SOFTWARE DESIGN PROJECT DOCUMENTATION

Group: MNST
Duy Anh Vu H294381
Nghia Duc Hong H292119
The Anh Nguyen H292126
Thuy Phuong Nhi Tran H291937

1. Introduction

The purpose of the project is to design and implement a piece of software application for monitoring electricity consumption as well as weather forecast. The aim is to find the relationship between these two factors, how one affects another at some aspects.

2. Application structure

The application is separated by 2 main components:

- Server side (backend) provide formatted APIs to provide the raw data but in a predictable JSON object that the client size can use the APIs to fetch necessary it needs.
- Client side (frontend) would call the APIs services from the server and make the raw data to represent the desirable graph based on user input.

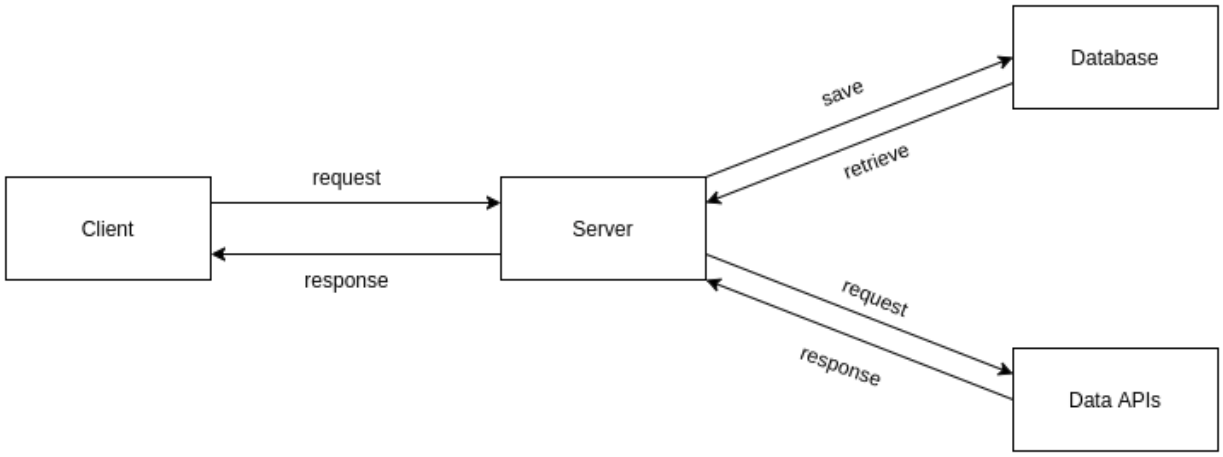

Figure 1: Application structure

2.1. Server

- The server is now implemented with fully functional requirements which contain most of relevant components. The main server contains types of components: A router with different endpoints, different middlewares (request body extractor), the controllers to fetch the data, and a controller to save the data to a basic database.
- The data flow is made to be as simple as possible. When the server receives a request, first, the request goes through the middlewares to extract the right types of data needed to be fetched. These types are used to determine where the controller should call for data (FMI or FinGrid). After this, the request would be served with endpoint controllers. By using extracted types and corresponding data sources, start time and end time in the request body, the controller can fetch data that the client is requesting. After fetching the data, the responses are merged into one response to be sent to the client. The merge is for the convenience in representing data on the client side.

- If the request is about saving a set of data and user visualization choices then the server would serve this request by using a “save controller”, handling the data sent from the client and saving it to the database.
- The responsibility of each component in the backend is generally divided and has a unique responsibility.

2.2. Client

The client side contains:

- Three different tabs for different presenting purposes: Electricity, Weather, and Combination of Electricity and Weather Data.
- Electricity, Weather and Combine tabs:
    - The date and time picker that allows user to choose specific time (start- and end-time) for the query data. If either start time or end time is not chosen, it will be set automatically to current time.
    - A list of checkboxes for user to choose which data they prefer to display. After dates and checkboxes are picked correctly, the user interface will response with a multiple-line graph presenting corresponding data. If user selects the checkboxes that require the graph to present more than two units, the alert bar would inform them, and the graph do not update until the user chooses correct selections again.
    - A multi-line graph to display the data required by the users. The users are able to hover one any line, then a tooltip appears to give users more detail about the data at specific time. Additionally, the graph can be zoom and pitch.
    - Save button allows user to save current selections or data in chosen times. After clicking the Save button, there will be a pop-up diagram which asks user a preference name. If no name is given, the reference name will be automatically set to the time when the action occurs. The user can add description for the data that the user wants to save, but this is optional. When the user gives the name that is already exist, the user will be asked if the user want to overwrite or not. The name of a preference is unique across the system.
    - Dropdown menu contains saved preferences that user can choose to present the data. When user chooses an item, the saved selections and graph will be restored. In addition, user can delete their reference by clicking the delete button in the dropdown menu. When displaying the saved data, it is possible for user to select more selections from the checklist to fetch new data while keeping data from the saved preference. However, changing time or location will result in a completely fresh data, which means the saved data will be replaced.
    - User can hover to the list item in the dropdown menu, there will be the tooltip appears to display the name and the description given when user save data.
    - Saving visualization button that allows user to save graph as image (.png). User can choose which directory in their device to save the image.
    - Update data real time toggle that will update the graph automatically every three minutes.

- Electricity tab:
    - Show percentage button is for user to have a visualization (pie chart) of percentage of different power form (hydro, wind, nuclear, and others)
    - Dropdown menu that allows user for picking preferable number of graphs.
- Weather tab:
    - Button for showing the visualization for the average, minimum or maximum temperature in a particular month.
    - Dropdown menu that allows user to pick preferable number of graphs.
    - Dropdown menu where user can choose the location for data display.
- Combination of Electricity and Weather Data tab:
    - There will be option for user to choose which data is going to be presented. There will be all options from Electricity and Weather tab. The restriction on maximum 2 units is still applied in this tab.
    - Dropdown menu where user can choose the location for data display.
    - We only allow one graph in this tab to keep the other two tabs meaningful.

3. Components

3.1. Server

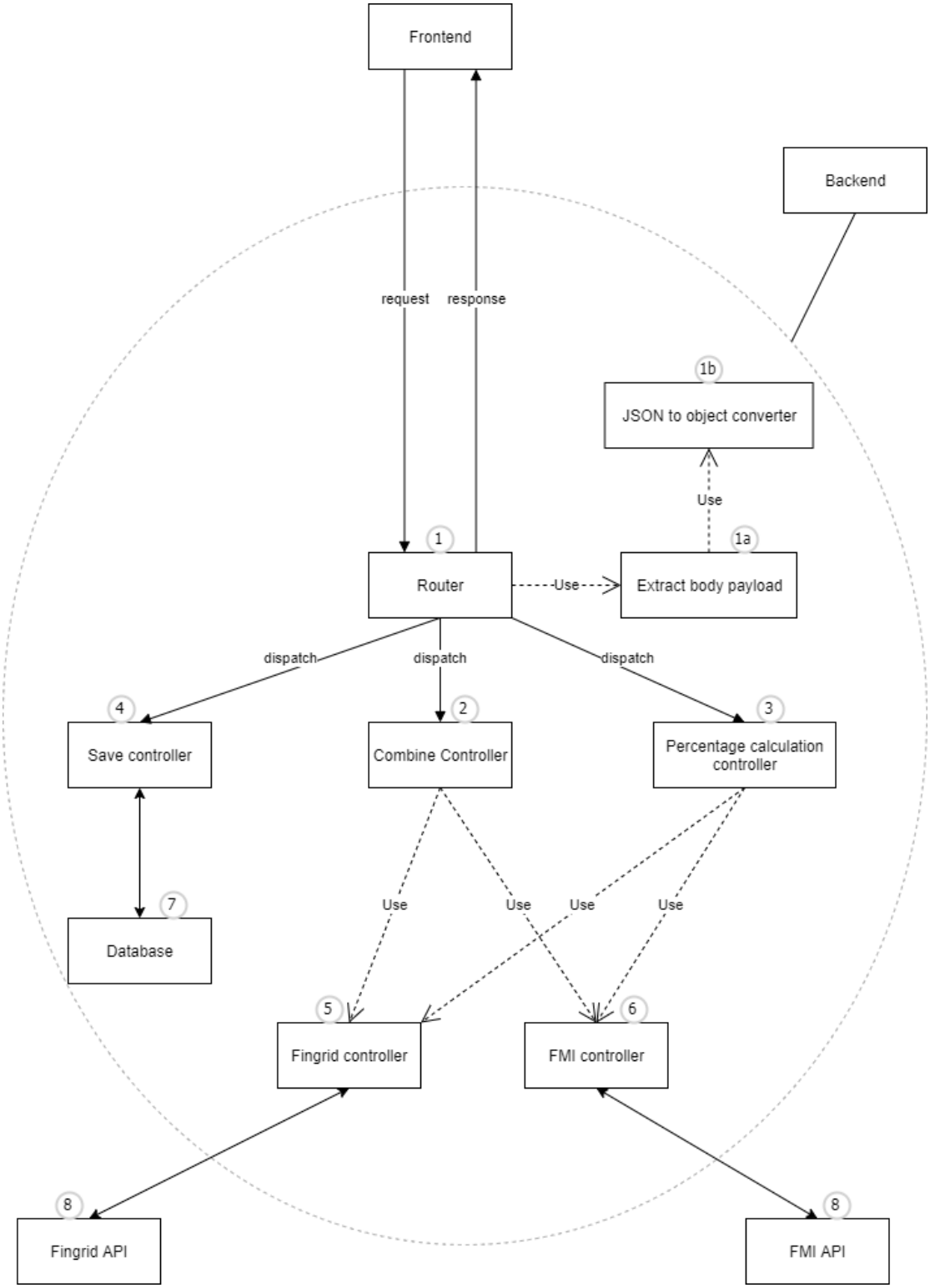

Figure 2: UML for server.

- The router (1): the responsibility of the router is routing the requests to different controller. This is done by ExpressJS.
- The middlewares (1a and 1b): The Body extractor would extract the types of data that needs to be fetched. For instance, in the field “types”, the client wants to have types = [“electricity-consumption”, “wind-production”], then it would be extracted to be = [193, 75], which corresponds to variable IDs provided by FinGrid.
- The controllers:
    - (5) FinGrid controller: fetch data provided by FinGrid and calculate power production average.
    - (6) FMI controller: fetch data provided by FMI and calculate average temperatures.
    - (2) Combine controller: The data would be mixed between different types of data between providers, the backend needs to route those mixed types to correct controllers and later aggregate the response from controllers to a single result that the client side wants. Currently, this controller divides the calls to two controllers, FinGrid and FMI, and combine the results together to send to client-side.
    - (4) Save controller: This controller will communicate with the (online) database and handle saving-related operations.

- Each controller will use smaller components:

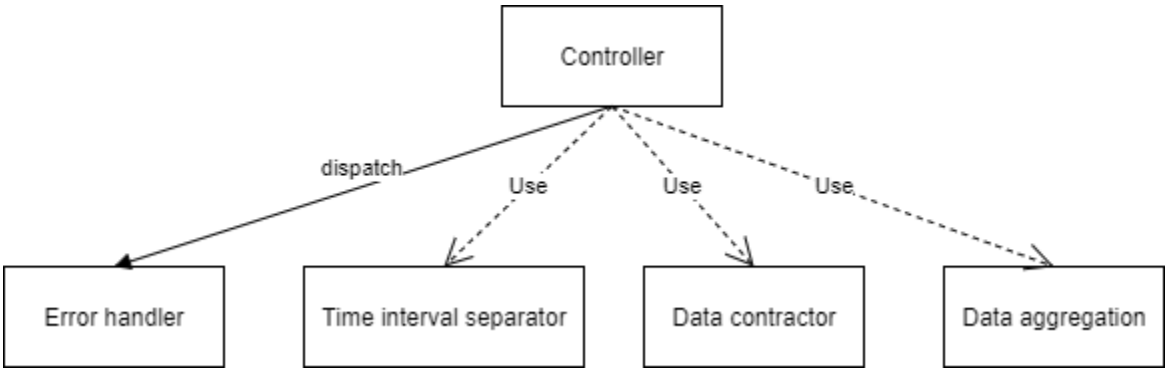

Figure 3: Usage of small components in Controller

    - Error handler: send error response when the request is not correctly formatted.
    - Time interval separator: split the interval between start and end time into smaller batches for the data provider to support. This is used when fetching from a long-time interval.
    - Data contractor: contracting data when the data is fetched from long interval or we want to calculate the percentage/average of data.
    - Data aggregation: merging response from different types before sending to client.
- The model: The model only stores the data, and the interface is provided by a framework of MongoDB so that the controllers can use this interface to interact with the model.

3.2. Client

For frontend part we use material-UI for implementing components and ReactJS to render them to the website. Our components include three main elements: Electricity tab, Weather tab and Combine tab. These main parts are constructed using small reusable components.

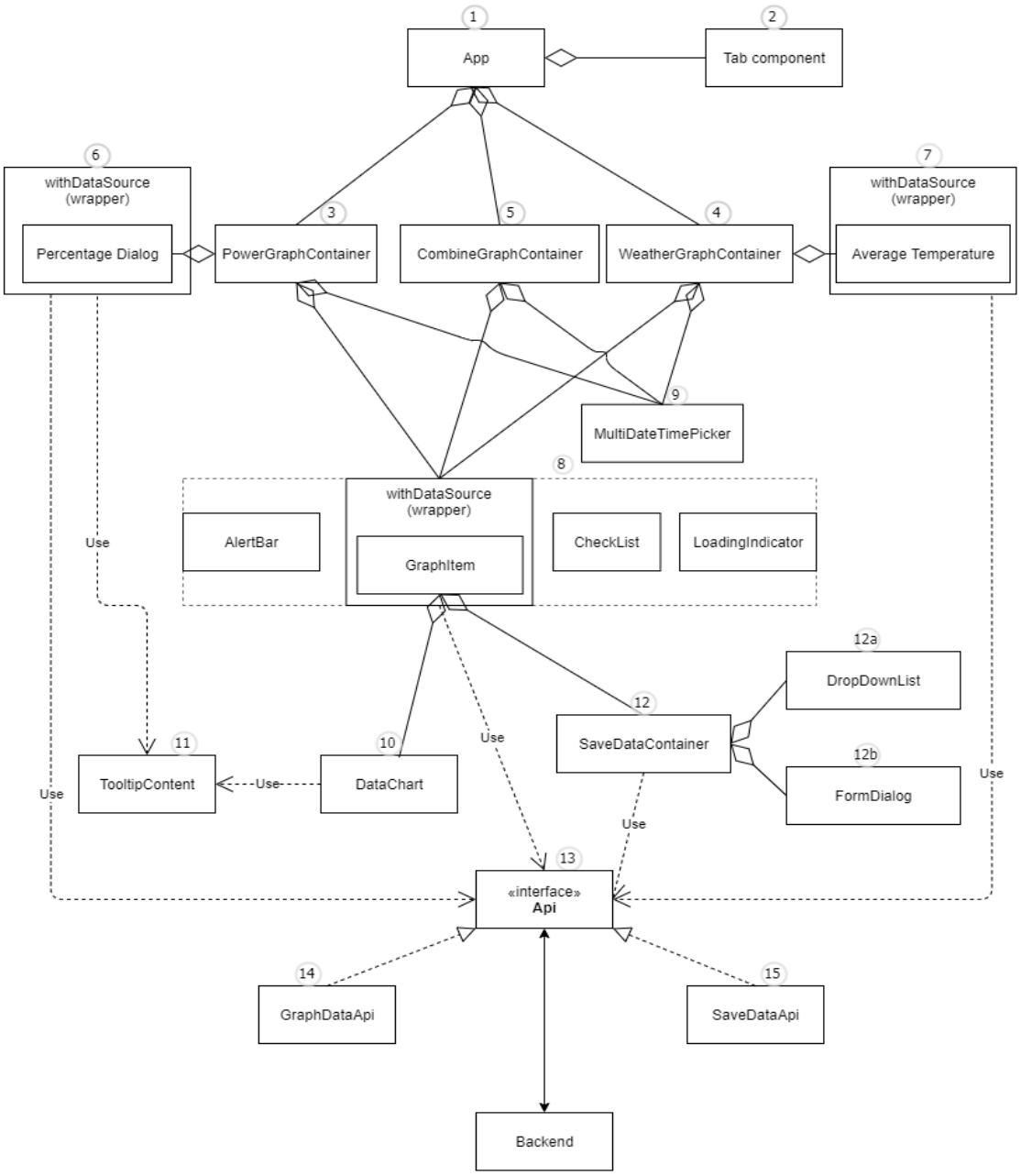

Figure 4: UML for client side

(1) App: this component is the main component of the application. App component is a container that render all other components.

(2) Tab component: this component allows users to switch between graphs based on their content group. Three tabs implemented are “Power tab”, “Weather tab”, and “Combine tab”.

(3) PowerGraphContainer: this component contains one MultiDateTimePicker, one drop-down menu that allow user to choose the number of graphs at once, one button that show calculations as a dialog (Percentage Dialog (6)), one switch that allow user to update data every 3 minutes and one SaveDataContainer (12).

(4) WeatherGraphContainer: this component contains one MultiDateTimePicker, one drop-down menu that allow user to choose the number of graphs at once, one drop-down menu that allow user to choose the location, one button that show additional weather data (Average Temperature (7)), one switch that allow user to update data every 3 minutes and one SaveDataContainer (12).

(5) CombineGraphContainer: this component contains one MultiDateTimePicker (9), one drop-down menu that allow user to choose the location, one switch that allow user to update data every 3 minutes and one SaveDataContainer (12) and one GraphItem (8).

(6) PercentageDialog: this component takes the time interval chosen from PowerGraphContainer (3) and calculate the proportion of each type of power production and display them as a pie chart. This component use TooltipContent (11) to display statistic when hover mouse on the pie chart.

(7) AverageTemperature: this component contains one time picker that allow user to choose the month when the additional statistics (average temperature, average maximum and minimum temperature) are needed and display them.

(8) GraphItem: this component contains one Checklist that allow user to choose the data types from a list, one AlertBar that will notify if the time interval is too long (more than 1 year) or more than 2 units are selected, one LoadingIndicator that display a circular progress while the data are fetched from backend and one DataChart (10). This component receives the time interval from the PowerGraphContainer (3) or WeatherGraphContainer (4) and send request to backend based on this information and on selections of user. The response from backend will be parsed to DataChart to show the data got from the API.

(9) MultiDateTimePicker: this component allow user to choose the time interval for needed information.

(10) DataChart: this component receives data fetched from backend and display data as line graphs, each line shows information on one data type. The users can download the current graph to an image file to their devices.

(11) TooltipContent: this component gives further information about the graph when the user hover over a line in a line graph or pie chart.

(12) SaveDataContainer: this component includes one FormDialog and one DropdownMenu.

> (a) FormDialog: this component shows a dialog that allows user to choose to save current data or selections with unique title. If the title is left blank, the option will be saved with current time as its title.
>
> (b) DropdownMenu: this component shows the titles that users saved. Selecting an option will update the GraphItem. Users can delete their saved data by clicking the button after the title.

(13) Api: this component provides methods for components to interact with the backend.

4. Design idea

- We realized that the data flow is quite simple in this application, that is why we want to keep the server side to be lightweight as much as possible. As this reason, we decided to make the server provides enough the raw data but in formatted way, sending it to the client side. It would be fast for the server to work in this way, and it would not take lots of resources or any complicated design relating to keep the track on the client side or drawing the graph by the server and then send it to the client side. On the client side, with only the fetched data, we can draw the graph separated from the server, as we see that it is not necessary to have any relationships between model and the view. With this design, we can also separate the work between team member working on different components at the same time without interfering each other. In the future, the documentation of APIs and the formatted data provided by the server would make the work even more separable.
- Decision of choosing programming languages:
    - The backend side relates to fetching data from external sources and obviously JavaScript provides a lot of ready-made library from open source to perform fetching and processing data with xml and json format. The Qt library is useful in C++ but we noticed that we will have to learn to use the library and this process is not really confident with the current documentation of the Qt library.
    - The frontend side relates to visualize the data. With modern web tool platform, we are using JavaScript to visualize the data with the data provide by the backend side. The way of manipulating the graph with data is the same with the QML provided by Qt but we are more familiar with web technologies, so we decided to use web platform.
- Decision of choosing web technologies:
    - The communication of backend and frontend mostly is about transferring the data between two ends. We chose HTTP for resource transferring as it is simple and the way the resources is sent is compact and convenient. The process of transferring the data is stateless, which means there is no state is kept between two ends, and the information is exchange when the frontend makes a request, and the backend sends a response. This way, we can easily exchange the data between ends without increasing the complexity and more capable to implement the software separately without knowing a lot of dependencies.
    - The APIs provided by HTTP contains various methods which are handy for uses in both ends. The frontend can call the API through the web address (host) with the given predefined-formatted request. The backend will be implemented to fulfill the API it has to provide so the data can exchange between ends correctly. This mechanism is similar to using interface in OOP.
    - The framework ReactJS recently becomes dominant in web technologies, as it is robust in visualizing and managing the software GUI. The beneficial of this framework is that the GUI will be divided into components which has their own properties and state. The properties are passed within components to share the information. With this way of architecture, we can take advantage of reusability of many components as they are meant to follow the Single Responsibility Principle as much as possible. Most importantly, the framework provides a mechanism that the components will be updated themselves immediately when a property/state changes. The ready-made components provided by open sources are easy to use with clear documentation and are proved to be good by a big community.

- The framework ExpressJS gives us an intuitive and compact way in routing the requests from the client side. The framework provides the interface to define how to response those requests from clients with given HTTP method with a call-back function. Each API provided will be handled by a callback function and this function will send a response corresponding to the request information. The framework also provides some “middlewares”, which are the functions extracting the relevant information from request. The client does not know how the information is processed by the backend. They only know that they are querying for data from the backend and the backend will response with result. This will increase the modularity of the applications.
- The model is using a framework called MongoDB. This provides a way to move the model to online stream with a URI. The benefit of moving the model online is that our model will not be reset when the application shutdowns. The model runs independently with the application, and the application only provides model schema to run on.

5. Boundaries and interfaces (internal)

- We specifically have the router, so we do not need to have a common interface of controller to provide the services for client side, but we are making the controllers in a way that they are very similar to subclasses implementing a common interface in a non-OOP programming language. More information on the *Design Patterns* section.
- The save controller does not go through any middleware and provides different operations as it interacts with the database differently to other controllers. We can fetch all the saved graph, add, modify, and delete the graphs by using the model interface.

Information flow:

- Case the user wants to visualize the data:
  - The client would make a request and specify which types of data he wants to fetch with the parameters in the body: start time, end time, descriptor (the object containing data types and locations). Then the router will route to the exact end point to serve this request and then send the data to the client. After the data is sent, the logic under client implementation will visualize the data as graphs.
- Case the user wants to save the graph with data/preference:
  - The client after sending the parameters to have a glance the graph he will save, the client would send a graph object back to the backend side, the backend side will use the save controller and save the data to the database for future use and the graph and preferences can be pasted in the future fetching.
- Case the user wants to save the current graph as image:
  - The client side would use the current representation of the graph and generate an image that the user can save it to his computer. This does not relate to backend side because it is bulky to have the backend side generating the image and then send it back to client. (No extra dependencies)
  - The way we update the data with the real time is that we would make a request every 3 minutes. It is better to do this way because if we do this in backend side, then the backend still makes a request every 3 minutes to check whether if the data has been changed. Doing this way, the feature can be implemented in compact way and no adding any dependencies.

6. Restrictions, limitations, and notes on functional requirements

- For graphs, user can select as many series from the checkboxes as they wish, as long as those selections lie inside maximum 2 distinct units. For instance, it is not possible to choose Electricity Consumption, Temperature, and Observed Wind as the units are MWh/h, degree C, and m/s.
- All data values are rounded to nearest integer for the sake of nice visualization. This is easy to change if users want.
- FinGrid provider supports query forecast history, but FMI does not. Therefore, saving data points into preference is needed.
- For percentage visualization of different power forms, only time selections are considered. Selections on checkboxes do not affect the result.
- For average temperature visualization in Weather tab, the latest month the user can query is the previous full month of current date. For instance, it is mid-April 2021 now, so the user can select at most until March 2021. This is done because the data for April 2021 has not existed fully yet.
- We decided to limit the interval between start time and end time to 365 days. Of course, this can be extended if needed. The backend can support any interval length. We put a limit here as we believe there is no use case for querying more than 1 year time. In addition, querying a very long period would be quite slow.
- Whenever the user queries for data, there would be a loading indicator to inform that the data is fetching.
- Long time interval will be separated into smaller pieces so that the fetching to external provider can happen concurrently. Another reason is that real-time series from FMI do not support time interval more than 168 hours. The result from concurrent fetches will be merged later on.
- The number of data points lies between 250 and 500. It is not meaningful to show a huge number of data points into the graph, and this also burdens the rendering. We contract the data by taking average of consecutive data points. This explains why sometimes the time of consecutive data points seem to be a bit weird.
- Sometimes switching between preferences would cause the system to fetch new data again and result in empty graph. This happens to Weather and Combine tabs as FMI provider do not support forecast in the past.
- We currently have not supported same series at different location, e.g. two lines representing temperature in Helsinki and Tampere. Therefore changing location after selecting a saved preference would result in completely new data.
- If auto-refresh toggle is on, user might see a loading indicator shows up even they do not do anything. This is expected as the auto-refresh refresh the data after every three minutes. For auto-refresh, the end time is taken as current time, so the end time the user selects is ignored.
- If the error “Maximum update depth” error jumps up, it is probably because of the Grid component from the UI library. We have done our best to make this disappear, but it might happen to some particular screen sizes. We recommend using the application with full expanded browser.

7. Design Principle

- ***Backend***
  - We are applying Single Responsibility Principle to all components in the backend. Each controller is responsible for one root endpoint in the request URL; for example, the root endpoint for FMI is /api/fmi. One controller handles two main child endpoints, one for getting normal data and one for getting average data. Each child endpoint is handled by a separate function. These two main functions use a "private" function to fetch data from external data provider (FMI or Fingrid). This private function expects to receive as argument a start time, an end time, and a descriptor which contains specific information about selections from different external data source.
  - Chain of Responsibility principle is applied for the backend. A request from client would go through several services, each with different responsibility, before returns to client as result.
  - We do use something similar to Façade with the combine controller. Combine controller is in fact an intermediate to hide the FMI controller and FinGrid controller from client. We are currently success in preventing this Façade to be a god class by making the interface of FMI and FinGrid controllers similar to each other. Therefore, there is no need for special handling case for each controller in the Façade.

- ***Frontend***
  - Most of the frontend follows SRP and DRY (Do not Repeat Yourself) principles.
  - We are separating UI component into small pieces as much as possible to reduce the responsibility of the component as well as enhance reusability across the system. We would use parameters to customize the components, and of course, the number of parameters is kept in a reasonable amount. All UI components are related to each other through composition or aggregation.
  - This way of diving components is much like creating templates. However, we do not always try to create everything as templates in the UI as they are highly difficult to customize. This is the case with Power, Weather, and Combine graph container. Although these three containers share many common code, we decide to separate them in order for further customization. For instance, we might want to display the temperature in different form like using sun and cloud icon.
  - We have a decorator in the UI. For JavaScript, we use a high-order component to wrap around another component so that the wrapper can provide additional, abstract fetching function for child component. This is also similar to Strategy pattern in some senses as well.
  - The states in the UI are observed through an observer. Whenever one changes, a function will be invoked to handle the state change.

8. Self-evaluation

- The architecture of backend does not change since mid-term. For frontend, we have separated the graph containers into separate components and we also introduce just one or two new components into the architecture. We also split out the checkboxes and the graph into a component so that we can clone that one for multi-graph feature. Thanks to the composition nature of ReactJS, we can flexibly style and restructure the components in the UI without any major changes.
- The design helps us a lot in finishing the assignment. For instance, when we come to implement the long interval fetching, we do not have to significantly change any existing functionalities or structures of controllers. We only need to add a time interval splitter before fetching external API and combine the results together afterward. We also added a contract data function before sending to client to support the long-time interval fetch feature. All of our new functions are very easy to integrate into existing code because they all follow the Chain of Responsibility principle that we set in advanced for the whole backend. This is definitely a sign of good scalability.
- The design was done to help integrate new data source as easily as possible. We advocate generic solution instead of hardcoding ones. Depends on the restrictions and structures of new data source API, however, it is hard to say if a design can support integrating new data sources without major changes. Even FMI and FinGrid APIs are quite significantly different. Adapters might be needed in order to integrate new data source into existing system.
- In general, the big picture we had initially remain when we finish developing the program despite some minor detailed parts have changed. The architecture and principles are kept as they used to be. We are confident to say that we have succeeded to achieve what we planned.