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

Design Document

Team 17

Olli Koskelainen
Murad Tanmoy
Jun Jung
Vladimir Vechtomov

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

Technologies:

- Language: C#
- Framework: WPF (Windows Presentation Foundation)
- Libraries:
    - System.Windows.Control.DataVisualization
    - Xceed.Wpf.Toolkit

Design Pattern: MVVM (Model-View-ViewModel)

> MVVM was chosen for the application architectural pattern of the group assignment. This is because this pattern allows us to separate the graphical user interface and the business logic. It brings benefits to develop and maintain each module easier without interrupting other modules.

Functionalities:

- Display electricity data on graphs or plots
    - Adjust the parameters of the visualisation (timeline, power source, etc.)
    - Combine electricity data in one view.
    - Calculate and visualise of percentages of different power forms.
- Display weather data on graphs or plots
    - Adjust the parameters of the visualisation (timeline, location, selected weather information)
    - Calculate and visualise average temperatures at certain location in certain month.
    - Calculate and visualise average maximum and average minimum temperatures at certain location in certain month.
- Display combined data of weather and electricity into one window.
- Initiate data collection.
- Save certain data sets and produce visualisation.
- Save preferences for producing visualisations (parameters)

Modules:

- UI
    - Electricity consumption (load)
        - 24hr forecast
    - Electricity production (generation)
        - 24hr forecast (hourly energy)
        - Wind power forecast
        - Nuclear power
        - Hydro power
    - Visualisations of these stats and percentages between them
        - https://www.c-sharpcorner.com/UploadFile/mahesh/charting-in-wpf/
- Core

-
  - All business logic for handling program functionality and data
    - Combining data
- Data
  - Getting data from HTTP servers
  - Interfaces for reading this data in a specific format

Source APIs:

- https://data.fingrid.fi/open-data-api/
- https://data.fingrid.fi/en/dataset
- https://data.fingrid.fi/en/dataset?groups=load-and-generation
- https://data.fingrid.fi/en/dataset?groups=state-of-power-system
- https://en.ilmatieteenlaitos.fi/open-data-manual

Tasks:

- Create a module to pull information from source APIs
  - Most likely using HTTPclient
- Create a module to display information in the UI
  - Need to determine the format of the information

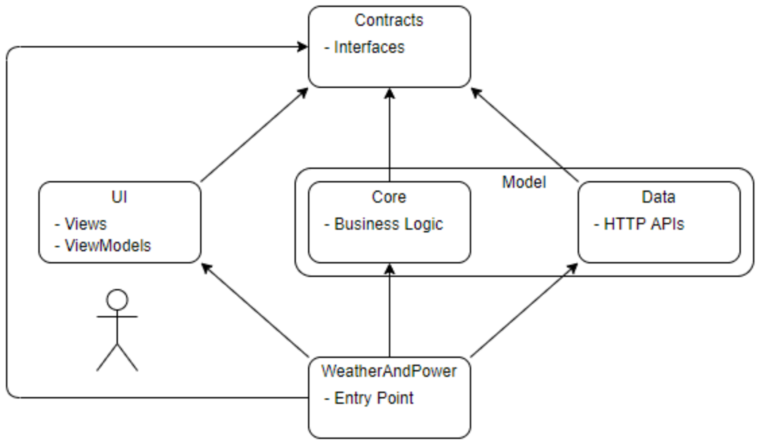
Interfaces:

- Weather Data
  - Temperature
  - Observations
    - Wind
    - Cloudiness
  - Predictions
    - Wind
    - Temperature
  - Supplied by Data module
- Power Data
  - Consumption
  - Production
  - Source (type of power, nuclear, hydro, etc)
  - Supplied by Data module
- Grapher
  - Implemented by UI module. Allows for drawing graphs and other visualizations.

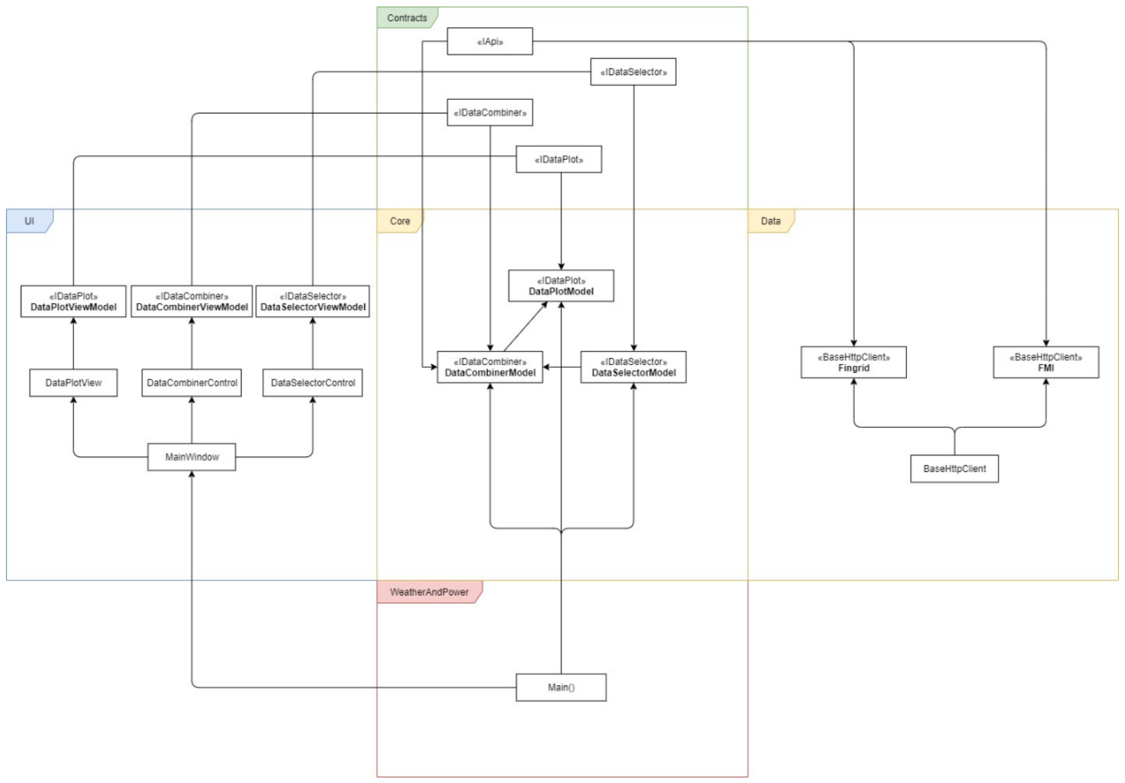
Environment variables:

- FINGRID_API: API key for fetching power data from Fingrid

The Core module can request information from the HTTP source API:s through the Data module This will happen most likely through a query event activated by the user through the UI, or a running timer in the case of automatically updating real-time data. The Data module will process HTTP raw data (XMLs, JSON) into C# readable datatypes. Data combination and processing into various forms for display in the UI will be done in the Core module.



Application architecture



Class diagram

Self-evaluation

Design

Our initial design pattern choice of MVVM has paid off so far, making minor tweaks to the Contracts module or Data module possible. We do not expect to make major changes in the future, rather we will continue to expand the existing functionalities that we have already designed and implemented. The division of graphing and API handling, for example, has allowed us to work concurrently on separate problems and combine them once they were ready. Making even somewhat major changes to one part of the program did not destroy all the work somebody did in the other part.

Changes to original design

One change we had to make to the original design was to switch from a `Main()` function entry-point to a WPF Application `app.xaml` entry-point. The reason for this change was that creating the internal CustomChart object efficiently programmatically required access to XAML styles, and because the custom control has no XAML of its own, the easiest way to access XAML styles is through the Application object.

In addition, we added another window to provide interface to users for better usability when adding new data plots into the graph. In the beginning of our design plan, all user controls belonged to the 'MainWindow', but it makes GUI more complicated because of having too many features in a single window. Instead, we created 'AddWindow' where users provide information for fetching desired data and save their preference settings so that they can open the pre-defined settings in the future.

Charting library implementations

A lot of development time was invested in creating a TimeAxis extension of LinearAxis which takes time-series data as X-axis values and displays them in a user-friendly format. It was then discovered that a DateTimeAxis already exists in the DataVisualization charting library which already has all the necessary features.

Graphing Module

Function

The graphing module is built on top of the System.Windows.Controls.DataVisualization library. A custom control `CustomChart` is derived from the DataVisualization.Charting.Chart class. This custom chart overrides the Series property of the base Chart class, adding logic which observes changes to it and renders the results in the UI.

The CustomChart class also includes adjustments that tailor the graph to our specific needs:

- Time-series X axis with date formatting
- Separate Y axes for our data types
    - Temperature
    - Power
    - Wind
    - Cloudiness

- Enabled databinding for Series property
  - DependencyProperty
  - Observes changes, updates graph
  - Support for ObservableCollection

Custom Chart	
Function name(parameters)	**Explanation, Dependencies**
`private static void OnSeriesChanged(...)`	Responsible for hooking the SeriesChanged method to the CollectionChanged event if Series is set to an ObservableCollection<>
`private Style GetLineStyle(byte[] color)`	Builds a style object for use with LineSeries objects. This determines what the plot lines look like in the UI.
`private void SeriesChanged(...)`	Callback for when Series property is set (or its contents changed if it is Observable). Responsible for plotting the new DataSeries and removing plot lines for deleted DataSeries.
`private void Plot(DataSeries data)`	Creates a LineSeries object from a DataSeries and adds it to the Chart.
`private LinearAxis GetAxis(DataFormat format)`	Gets the Y axis corresponding to given DataFormat.
`private void RefreshAxisVisibilities()`	Refreshes the visibility property of all the Y axes based on what data is hosted by the chart.
`private void Clear()`	Clears all plot lines from the chart
`private void Remove(int id)`	Removes a specific plot line from the chart

FMI API Class

Function

This class handles the forming and sending the requests to the Finnish Meteorological Institute API. It is derived from the BaseHttpClient. The main purpose of the class is to return the list of all the DataSeries objects requested by the user. Naturally, it uses IData interface to handle output data. Most of the methods are static, to decrease used resources. FMI does not require an API key, instead the HttpClient fetches the data directly form the URL.

There are some exceptions associated with the FMI API output data, which are handled within this class. There are three main types of exceptions: erroneous request, missing data and NaN values within the datasets. This class is primarily called in the PlaceHolderModel.cs.

Function name(parameters)	Explanation, Dependencies
`public static string BuildQuery(string mode)`	This function creates a query string based on the user’s input
`public static string BuildRequest(string query)`	Creates the tail of the query based on the provided parameters
`private static XmlNamespaceManager CreateManager(XmlDocument doc)`	Returns a Xml manager needed to parse FMI’s custom namespaces

`private static DataFormat SetFormat(XmlNode node, XmlNamespaceManager mng)`	Selects the format of the DataSeries object based on the type of data requested by the user. Uses Contracts.Globals
`public static async Task<List<DataSeries>?> GetData(string url)`	Parses the Xml file and returns a List of all DataSeries objects requested by the user

Fingrid API Class

Function

This class handles fetching the power data from the Fingrid API which provides not only observation and forecast of electricity consumption and production, but also electricity power generation by different power sources (e.g., wind, nuclear and hydro). It is derived from the BaseHttpClient which contains a static HttpClient instance. The reason why using the static HttpClient is to decrease use of resource instead of creating new instance whenever API calls are requested.

Function name(parameters)	**Explanation, Dependencies**
`Private static string GetApiKey()`	Returns Fingrid API key from system environment. If there is no such variable, the system will be terminated.
`public static async Task<IData> Get(PowerType powerType, string format = null)`	Returns the latest single power data of the given power type.
`public static async Task<DataSeries> Get(PowerType powerType, DateTime startTime, DateTime endTime, string format = null)`	Returns a series of power data during the given time range of the given power type.
`private static void SetHeaders(HttpRequestMessage httpRequestMessage, string format)`	Sets temporary headers for HTTP request.
`private static string ParseRequestUri(int variableId, string query, string format)`	Wraps the given parameters to a single string of request URI for fetching data.
`private static string ParseDateTimeFormat(DateTime dateTime)`	Parses DateTime object to a formatted string which fits Fingrid API format.
private static string ParseParamsToQuery(DateTime startTime, DateTime endTime)	Parses the given data time parameters to a query string for API call.
private static IData ParseXMLToSingleData(int variableId, string XMLString)	Parses the result from API call to a single data object.
private static DataSeries ParseXMLToDataSeries(int variableId, string XMLString)	Parses the result from API call to a series of data object.