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.

2. High level description

Our program is divided into roughly three parts: GUI, controller and backend. GUI displays information to the user, who then does an action. The GUI signals the action to the controller, which is the link in between the front- and backend. Once the backend receives the signal, it makes a response to it, signals it forward to the controller, which forwards it to the GUI. GUI then updates itself with the information given. This is the general idea in our program design.

A more in-depth look at the GUI: all of the GUI's components are under main.qml. It has components that handle the charts, the buttons and the signals of their functionality. These are TabMenu, SaveImageWindow, GraphView, SaveDataWindow, LoadDataWindow and DataPanel. The DataPanel has most of the functionality itself, and handles the signaling to the Controller. The GUI uses QCharts, which facilitates the charts drawn by the program.

The controller, named AppController, is not a very complicated one. It's task is to deliver the signals in between GUI and Backend, so that one can make changes to either one without necessarily having to change the other.

The backend is the most complicated of the three parts. Its general structure is this: the class Backend is in charge of forwarding the requests and communication to the right classes that have the actual functionalities. The first one of the functional classes is DataManager, which is in charge of loading and saving data and preferences.

The second functionality is getting the data from APIs, which is a responsibility shared in between a few classes. The APICallManager routes the API call to the right Caller class, and also checks if there is a class for the API in case. After that the call goes to either APICallerFingrid or APICallerFMI, which are both derived from a separate APICaller interface-class. Both specific APICaller classes are built based on the API, so they both know how to gather data from the API and how to parse it. Both APIcallers use OpenSSL in order to allow the networking features.

To help this process, we have a struct called dataRequest and a class called Data. DataRequest is a struct that is used to store information about the data request, so it's easy to forward to APICallManager and APICallers. Data objects are created to forward the data in a simple form all the way to the frontend. The class also enables the easy examination and editing of the different attributes of the collected data.

The data is forwarded back to the frontend with signals and slots. The class picture is very big, so it won't be included here, but you can find it in docs->class_diagram.

The libraries we use in general are Qt's and std's libraries. We use them in order to store and edit data efficiently in our whole program.

3. Boundaries and interfaces

The GUI, Graphical User Interface supplies an interface for the end-user. It takes input from the user via DataPanel and MainWindow and, in return, updates its visuals to correspond with the requested information. After an action, the MainWindow sends the signals and information to the controller. The MainWindow is the interface that handles the actions inside GUI, and also the interface that communicates with the Controller.

The Backend is an interface between different commands coming and leaving from the backend. It communicates with the Controller, the DataManager and the APICallManager. So, if the user wants to save, load or store Data, the Backend forwards the request to the DataManager, which does as asked. The DataManager then sends the specified response back to the Backend, and the Backend sends it back to the Controller.

On the other hand, if the user's input is a data request, the Backend forwards it to the APICallManager. The APICallManager then recognizes the type of the request - in our case, Fingrid or FMI, and forwards it to the appropriate APICaller. The APICaller in question finds and parses the data, sending it back to the APICallManager, which in turn sends it back to Backend, and on to the Controller. This makes the APICallManager the interface for communicating about API calls in spesific, and does so with both of the API callers and Backend.

Finally, there's the controller, that acts as the interface connecting front- and backend. Either one of the data types the Controller gets, it sends it forward to the MainWindow, which in turn updates the GraphView, DataPanel and itself, thus giving the User the visual response.

Again, we have a visual representation of these boundaries and interfaces, but it's too large to fit in the document. You can find it in docs->interfaces.

4. Components and responsibilities

Controller

AppController

AppController-class handles routing the signals and slots of GUI and backend to connect them.

- The init()-method initializes the application by connecting signals of GUI to the backend and vice versa.

Backend

Backend

Backend-class is the basis of backend application logic and routes the communication between frontend and backend.

- The fetchNewData()-slot creates a DataRequest struct based on the parameters given from GUI and forwards it to APICallManager.
- saveData(), loadData(), savePreferences() and loadPreferences() -slots all call file handling operations of DataManager.
- requestPrefData()-slot parses a saved request preference into a DataRequest struct and forwards the created DataRequest to APICallManager.
- forwardData()-slot converts data received from an API request to QML compatible form and forwards it to GUI.
- sendError()-slot forwards backend error messages to GUI.
- removeData()-slot calls DataManager to remove specified stored data.
- loadAPIConfig()-method reads and parses API keys from apiconfig.txt file.

APICallManager

APICallManager-class handles registering APICaller-classes and routes DataRequest objects to APICaller matching the data type of the request.

- Register()-method calls CreateAPICaller to register a new APICaller class for APICallManager to use. Also stores given API name and key into a data structure, which is used to route a DataRequest to the APICaller matching the request's data type.
- CreateAPICaller()-method is a factory function that creates an APICaller object.
- fetchData()-method forwards a DataRequest to an APICaller matching the request's data type.
- forwardData()-slot forwards Data object created from an API reply to backend.

APICaller

APICaller is an interface class to derive API-specific APICaller-classes from.

- pure virtual fetchData()-method creates an API request based on the DataRequest struct given as parameter. A finished API reply connects to parse()-slot.
- pure virtual parse()-slot parses data from an XML or JSON format API reply and stores the data into a created Data object.
- virtual error()-slot forwards errors that occur during an API request.
- pure virtual formURL()-slot forms an URL for the API request based on the DataRequest given as parameter.

APICallerFingrid

APICallerFingrid is an APICaller-derived class which handles fetching and parsing of Fingrid's data.

- Implements the pure virtual functions of APICaller.
- dataTypes()-method returns the datatypes that the APICallerFingrid can handle.
- Create()-method creates a new APICallerFingrid-object.
- calculatePercentages()-method calculates the percentages of nuclear, wind and hydro power productions.
- createNetworkRequest()-method creates a network request based on the parameters given.

APICallerFMI

APICallerFMI is an APICaller-derived class which handles fetching and parsing of Finnish Meteorological Institute's data.

- Implements the pure virtual functions of APICaller.
- dataTypes()-method returns the datatypes that the APICallerFMI can handle.
- Create()-method creates a new APICallerFMI-object.
- splitDataRequest()-method splits a DataRequest with a timeframe too long for a single request into multiple parts.
- calculateAverage()-method calculates average temperature for the given timeframe.

DataManager

DataManager-class stores Data objects and handles saving/loading data to/from a file.

- addData()-slot stores the data given to the DataManager
- saveDataToFile()-slot saves the data object to a file on the user's computer
- loadDataFromFile()-slot loads the data object from a file
- savePrefToFile() and loadPrefToFile() -slots save and load the user's data preferences
- toJSONPref()-slot creates a preference file out of all of the saved datas in DataManager
- removeData()-slot removes a data object from the DataManager

Data

Data-class stores data parsed from an API reply. Each API reply creates a new instance of the class for easy data examination and editing.

- getID(), getDatatype(), getUnit(), getLocation(), getDataValues() and getDataSource() -methods all give a specific element of the data to the caller
- fromJSON()-method creates a data object from a QjsonObject
- toJSON()-method creates a QjsonObject from a data object
- setDataType(), setUnit(), setLocation(), setDataValues() and setDataSource() -methods all set a specific element of the data to be a certain value
- addDataValues()-method appends new values to the data's existing values
- print()-method prints the member variable values of the data object

DataRequest

DataRequest-struct defines parameters for an API call. It has the values an API call requires: datatype, dataSource, startTime, endTime and location.

GUI

main.qml

Holds the main application window. This is where all the different signals and slots from the different GUI components are connected and where signals from the backend are routed to

the different GUI components. In other words, the main application window component glues the GUI logic together.

requestData(dataProperties) -function signals the data properties that were received from a data panel and sets a “Getting data” message to in the graphview for the duration of the data request.

addData(data) -function adds data to the corresponding data panel and the graphview. After that, it hides all messages in the graphview. If a datapanel corresponding to the data type of the data is not available, an error message is shown.

removeData(id) -function removes data with given id from the corresponding data panel and graphview.

showErrorMessage(errorMessage) displays a pop-up message that shows the error message.

DataPanel

The DataPanel QML type handles all the inputs from the user that are used as a parameter for an API call. One DataPanel holds the name of the API (data source) , a ListModel for datatypes and locations as well as datetime inputs. These parameters are then combined in an object and signaled to the backend with a push of a button. The DataPanel also holds a list of the datas that have been added. The datas can be selected from the list and removed. The removal of a data also signals the backend to remove the data from the backend side. The DataPanel also allows renaming a data by double clicking the data on the list.

validDateTimeInput(date,field) -function checks if the date input in the given field is valid. If not, it displays it in red and disables Add data -button.

dataTypeExists(model, datatype) -function checks if the given datatype exists in the datapanel.

requestDataSave(filename, url) -function takes the filename, url and currently active data’s id and signals them forward.

requestDataLoad(filepath) -function signals the filepath forward.

addData(data) -function first checks if the datapanel’s datatypes model has the type of data provided and if it does, it adds the data to the datapanel.

GraphView

The GraphView QML type holds the chart which displays a visualization of the added datas. The visualization is updated when a new data has been added and when a data is removed. The GraphView also has a toolbar assigned to it, which allows modifying the axis, zooming and saving of the chart as an image. The GraphView also handles various mouse events. The GraphView can be dragged around with the left mouse button and zoomed with the

scroll wheel. By dragging with the right mouse key, the user can zoom into a rectangular area in the chart.

saveChartImage() -function saves the chart image to a path given in the parameter.

addSeries() -function creates a lineseries from the data provided as a parameter and adds it to the chart.

removeSeries() -function removes the series that corresponds to the id given as a parameter.

changeSeriesName() -function changes the name of the series with the given id

showMessage() -function shows a message that is given as a parameter on the chart

hideMessage() -function hides the message on the chart

LoadDataWindow

The LoadDataWindow QML type opens up a file dialog that the user can use to select a file for loading. This component is used when loading data files and preference files. When a file is selected, the LoadDataWindow signals the file path.

SaveDataWindow

The SaveDataWindow QML type opens up a dialog window that the user can use to define a file name and select a path where the file will be saved into. Once the user has accepted the filename and path, they are signaled forward.

TabMenu

The Tabmenu QML type holds the different tabs which can be selected in the GUI. The tabs are used to switch between different data panels.

5. Design decisions

We have implemented our program using the MVC architecture, since it's an easy way to split the different responsibilities within the program. Our GUI represents the view in this program and it is used to display the data retrieved from the model to the user. Users also make requests to model via the view. When a request is made, it is handled by the AppController class, which acts as a controller of our program. When the request has been handled by the model, the controller signals this back to the view. Our model is represented by the backend class, which controls the data stored in the program.

In our program's backend we have used the following SOLID principles: single responsibility principle (SRP), open closed principle (OCP) and dependency inversion principle (DIP). SRP is used in our datamanager and API-classes. Datamanager only handles loading and saving data and APICaller is used to fetch and parse data from API which is then forwarded to other

classes. OCP is especially used in our API-classes. We designed them in a way that allows us to add support for different API's in the future. All you have to do is to derive a new class for the new api from APICaller class. We also used DIP when we implemented the APICallManager and APICaller class since APICallManager doesn't have to know how APICaller has been implemented for a specific API to work properly.

6. Self-evaluation

In our final version of the program, we decided to make one big change - remove the separate View C++ class. This was done mainly because of how hard it was for us to implement some of the more complex features, while editing QML with C++. After some consideration, we decided to remove it and do all GUI features right in only QML.

We don't think the decision was made because the original design was bad, but rather because none of us could've anticipated for it to be so difficult to edit GUI from the C++ View.

We did think about removing the controller as well because of this, but decided to keep it since it worked as a good connector between the front- and backend. It has been changed a lot due to the deletion of the view C++ class.

Otherwise our design follows our mid-term submissions design quite well. As anticipated, we've had to add some functions when implementing some of the features, but the base design is the same. Backend, DataModel, APICallManager and the base API class, from which the Fingrid and FMI are designed, are all as designed then. We didn't even have to add any classes as we anticipated, because the DataModel didn't become too crowded and the features all fit neatly in there.

If we think about our original design made long ago, compared to that, we have added the API-parent class in order to increase the extensibility of the program, and the Backend-class, which organizes the actions of the backend. In general, I think our design even back then was working, and could've been implemented as is. After the changes though, it is made with even better design decisions and principles, and worked well for us in practice.