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

Design Document

Team 17

Olli Koskelainen
Jun Jung
Vladimir Vechtomov

Table of Contents

High Level Description

Description of the Application

This application is for displaying historical and forecast data of electricity and weather in Finland. User can select what kind of data to display in the graph. User can save datasets to local devices and open them later. User can save the graph as an image file. User can compare data at a certain timeline in pie chart window.

Components

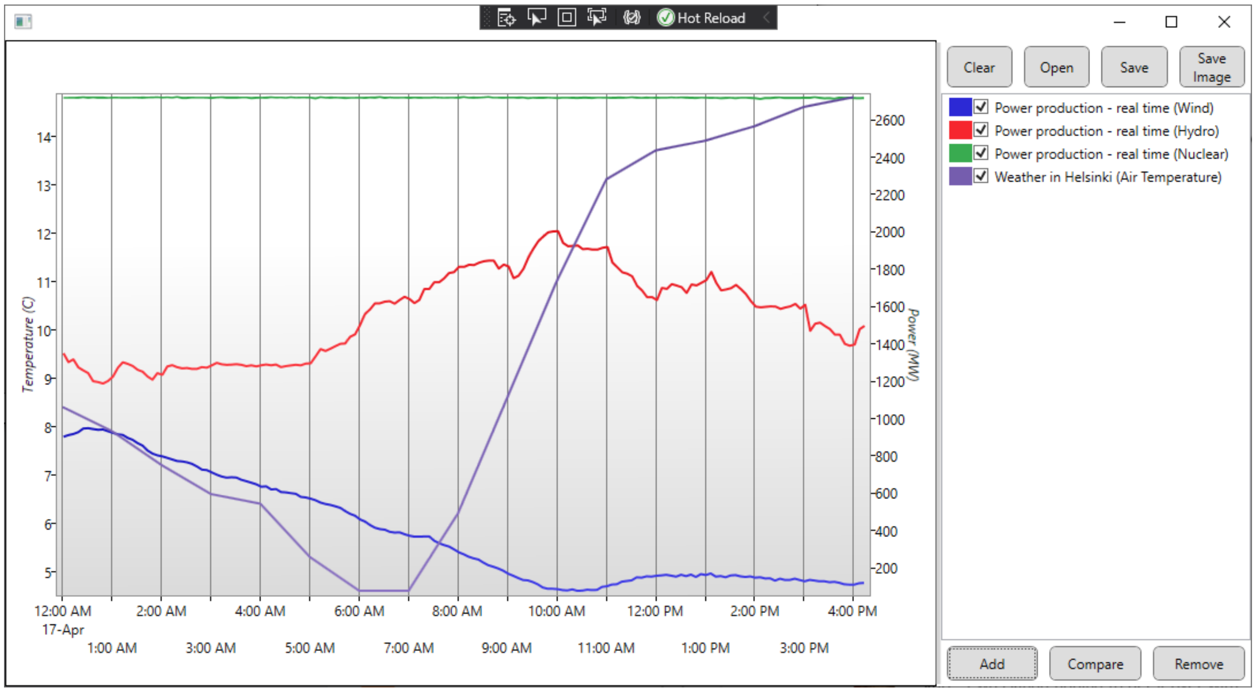
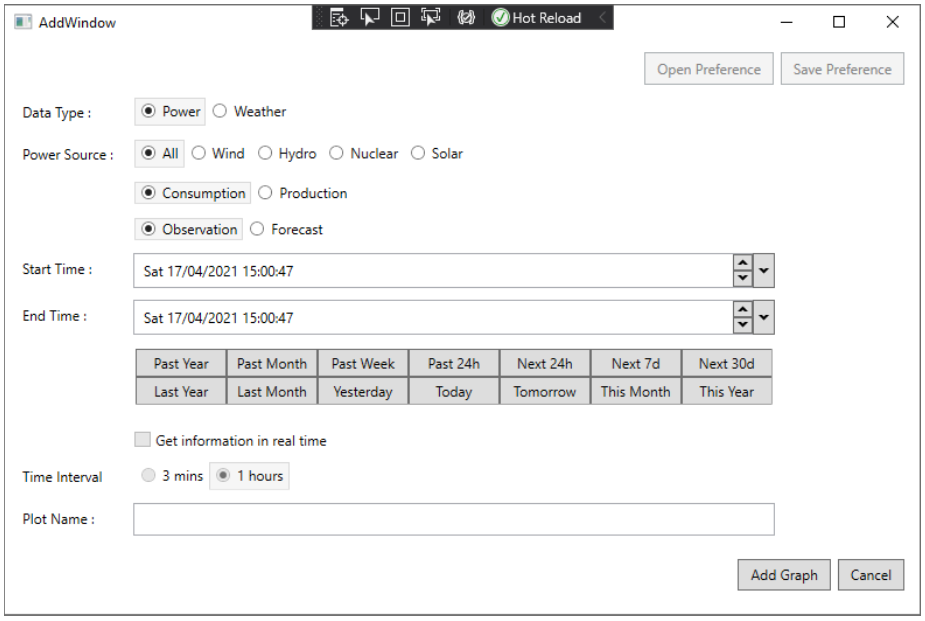


Figure 1. Main Window

In figure 1, the main window consists of a graph component in the left-hand side where the given data is displayed and a sidebar in the right-hand side where user can manipulate the data. The sidebar is composed of few buttons and list view component. The list view component shows what kind of data is displayed in the graph and the buttons allow user to add and delete datasets. Moreover, user can save and open the datasets in local devices.



AddWindow

Hot Reload

Open Preference | Save Preference

Data Type : [✓ Power] Weather

Power Source : [✓ All] Wind Hydro Nuclear Solar

[✓ Consumption] Production

[✓ Observation] Forecast

Start Time : Sat 17/04/2021 15:00:47

End Time : Sat 17/04/2021 15:00:47

Past Year	Past Month	Past Week	Past 24h	Next 24h	Next 7d	Next 30d
Last Year	Last Month	Yesterday	Today	Tomorrow	This Month	This Year

☐ Get information in real time

Time Interval 3 mins [✓ 1 hours]

Plot Name :

Add Graph | Cancel

Figure 2. Add Window

In figure 2, the add window consists of input field components and action buttons. It allows user to select what kind of data to fetch during a periodic datetime range. Data type is defined by selecting parameters in radio buttons and the datetime range is defined though datetime picker or by clicking predefined datetime range buttons. The input fields are switched to power or weather input fields depending on the selected data type property.

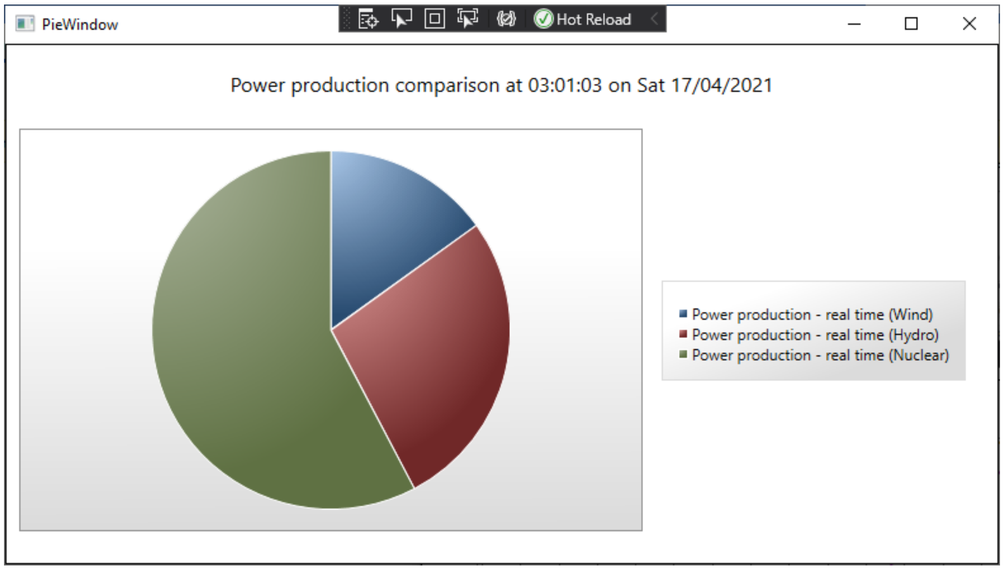

PieWindow
Hot Reload
Power production comparison at 03:01:03 on Sat 17/04/2021
Power production - real time (Wind)
Power production - real time (Hydro)
Power production - real time (Nuclear)

Figure 3. Pie Chart Window

In figure 3, the pie chart window displays the ratio of electricity production by sources at a certain timeline.

Modules

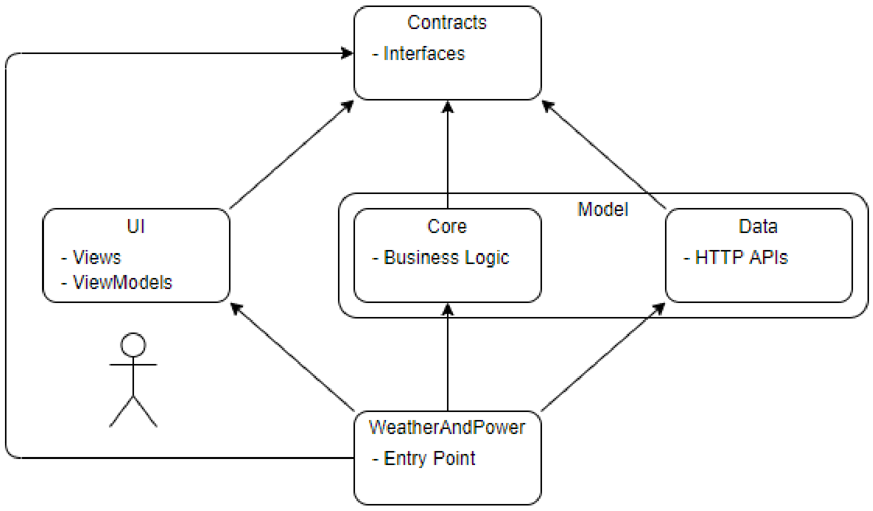


Figure 4. Framework Diagram

This application consists of four modules shown in figure 4: UI, Contracts, Core and Data. 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. More detailed information about responsibility of each component is explained in the section 'Component and Responsibility Description'.

Libraries

'System.Windows.Control.DataVisualization' library has been used for graphing module.

A third party libarary 'Xceed.Wpf.Toolkit' has been used in DateTimeInputView to display datetime pickers because there is date picker but not time picker in basic framework.

Class Diagram

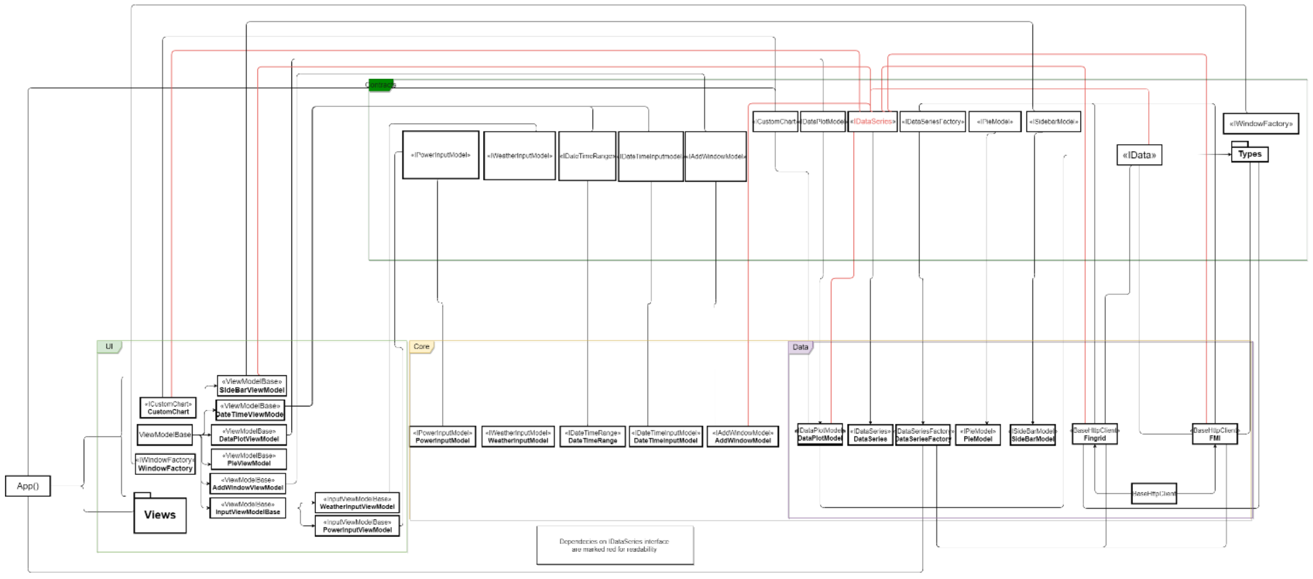

«IPowerInputModel»
«IWeatherInputModel»
«IDateTimeRange»
«IDateTimeInputModel»
«IAddWindowModel»
«ICustomChart»
«IDataPlotModel»
«IDataSeries»
«IDataSeriesFactory»
«IPieModel»
«ISidebarModel»
«IWindowFactory»
«IData»
Types
UI
«ViewModelBase» SideBarViewModel
«ICustomChart» CustomChart
«ViewModelBase» DateTimeViewModel
ViewModelBase
«ViewModelBase» DataPlotViewModel
«IWindowFactory» WindowFactory
«ViewModelBase» PieViewModel
«ViewModelBase» AddWindowViewModel
«ViewModelBase» InputViewModelBase
«InputViewModelBase» WeatherInputViewModel
«InputViewModelBase» PowerInputViewModel
App()
Views
Core
«IPowerInputModel» PowerInputModel
«IWeatherInputModel» WeatherInputModel
«IDateTimeRange» DateTimeRange
«IDateTimeInputModel» DateTimeInputModel
«IAddWindowModel» AddWindowModel
Data
«IDataPlotModel» DataPlotModel
«IDataSeries» DataSeries
«IDataSeriesFactory» DataSeriesFactory
«IPieModel» PieModel
«ISideBarModel» SideBarModel
«BaseHttpClient» Fingrid
«BaseHttpClient» FMI
BaseHttpClient
Dependecies on IDataSeries interface are marked red for readability

Figure 5. Class Diagram

Note: Clear and appropriately sized Class diagram can be found here: https://course-gitlab.tuni.fi/comp-se-110-software-design_2020-2021/team-17/-/blob/master/WeatherAndPower/Documentation/diagrams/final_diagram.png . We recommend downloading the image or using "Web IDE" button on Gitlab.

Boundaries and Interfaces

Graphing

ICustomChart

Interface for accessing chart from Model. This exposes the Pick() and SaveImage() functions, which enter Pick Mode and save the graph as an image respectively. Demanded by Model and implemented by ViewModel

IData

Base interface implemented by all datatypes that we have, such as Temperature, Power, Precipitation and so on. This makes it possible to draw all our data types on the same graph. Exposes a single property, which is a double value. Implemented by all data types in Contracts. Demanded by Model and ViewModel

IDataPlotModel

This is the Model for the DataPlot (chart, graph). It provides access to the model from the ViewModel for properties such as the Data property (all the series we have), and some functions for saving, loading, and manipulating the chart Data. Implemented by the Model. Demanded by ViewModel.

IDataSeries

This is an interface for our Data series. Each IDataSeries object represents a single line on the Chart. Provides access for the CustomLineSeries to all the relevant properties we have for lines. This includes its name, color, format, the data it contains, as well as other properties. Implemented by the DataSeries class in Contracts. Demanded by ViewModel and Model.

IDataSeriesFactory
This is an interface for the DataSeriesFactory. The DataSeriesFactory is provided to our API classes in the Data Module and can be used to create new DataSeries objects. Implemented by the Model. Demanded by Data Model.

IPieModel
Basic interface for the pie graph window used for comparing power production amounts. This interface is implemented in the Model and used for binding data in the PieWindow view through PieViewModel. Implemented by Model and demanded by ViewModel.

AddWindow

IAddWindowModel
This is an interface for the add window used for displaying correct input fields for power data or weather data and binding the parameters in the AddWindow view through AddWindowViewModel. Implemented by Model and demanded by ViewModel.

IDateTimeInputModel
This is an interface for the datetime input fields for selecting a desired datetime range to request data between the range used for power and weather input fields. User can choose a start datetime and end datetime through DateTimeInputView. The datetime range is automatically adjusted in DateTimeInputModel depending the data is whether historical or predictive data. Implemented by Model and demanded by ViewModel.

IDateTimeRange
This is an interface for the predefined datetime range buttons for adjusting starting and ending datetimes in datetime input fields. The interface is implemented in the DateTimeRange class and used in DateTimeInputModel to calculate a certain periodic time range given through a button in DateTimeInputView. Implemented by Model and demanded by ViewModel.

IPowerInputModel
This is an interface for the add window for displaying input fields and binding required parameters to fetch power data from Fingrid API. This interface is implemented in the PowerInputModel that retrieves a corresponding power category to user’s selections through PowerInputView. Implemented by Model and demanded by ViewModel.

IWeatherInputModel
This is an interface for the add window for displaying input fields and binding required parameters to fetch weather data from FMI API. This interface is implemented in the WeatherInputModel that combines selected parameters by user through WeatherInputView. Implemented by Model and demanded by ViewModel.

Other

ISidebarModel

The sidebar is the main user interface on the right-hand side of the main window. This interface exposes sidebar functionality to the ViewModel. It is implemented by the Model and demanded by the ViewModel. The ViewModel uses it to execute functions in the Model.

IWindowFactory

This interface defines a window factory that allows the Model to create windows in the UI. It is implemented by the ViewModel and demanded by the Model. It exposes an overloaded CreateWindow method which produces a certain window depending on the Model that is supplied to it.

INotifyPropertyChanged

This is an interface that we created but it is such an integral part of MVVM and therefore our program that it deserves a short explanation. INotifyPropertyChanged is an interface that can be used to notify its subscribers that a property has changed. When two-way binding data in a View to a property in a ViewModel or Model (through a ViewModel), that module can implement INotifyPropertyChanged. If the interface is implemented, the NotifyPropertyChanged() method can be used (for example: in the setter of the property) to notify the View that the property its value is bound to has changed, and the View should get this new value and display it.

Component and Responsibility Description

Our program consists of 5 discrete components. The View, ViewModel, Model, Data Model and Contracts. In our implementation we have the Model depend on the Data Model. This simplifies API access from the Model slightly. Removing this dependency would be trivial but require some time we do not have. Essentially what would be required is an interface for each API which exposes data getter methods with a certain set of arguments. The FMI and Fingrid classes would implement these interfaces. These dependencies would be injected when the program starts in the App code-behind. There are no direct dependencies between the backend (Model, DataModel) and the UI (View, ViewModel). This means that the backend and frontend can be developed separately, as long as the interface remains the same.

View

Responsible for displaying information for the User. The View is responsible for how the information is displayed.

ViewModel

Responsible for providing information for the View to display. The ViewModel transforms Model information into the format required by the View. The ViewModel can also forward commands (such as button presses) from Views to the Model.

Model

The model houses all business logic of our program. The model is the backend of our program and it is responsible for receiving commands from the UI through ViewModels, requesting data from our APIs and maintaining our data in memory. In our implementation we have split off our APIs in to a separate

Data Model module, but did not have time to completely eliminate direct dependencies between the two modules.

Data Model

This is part of the Model module in our MVVM implementation, but it has been split off into its own module due to it having a bespoke purpose, which is to communicate with our online APIs. The purpose of this split is to make it possible to swap out the Data Model with another model that implements different set of online APIs without changing anything else about our programs functionality. All API related functionality is done in this module, such as creation of REST URLs and parsing response XMLs

Contracts

The contracts module is basically the overhead dependency library of our whole program. It houses all our interfaces, our custom types and global values. The contracts module does not have any dependencies to any other module, and it only serves to provide shared features for our other modules.

Design Decisions

We decided to implement this program in C# and WPF as a learning exercise for a new language. MVVM is our pattern of choice as WPF was basically built around it. C# has multiple tools to create efficient MVVM implementations.

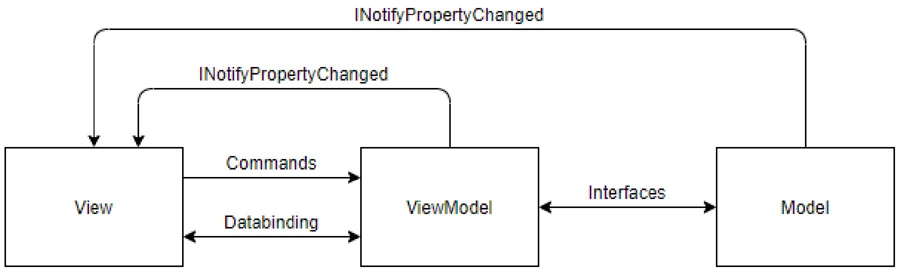


Figure 6 - Our MVVM Implementation

One of the requirements of this project was to create the program in such a way that its components can be interchangeable and MVVM provides this. Each separate component is linked through interfaces and their dependencies can be injected at startup. This setup also helps Separation of Concerns, as all the irrelevant information of a module can be hidden behind the interface. For example, the ViewModel is not concerned about the size or shape of a certain TextBox in the View, and therefore it does not need to know about it. Likewise, a DataPoint on the graph does not need to know what REST URL it came from. It only needs to know its own value, so it can be drawn on the correct spot on the graph.

We used various SOLID principles, such as Dependency Inversion, Interface Segregation and also loosely followed the Single-Responsibility principle. Inversion of Control is also an important principle where control is inverted from the code to the framework. In our program we have implemented IoC using Dependency Injection, either through the constructor, or via a parameter post-instancing.

Self-evaluation

Design

Since mid-term submission we have not made any significant changes to the overall design. The initial choice of MVVM has enabled us to be flexible and to make changes as we discover flaws and features in the implementation of smaller parts. We stayed true to the division of the Model to Core and Data submodules, which made concurrent working on different parts of the program easier. Some of the smaller components have been changed quite a lot, but heavy usage of interfaces and MVVM design patterns made them a lot handier to fix.

Overall, the initial design pattern choice was a great success.

Changes to design

We have created a new helper class – TimeHandler. This class assists FMI and Fingrid in dealing with user-defined timespans. Both FMI and Fingrid needed the same functionality for local time conversion to UTC, warning user about large datasets and checking the validity of user-defined timeframes. The most suitable place for this new class is Data module, where it was added.

We also have improved overall faisafing of the whole project, now most of the non-critical errors are handled in the background and simply notify or warn the user without obstructing the program execution.

The functionality of the user interface was greatly improved, we have added more intuitive and pre-made controls, where the user does not have to type anything. The UI has been made more readable and interactive. Such improvements as multiple visible axis, possibility of removing existing plots and options to save plots have been implemented.

Appendices

Graphing Module

CustomChart.xaml.cs	
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 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 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
`private LinearAxis GetAxis(DataFormat format)`	Get the axis corresponding to the given format
`public override void OnApplyTemplate()`	Loads some views in to variables when the CustomChart view template is applied. The loaded views are required for the Vertical Cursor
`private DateTime GetDateAtCursorPosition(...)`	Used to get the date at mouse position for the vertical cursor.

TimeHandler

Function

TimeHandler is a helper class to Fingrid and FMI. It handles all the necessary conversions and checks related to API's timespans. It is located in the Data module.

Function name(parameters)	**Explanation, Dependencies**
`public static DateTime ConvertToLocalTime(DateTime time)`	FMI and Fingrid handle time in UTC, so we need to convert start and end times from local time to UTC. This function does the conversion and returns corrected time.
`public static bool DataTooBig(DateTime start, DateTime end, double timestep)`	Handles a warning to the user if the requested dataset has a huge number of points (>1000). Gives user options whether to continue or not and based on user choice returns a bool value.
`public static bool ForecastTooFar(DateTime startTime)`	In the FMI the forecast only goes a set amount of time forward, so if the user requests more than that he gets a warning.
`public static List<Tuple<DateTime, DateTime>>`	FMI can handle only a week of data at a time, so if the user requests more, the request needs to be split.

`SplitFMIRequest(DateTime start, DateTime end)`	Returns a list of start-end time pairs for the FMI to process.
`private static void AddTimePair(ref List<Tuple<DateTime, DateTime>> split_times, DateTime start, DateTime end)`	Helper function to the above func.
`public static bool IsTimeValid(DateTime startTime, DateTime endTime)`	Checks the validity of the timespan that user has chosen. Throws an exception if something is wrong or returns true is everything is correct.

FMI API

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 and IDataSeries interfaces 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. One of the features of the FMI API is that the maximum timespan for queries is 168 hours (1 week). For fetching larger data, we split the large queries into week-sized ones. TimeHandler class is utilized for this purpose.

Compared to the previous submission, we have drastically improved failsafing and exceptions visible to the user. Now, even if the requested data has some missing parts, it is still displayed, and the user is notified.

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.
`public static async Task<List<IDataSeries>> GetSingleData(string url, bool is_first_chunk = false)`	This is the function responsible for sending the request to the API. The url that other functions have created is taken as an argument. Boolean flag is just for warning display. This function is called multiple times if the time period is longer than a week. Returns a list of plots.
`public static Dictionary<string, IDataSeries> GetAllData(DateTime startTime, DateTime endTime, int interval, string graphName, string cityName, string parameters, WeatherType.ParameterEnum parameterType)`	This function decides how many queries are made and feeds them one by one to GetSingleData. After that it combines all split graphs based on parameter. Outsiders call this function to plot weather. Returns a complete dict of all available plots sorted by parameter name.

`private static void AddToDict(ref Dictionary<string, IDataSeries> dict, IDataSeries plot)`	Helper function to GetAllData, handles custom behavior of adding to dict and combining charts within it.
`private static TypeFormat GetTypeFormat(XmlNode node, XmlNamespaceManager mng)`	Creates and returns TypeFormat based on parameter. TypeFormat is a custom struct that pairs DataType to DataFormat both ways.
`private static string GetParameter(XmlNode node, XmlNamespaceManager mng)`	Fetches and returns the parameter from the requested XML document. Returned parameter is used for TypeFormat and GetAllData dict sorting.
`private static DataFormat GetFormat(TypeFormat typeformat)`	Returns the format of TypeFormat
`private static dynamic GetType(TypeFormat typeformat, double value)`	Dynamically creates an instance of the correct DataType based on TypeFormat. This instance is in datapoints that populate the plot.
`private static void TellAboutGraphs(List<string> missing_graphs, List<string> found_graphs)`	Handles warning about missing graphs.
`private static string AddDivs(List<string> graphs)`	Non-essential function prettifies the warning message of above function.

Fingrid API

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.