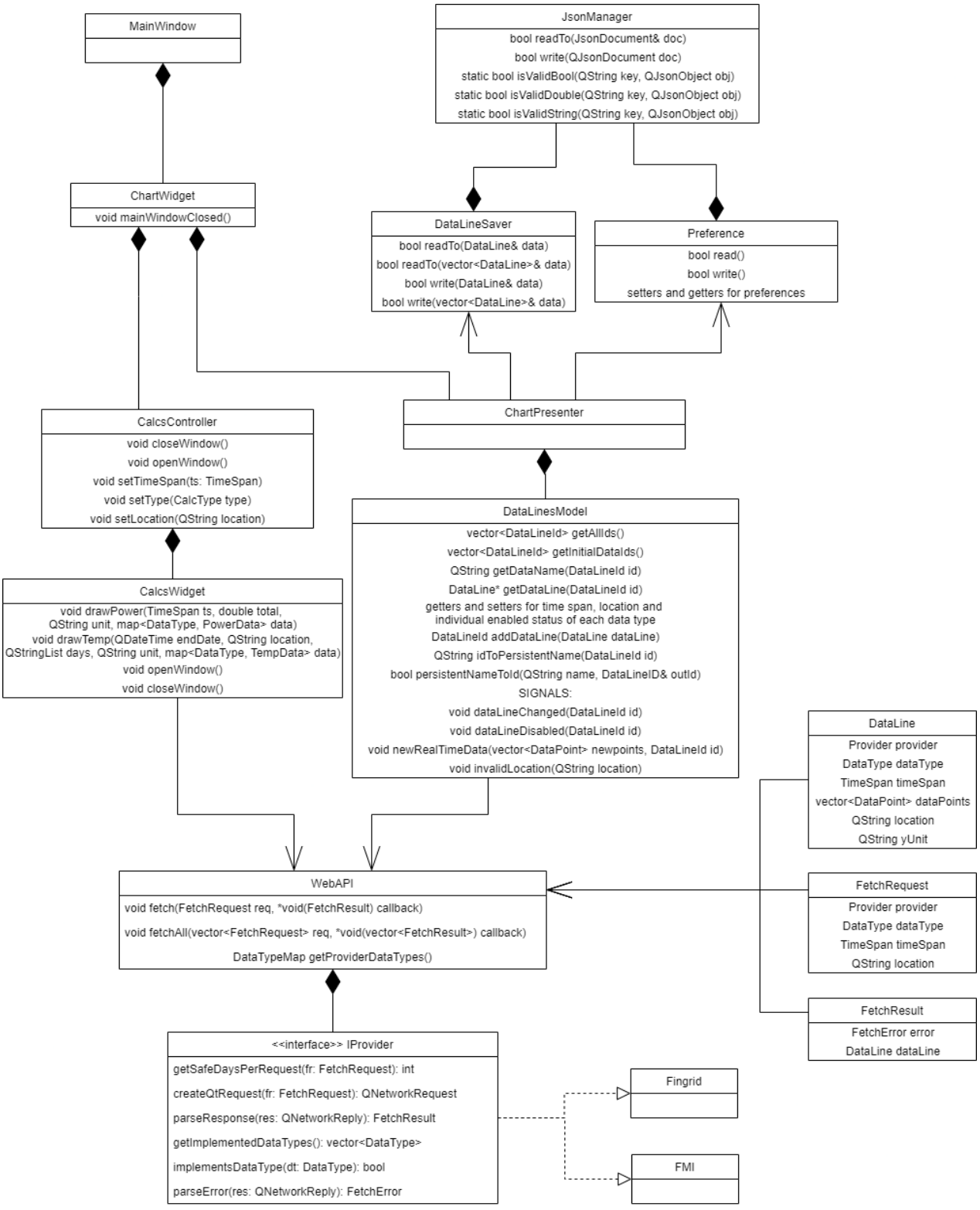
Classes

Currently, the application consists of the following classes:

- MainWindow: The main window for the program. Charts will be shown here.
- WebAPI: Handles networking required by IProviders. Uses QNetworkAccessManager.
- IProvider: Handles QNetworkRequest creation and QNetworkReply parsing for the specified provider API.
- Fingrid: Implementation of IProvider for Fingrid open data
- FMI: Implementation of IProvider for Finnish Meteorological Institute open data
- ChartWidget: Creates and holds interface controls.
- ChartPresenter: Handles UI state updates and sends the information forward to classes DataLinesModel, DataLineSaver, and Preference
- DataLinesModel: Does business logic of getting data lines from the web and maintaining them.
- CalcsController: With a user's request, it independently from the rest of the program, request data from Fingrid or FMI depending on calculation type. This class uses namespace Utils functions for necessary calculations. Other data handling is handled by CalcsController.
- CalcsWidget: Gets its data from CalcsController to open a new window and visualize requested calculation with a graph.
- JsonManager: A tool for writing JSON into a file and for reading JSON from a file into a QJsonDocument.
- Preference: Contains states the end-user can change in the GUI, enabling reading/writing them from/into a file.
- DataLineSaver: A tool for saving and loading struct DataLine.

Interfaces

The application uses an IProvider interface that is and can be used to create specific API requests. Classes FMI and Fingrid are implementations of IProvider. Other classes are not used through abstract interfaces. They can be made relatively easily if needed in the future, though. For example, an IDataLinesModel interface could be made based on the current model and implemented by some unit testing models.



Final class diagram

Implemented features

- The application fetches the data from Fingrid and FMI APIs and draws the graph.
- The application could be expanded for a greater amount of data sources (IProvider interface)
- The user can request calculations and visualizations from power/weather data.
    - Percentages of different power forms (hydro, wind, nuclear)
    - The average temperature at a certain location in a certain month
    - Average maximum and average minimum temperature at a certain location in a certain month
- The user can choose data types from a vertical checkbox list.
    - Electricity production/consumption real-time
    - Electricity production/consumption forecast for the next 24 h.
    - Wind, nuclear, and hydropower production real-time
    - Temperature and observed wind real-time and forecast.
    - Observed cloudiness in real-time.
- Selected data will be drawn on a graph.
- Multiple data types can be shown at the same time.
- Location can be set for the weather data.
- Power and weather time scales can be set from the window.
    - Very large time scales allowed (e.g., 10 years of electricity data with 3 min interval)
- The user can export the current graph view data and import it to the view.
- Save view to an image.
- The current view preferences can be saved to a file and can be loaded back to the application.
- Visualization window updates real-time
- More plotting options (line, scatter-, bar- and pie charts)

A detailed description of the features

Data Visualization

Window

The graphical window of this program is created by ChartWidget class. This class constructs windows, creates layouts, and adds graphical elements into those layouts. Elements in the window are set by three layouts, top layout, middle layout, and bottom layout. Top Layout contains buttons for Import/export operations and a combo box for changing plotting style. The middle layout is for graph view and checkboxes. The bottom layout contains user input elements like date-edit boxes and line-edit boxes for weather places.

Graph view and data drawing

In the middle of the program is the graph view area. This graphical element is called QChartView, and its function is to visualize data for the user in different kinds of graphs. QChartView is only a visualizer of QChart. QChart is a class that contains information on datapoints series and axes.

Data series for QChart are created in ChartPresenter class. When the user checks a checkbox, the wanted series is created and added to QChart. In case if the user unchecks the checkbox, the series will be removed and deleted from QChart. Datapoints are get from DataLinesModel class, which contains wanted datapoints. After adding series to the chart, the chart axes are created. The x-axis shows time, and the y-axis shows data units.

Axes are always calibrated for the data in the window. X-axis always shows range from earliest time of all data points of all series to latest time of all data points. Y-axis always shows range from minimum to maximum in the units of shown series. More than two y-axes cannot be shown at the same time. If the user chooses to see more than two data lines, y-axes will disappear, and the user can only compare lines visually.

Different plotting options

The plotting style of the graph can be selected from the combo box at the top left corner. The user can select a line graph or scatter graph. By selecting an option, the graph will be converted to the selected plotting style. The default option is a line graph.

Realtime Data

Graph updates automatically to show new data. New real-time data points are added if the new points are between “fromDate” and “toDate”. Power data points are updated every three minutes and weather is updated every 10 minutes. The new data is added only for visible data series and will automatically update the graph when the ChartPresenter’s onNewRealTimeData method is called. For forecasts, the whole graph line is updated, because forecasts may change and become more accurate over time. For this, the usual onDataLineChanged method is called.

Calculations

Power production percentages:
The user can request calculations of percentages of different power forms. On the bottom left of the application window, there is a combo box menu where the user can choose *Power%* as previously mentioned power forms calculation. With *From Date* and *To Date* the user can choose the timespan for the *Power%* calculation.
If the timespan is set to earlier than the date 27.11.2012, the beginning date will default to that value and the ending date will be set to one day after the beginning date. Wind data will begin on that date. If the timespan is entirely in the future, the calculation window closes at once and an error message will pop up on the console.
And if the date is partially in the future the results will show on the timespan that the data fetching was successful. On a successful request, a new window will pop up when the button *Get calculations* is pressed. The window has a pie chart that shows the percentages of wind-, hydro- and nuclear power productions from selected timespan and a combined power production in that timespan in megawatts.

Average temperature for a month:
The user can request average temperatures for a certain month and a certain place. This calculation type can be found in the bottom left combo box menu as the second item on the box called *Average temperature.* To use this feature correctly the user must give a valid location in Finland. With an invalid place, the calculation window will close. The user can select the right month with the “To Date” date

selector. This calculation needs only the month and the year for the time data. For the selected month, every single valid day will be fetched from the API. The dates that have no data, those that are and will be in the future will be ignored. So, the current month that has not been completed will show only the days that have passed for the selected month. For example, if today's date is 8.4.2021, the calculation window will show only this month's 8 first days and an average for that. Dates before 1.1.1970 will default to that value. Note: Most of the locations do not have data that far so the window will close as the graph would otherwise show an empty graph.

On correct data, the window will show a bar chart of the daily average temperatures for a given location. The whole month's average temperature is an average of these bar values and is shown as a vertical line. The value of the average temperature for the month can be found at the bottom of the window. The corresponding label has the value.

Average minimum and maximum temperatures for a month

The user can request a calculation of minimum and maximum average temperatures of the requested month for a given location. This feature acts similar fashion compared to *Average temperature* calculations. The fetched data is the same, but the bars and vertical lines are different. Now for each day, there are 2 bars, the left one stands for the average minimum temperature and the right one the day's maximum temperature. For these minimum and maximum values, their month averages are calculated and are shown as vertical lines. These line values can also be seen on the labels.

The program first tries to search if already fetched data can be found in the data_ map stored in the CalcsController class. The search looks at the similarities with the timespan values on temperature and power data, and location values on temperature data. If the data does not exist on the map or the values differ from the requested ones, new data will be fetched from the API.

The calculations will handle the errors that are caused by erroneous data fetched or no data fetched at all. The errors can be caused by invalid location, or the dates are set to future where there is no data yet.

The error messages are descriptive of what might have caused the problem. In general, if the calculation failed before, and the next one fails too, the message "*Data fetching has errored previously, and the graph cannot be shown. Try changing the timespan or location values!*" will show on the console.

Internal model updates

ChartPresenter forwards the states of the relevant UI settings to the DataLinesModel. The DataLinesModel holds the business logic of getting data lines from the web and maintaining them. It provides an interface with settings setters that trigger automatic data fetching when it is needed. It has signals like dataLineChanged(id) that notify state changes to ChartPresenter. Real-time updates are always on and handled automatically. The WebAPI class is used to make requests and receive responses with DataLines from the web APIs.

The WebAPI class uses QNetworkManager for low-level web request handling. WebAPI has a function for getting all supported data types for all registered providers. It also holds a fetch function, that can be called with the FetchRequest object as a parameter. The FetchRequest object has all user-controlled

information about the kind of data to fetch and where to fetch it from. The fetch function handles splitting the request if it is too large and combining them in the end. The user does not have to worry about the request size. The fetch function uses IProviders to create the requests and parse their corresponding responses. The fetch function calls the callback with an object of type FetchResponse, unifying the responses of all IProviders. The WebAPI also has a function fetchAll, which is a helper function for calling fetch multiple times in parallel.

Export/Import

In the GUI, the user can click the buttons Import Preference / Export Preference / Import Data / Export Data / Export Image. All these open a file dialog for file selection. The buttons have been created in ChartWidget, as they are part of it. ChartPresenter is responsible for its functionality, primarily using models like DataLineSaver and Preference. ChartPresenter::connectButtons connects the functionality to the buttons and is called indirectly from the constructor. If importing/exporting fails, an error dialog is shown. Data and preferences are imported/exported from/to JSON. They have different JSON types for their purposes.

Exporting/Importing Data

Data is exported/imported as DataLine JSON. Exporting exports the data that is currently visible in the figure, I.e., the data corresponding to the checked checkboxes. After a successful import operation checked checkboxes will be added for the imported data types and the data should be visible, plotted in the chart.

ChartPresenter::onExportData and onImportData are connected to the Export Data and Import Data buttons. They perform the required GUI dialogs and use the functions exportData and importData for doing the actual logic. These functions utilize DataLineSaver for writing/reading the data to/from JSON. DataLineSaver is responsible for converting and validating the data. JsonManager is used for performing the actual file operations and for common JSON validation utilities.

Exporting/Importing Preferences

Preferences are exported/imported as Preference JSON. Exporting exports the preference state that is currently set in the GUI: the states of the plot chart data checkboxes, the calculation type, the weather place, from date, to date and so on. A successful import operation will restore the state defined by the Preference JSON. Only meaningful parts of preferences are imported. For example, if the timeDateInterval of the Preference is null, it is not imported. It is not currently possible to store exported data checkbox states into a preference.

onExportPreference and onImportPreference are connected to the Export Preference and Import Preference buttons. They perform the required GUI dialogs and use the functions exportPreference and importPreference for doing the actual logic. These functions utilize Preference to write/read the data to/from JSON. Preference is responsible for converting and validating the preference. JsonManager is used to perform the actual file operations and for common JSON validation utilities.

Exporting Images

Images are exported as PNG.

onExportImage is connected to the Export Image button and performs the required GUI dialogs. The actual exporting of the image is just calling QChartView::grab() and QPixMap::save(filePath).

Design decisions

Our current design for the main chart is MVP, model-view-presenter design. Widget class instantiates UI components that combined act as a view. The presenter class listens to updates in the UI and forwards the information to the model. It also "presents" the state of the model in the UI which requires some processing. The model handles business logic and calculations. We decided to try this after the second submission. Separation of model and presenter surely resulted in cleaner code and division of responsibilities.

CalcsWidget has a simpler design. Its structure was not updated after the second submission because its scope was smaller. It does not have a separated model, instead, all logic is done in the class CalcsController. It is fine this way but separating a model would probably have been better.

In general, each class tries to have its singular purpose.

Self-evaluation

We have not had to change the components of the application much according to the current design. Only on the prototype version, CalcsWindow had to be split into two separate pieces, CalcsWidget and CalcsController to follow the similar design that ChartWidget and ChartController had in the second submission.

Our design from the second submission would have worked for this final submission just fine, but we still decided to make some architectural changes to improve it further. We decided to split the original ChartController into ChartPresenter and DataLineModel. Also, we moved the rest of the UI handling code from ChartWidget to ChartPresenter. These changes improved the division of responsibilities. Otherwise, our initial project structure stayed the same.

We of course did internal refactoring of classes to make them simpler and easier to understand. E.g., CalcsController needed some refactoring but mostly with the functions that did more than they should be doing. So, these functions were split into smaller pieces.

The implementation of exporting and importing data lines and preferences leaves something to be desired. We feel that ChartPresenter probably should not have to handle applying and extracting all preferences, nor should it create file dialogs by itself. These operations should be moved into some other class, but because of time constraints, we could not achieve this.

We managed to implement every major feature that we planned. From this viewpoint, we succeeded wonderfully.