UNIVERSITY OF TWENTE. BEDRIJVENTERREIN ECOFACTORIJ openremote

Design Report

User Interface for a Digital Twin of a Business Park; Ecofactorij

Authors:
Daniar Baialiev - s2425548
Hristo Bizhev - s2855895
Jordan Sberlo - s2752751
Machiel Luning - s2795574
Sekai Ariji - s2731231
Yasin Omidi - s2730006

Supervisor:
Juan López Amézquita

Faculty of Electrical Engineering, Mathematics and Computer Science
TCS Design Project

Contents

Introduction

Ecofactorij is a sustainable business park located in Apeldoorn. At the heart of this business park is a closed distribution system for electricity, which is completely owned by the cooperative “Parkmanagement Ecofactorij” and is unique in the Netherlands. This allows businesses to collaboratively manage energy consumption, which reduces costs and minimizes the environmental impact.

A digital twin of this closed distribution system has been created, which by integrating real-time data from various sources in the park can provide us with data such as energy usage patterns, solar power generation, and battery storage efficiencies. It cannot however visualize this data at the moment.

This report shows the design and development process for creating a user-friendly, interactive web interface for the digital twin, intending to make the provided complex energy data easily understandable and actionable, even for users without a technical background.

Stakeholders

The stakeholders that were identified for this project project are:

- CAES
- Juan López Amézquita - Client representative & Supervisor
- Ecofactorij - Client
- Current companies located at Ecofactorij:
    - Ameco
    - Breads Chemistry
    - Campus Woudhuis
    - Greenferm
    - Grolleman Coldstore
    - ID Logistics
    - Harbers Trucks
    - ITB2 Data centers
    - Royal Oosterberg
    - Preco
    - SILS
    - Sparkling Projects
    - VDL Weweeler
    - Wasco B.V.

Limitations and Constraints

OpenRemote IoT Platform

As a requirement for this system, OpenRemote provides us with a simplified design process and access to some pre-existing features, improving our efficiency in delivering a finished product on time. However, being limited to OpenRemote meant we had less agency over design choices, especially regarding user experience. OpenRemote provides an intuitive and simple layout but introduces some restrictions to the developers and reinforces existing architectural processes, some of which do not align with our client's vision. A notable example was the lack of support for historical data points, for which we ended up implementing our own service.

Digital Twin API

Before the design and implementation of this system, an API had already been partially developed to access the data measuring the physical twin. While drafting the requirements of the system, many of the endpoints were not yet functional but had to be accounted for in the final product. As the front end was being developed, so was the API and adjustments had to be made accordingly. This meant that some features were prioritized based on data availability at the time. However, the data structure and type had remained more or less consistent. Additionally, since the API was developed alongside the front end, some unexpected issues and bugs occurred, which had an impact on the efficiency of development. As a consequence, some features may be more polished than others.

System Requirements

The following requirements are essential to guide the design and development of the project. They outline the functional and user experience specifications necessary to achieve the project's goals and ensure we meet the stakeholders' expectations.

Functional Requirements

- The system must display parameters of Ecofactorij devices such as batteries, circuits, heat storage loads, and PV (Photovoltaic)
    - As a user, I want to see the amount of energy generated by each PV
    - As a user, I want to see the efficiency of each PV
    - As a user, I want to see the parameters of a battery such as discharge and efficiency
    - As a user, I want to see heat storage load efficiency
    - As a user, I want to see active and reactive loads (consumption patterns)
- The system must implement access control, where user's privileges are defined upon the creation of their accounts.
- As an admin, I want to be able to create accounts with different permissions and access levels.
- The system must fetch the data from the API and update the database regularly.
    - PV, circuits, and battery parameters must be fetched every 5 minutes.
    - All other data must be fetched every 24 hours.
- As an admin, I want to set global static parameters, such as lag_time presets and thresholds for firing notifications.
- As a user, I want to see a map with the locations of all the connected devices
    - As a user I want to see information about said device when selecting it on the map.
- As a user, I want to receive a notification when certain asset parameters break through a threshold set by a system administrator
- As a user, I want to be able to visualize asset data in graphs.
    - As a user, I want to be able to see the standard deviation values as error bands on the graph.
- As a user, I want to be able to see the forecast data in a graph with error bands.

Qualitative Requirements

- The UI should be understandable and actionable, even for users without a technical background.

- The UI should be displayed on Mobile and Desktop displays.
- The system must integrate with the external digital twin API
- The system must be deployable on any operating system by using Docker
- The system color palette and design should align with Ecofactorij's design

Risk Analysis

Failure to understand system requirements fully could lead to incomplete systems, which poses a significant risk. To mitigate this, we have interviewed the client at multiple stages of development to gain an understanding of their needs and expectations and prepared a document that clearly defines the system requirements.

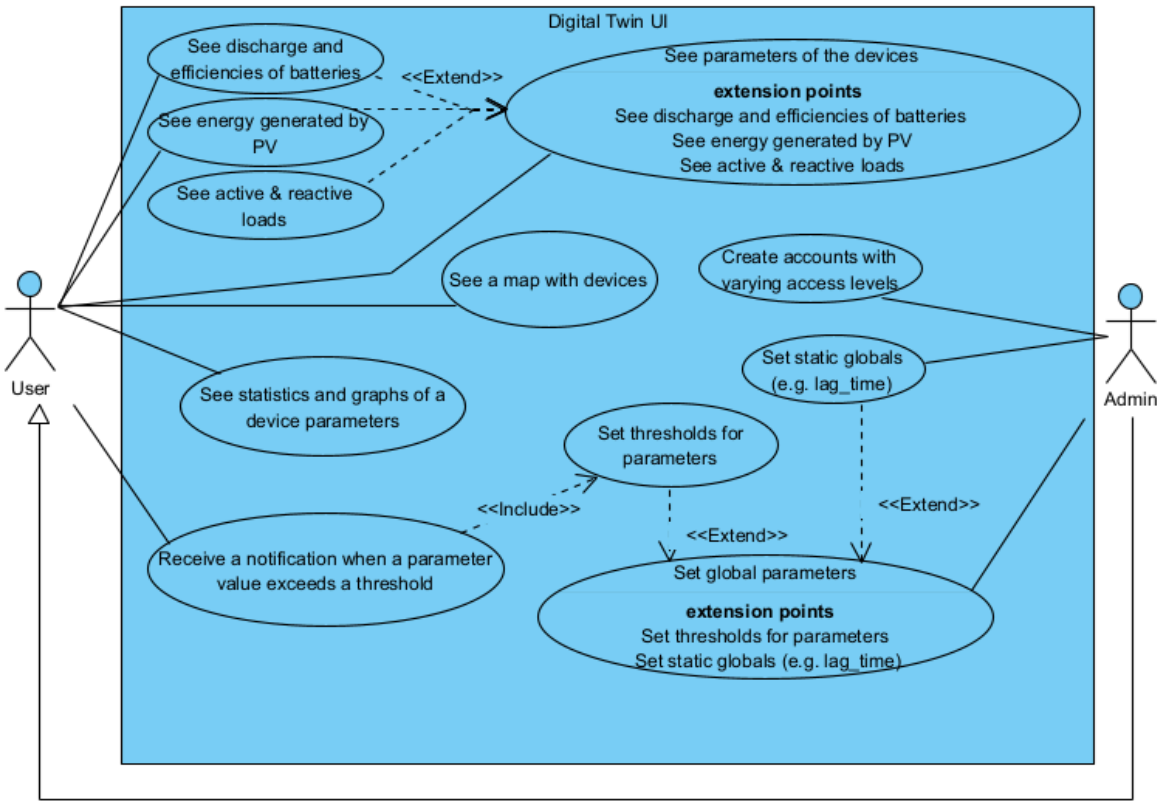
Another risk associated with the development of the digital twin is the dependency on the OpenRemote platform. The platform may have limitations, bugs, or unforeseen constraints that could affect the development of the digital twin. To address this, we conducted early research into the usage of OpenRemote to discover potential limitations and plan accordingly.

Additionally, inaccuracies in energy data or failure to update data in real time could lead to faulty visualizations based on incorrect information, which poses a risk. To mitigate this, it is necessary to regularly test the API integration with the digital twin to ensure real-time data is being accurately processed and displayed. Implementing error handling to identify and resolve any inaccuracies in data also helps to ensure the integrity and accuracy of the information presented in the digital twin.

Design Process

Upon contacting the client, our team requested a meeting in which we could gain a thorough understanding of the system and the desired functionality. Based on the information gathered in that initial meeting, we have drafted a proposal, including our technical and qualitative assessment of the desired product (see Appendix A). After communication with the client and refinements to the proposal, it had been approved and we had our system requirements finalized. We proceeded to meet the client on a weekly basis to gather valuable feedback and stay informed about changes to the API.

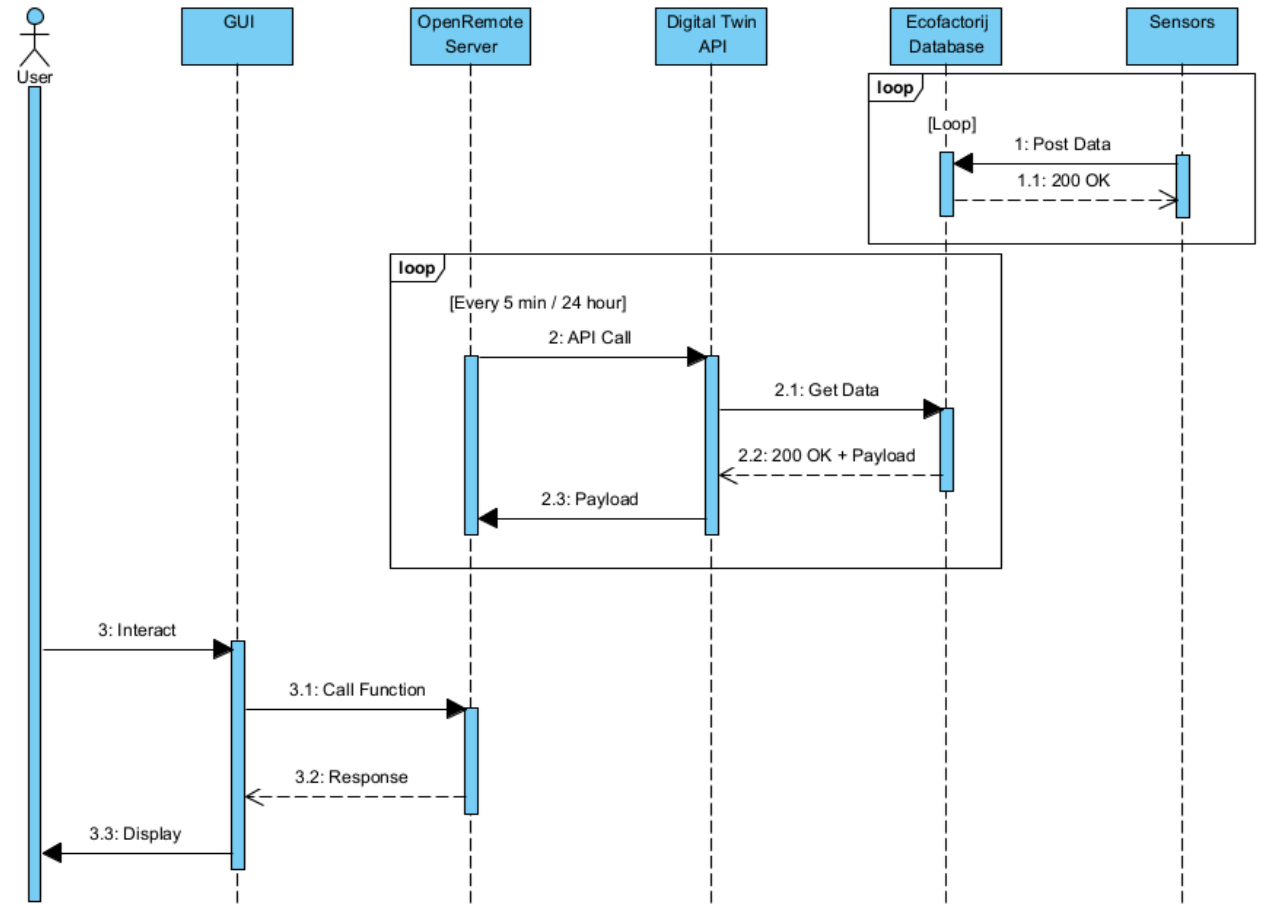
Use Cases



Once all use cases were identified, we explored the tools provided by OpenRemote to see which use cases could be realized with existing features, and which had to be implemented. OpenRemote appeared to offer most of the functionalities we needed for the system with few exceptions such as error bands.

User - Assets Sequence

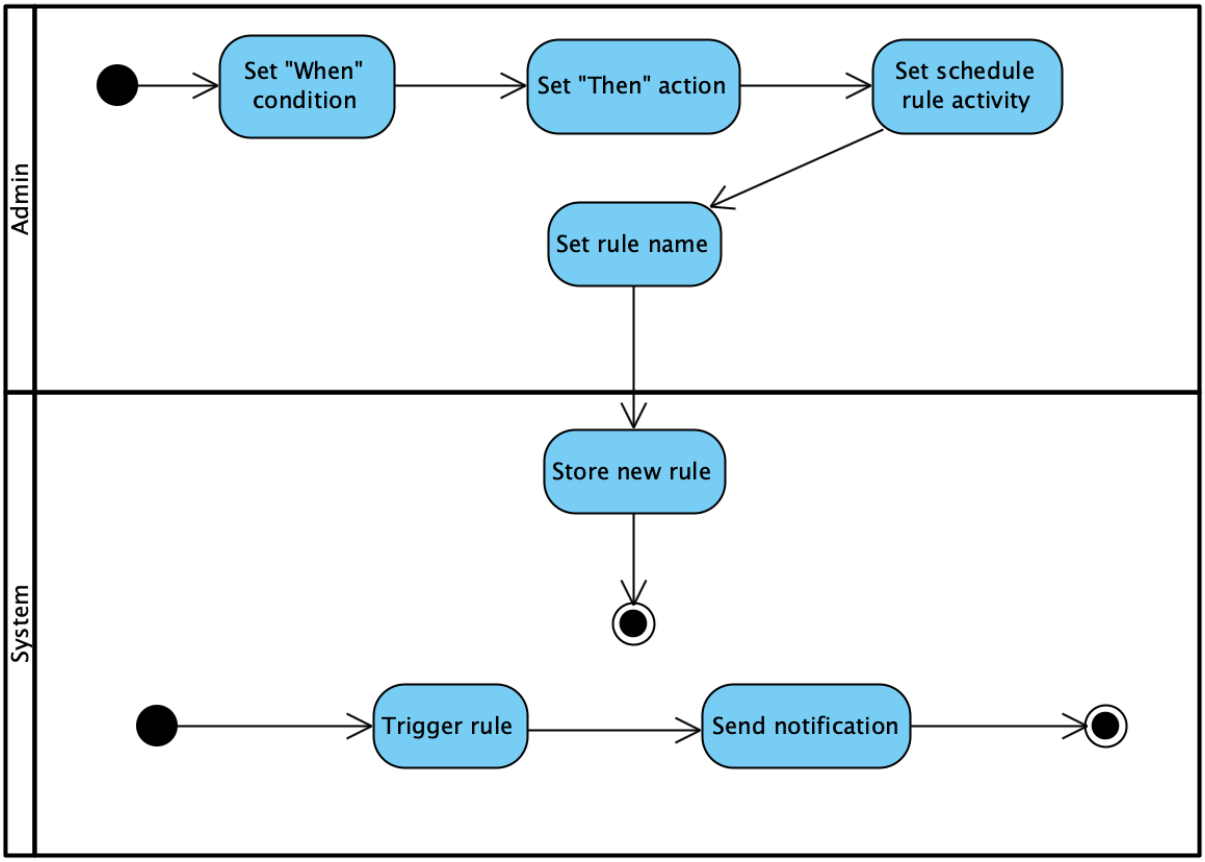
The OpenRemote Manager Server (ORMS) was deployed on a dedicated docker container so that it can run in a variety of environments. While Ecofactorij's database is continuously logging the output coming from the sensors, the ORMS periodically sends requests through the API. Some parameters are being updated every 24 hours, while others every 5 minutes, based on (see requirements). This was done in order to avoid direct interaction between the front end and the Ecofactorij database and to improve the overall responsiveness and security of the system.



This diagram illustrates the flow of information between the different components of the system. The physical twin provides constant data output to the digital twin, while the front-end ORMS periodically fetches this data.

Rules and Alerts

The ability to define rules for certain alerts is a key part of the system. For example, it may be desirable for the user to know when the battery voltage drops below critically low levels, or if a circuit temperature is critically high. OpenRemote offers a variety of rules to be set. In our case, “When - Then” rules serve our purpose perfectly. The (Admin) user has the ability to select an asset or a class of assets and set a “When” condition to fit their criteria. Once the (Admin) user sets the condition, they are prompted to provide a “Then” response. Here they can decide the severity level (1~3) of this event.



Implementation

This section describes the core customizations and extensions applied to OpenRemote to meet the specific requirements. These modifications were necessary to support digital twin integration, enhance data visualization, and provide administrative control of forecasts and asset data.

White Labeling

White labeling OpenRemote involved customizing the OpenRemote platform to replace its original branding and map with that of Ecofactorij. To change the logos, one would normally use a configuration file to instruct OpenRemote to use the custom logos, as explained in their documentation. This however didn't work for us and thus we had to replace the original files, which did work.
For the map, we used OpenStreetMap, which has a vector tile map of the Netherlands. We needed to extract a smaller tileset from this map since we only require a map of the business park. For this, we followed the steps outlined in the documentation of OpenRemote.

OpenRemote Extensions

Line Diagrams

We extended the functionality of line diagrams from the base implementation of OpenRemote by adding support for error bands and forecasts.

Error Bands

To implement the support for error bands, we had to modify the or-chart and or-dashboard-builder components. In the or-dashboard-builder component, we extended the settings menu for the line chart to include a toggle to show or hide the error bands.
In the or-chart component, we extended the data retrieval process to also fetch the standard deviation data points if they exist for the given attribute. In order to also support this feature for forecasts, we had to change this slightly since forecasts don't provide us with a single standard deviation value, but with upper and lower bounds for the error bands.
During the rendering process, we check the settings for the chart, and if the showing of error bands is selected we then add these to the chart. If something goes wrong with this, we make sure that the main data is always rendered.

Forecast

OpenRemote already has some support for forecasts but it offers very little flexibility in its implementation and customization with the only available forecast method being weighted exponential averaging. However, we decided not to use this built-in forecast functionality since it

did not support error bands and it could not be integrated with the forecast API provided and required by our client.
To get around that limitation we implemented the forecasts by creating a child forecast asset and inserting the forecast data into the future data points of that attribute. So the forecasts can be viewed by the user by graphing it in their custom dashboard like they would any other asset attribute. Additionally, we added a “Next 3 days” data range option which would set the range of the graph to be 3 days into the future, giving a complete overview of the forecast. This approach also allowed for the same error bands implementation to work with forecasts as well as historical data.

Digital Twin Devices

OpenRemote has some predefined “assets” to represent different devices. Some assets, like PV and Load assets, already existed within these predefined assets. However, the circuit and batteries were missing. Additionally, the predefined PV and Load asset included some attributes that we didn't need which cluttered the UI and also there were some missing attributes that we did need. So we decided to not use these predefined assets and implement a custom asset for each of the devices from the DT API with all the attributes returned by the API. These assets are listed in Appendix B1.
This setup allowed us full control over these Ecofactorij assets without having to modify the original OpenRemote code.

Admin Config Asset

This asset holds the configuration data of the to-be-fetched forecasts and the 5 min. requests. Admins will be able to change the horizon period (between 15 minutes, an hour, a day, and 3 days) and resolution points (between 1 minute, 15 minutes, and an hour) of the forecasts and set a certain number of days for the lag time of the 5 min. data. The default parameters for the asset are a 3-day horizon, 1-minute resolution, and 7(days) of lag time.

Custom services

In order to fetch, process, and store the data from the digital twin API we had to implement some services. The class diagram of custom services is in Appendix B2

DAO (Data Access Object)

While most of the endpoints of the DT API return some form of historical data, the standard implementation of OpenRemote has no support for inserting data points in the past. Which is a limitation we discovered pretty early on. To work around this limitation we implemented the DAO service which extends the OpenRemote AssetStorageService with additional capabilities, most notably the ability to insert data points into the historical or future record of an asset.

DigitalTwinHTTPclient

In order to communicate with the API we made a custom HTTP client that can take relevant parameters and generate a request to send to the desired API and route and ultimately return the response.

AssetCreationService/AssetUpdateService/AssetForecastService

We implemented the asset creation, update, and forecast services that use the DigitalTwinHTTPclient to fetch the data. They then process and transform the returned responses into a data object which can then be stored in the database with the help of the DAO. These services are used in scheduled events to regularly update the data.

AdminConfigService

The purpose of the admin config asset is to ensure that the adminConfig asset always exists. If it doesn't exists the service assigns the default values for admin parameters: `Horizon.DAY_3, Resolution.HOUR_1, lag_time = 7`. Additionally, this service retrieves the parameters from the Admin asset.

Integration

The custom http client was a handy tool for integrating the Digital Twin API system with our project. This client facilitated API-compliant requests, enabling consistent interaction with the Digital Twin system. It efficiently handled scenarios where API connections were unavailable or when responses were empty or unexpected, aborting the update process gracefully to prevent errors.

Models

The AssetUpdateService manages data updates by retrieving information from the /models API through the DigitalTwinHTTPclient and updating the data via the DAO object.

Historical data

Updating historical data in OpenRemote was straightforward. All of the endpoints with no “Hours” array shown in diagram Appendix C1 had timestamps which allowed for direct timestamp-based updates. The only data cleaning done for this type of data was disregarded of the null values. The historical data entries are highlighted in green within the Appendix B1 diagram.

Daily data

For data recorded in arrays of length 24 (e.g., seasonal efficiencies), we had to invent a workaround to be able to store and view it, as OpenRemote only supports timestamped data. We decided to assign timestamps of a current day to each datapoint and assign hours from the hours array. This method enabled visualization, allowing users to select a “current day” view option, represented as an abstract day in diagrams. This data is marked by yellow colour on the diagram Appendix B1. Additionally, we set up a one-day age limit for this type of data.

Attribute values

Asset attribute values reflect the latest state of each parameter and are shown in the overview of an asset. We need to update these values for the most recent historical data points and the current data of an asset highlighted in blue in the Appendix B1 diagram. However, updating these values created a new historical data point due to the OpenRemote treating the data with timestamps in the past with low priority, assigning it a new timestamp with a current system time. This core OpenRemote implementation was altered to integrate it with our project. Now, timestamps are only assigned when none are available, avoiding unnecessary updates to timestamps when attribute values are added.

Forecasts

Forecast integration required a unique approach to the request construction, as the forecast API endpoints (Appendix C2) used parameters in the request body rather than the URL. A different

set of methods was implemented in DigitalTwinHTTPclient for this purpose. The resolution and horizon parameters are supplied by AdminConfigService as mentioned in the sections above. Additionally, the Forecasts required constant wiping, because after a new forecast is created, the old data is obsolete, so the `wipeForecast()` method was implemented in AssetForecastService.

Testing

Testing plays a vital role in ensuring data accuracy, system functionality, and a user-friendly experience when developing a user interface for the digital twin within OpenRemote. The testing has been split into three different parts, consisting of the testing of the digital twin's API, the testing of the integration of this API into OpenRemote, and finally user testing. The testing of the API and its integration are done in Groovy and make use of the Spock framework. A detailed testing breakdown of these tests can be found in Appendix D.

API Resting

The purpose of the testing of the digital twin's API is to verify that the responses received from the digital twin are structured and formatted as expected. This is done due to the ongoing development of the digital twin and differently structured and/or formatted responses from the API would result in errors when trying to parse this data, which would further result in the data not being shown correctly or being absent.

API Integration Testing

The API integration testing is done in order to verify that the data from the digital twin's API is accurately and completely integrated into OpenRemote, ensuring that the data is displayed correctly in the interface. This API testing has been split into two different parts, each of them testing the integration of a different part of the digital twin.

Models

In this part, we verify the data from the models part of the digital twin. This includes verifying that an asset is created in OpenRemote for each device in the digital twin. Furthermore, for each endpoint, we compare the data from the digital twin with the data that has been inserted into OpenRemote, ensuring that there are no differences and that the data is complete.

Forecasts

In this part, we verify the data from the forecasts part of the digital twin. For each device, we check if we have created a forecast asset in OpenRemote if there exists a forecast for that device. However, unlike the models testing, we can only verify that there are a correct amount of data points in OpenRemote, since the values of the forecast change slightly each time a forecast is requested from the digital twin.

User Testing

User testing was conducted to evaluate the functionality and user experience of the system. This user test emphasizes qualitative analysis and is conducted with a direct customer who is

familiar with the OpenRemote system we used and with a general user who is not familiar with the OpenRemote system at all.
In order to facilitate smooth user testing, a protocol for conducting this test was created beforehand (see Appendix E).

What User testing revealed

The test was conducted with two types of participants, and although there were some points where the complexity of the user interface design made participants confused, we were able to confirm that both participants were able to smoothly follow the instructions given.
From this testing, we were able to find points that users found complicated, unnecessary system choices, and bugs. In addition, the overall design and placement of each functionality were found to be intuitive and easy for users to understand. The feedback also allowed us to make further user-friendliness improvements, such as adding user guide documentation on the system.

Discussion

Throughout the project, we met regularly with our supervisor on a weekly basis, communicated mainly via Teams, and carefully implemented the system to ensure that the system met the client's needs. Comparing the initial system requirements listed in the project proposal with the final system requirements, it can be seen that there is no significant difference between the two system requirements. The final system requirements included more detailed descriptions of several points than the initial system requirements, and these points evolved through communication with our supervisor during the implementation process. Focusing on user-friendliness for people of various backgrounds, frequent communication with the supervisor helped to align development with specific functional needs.

Challenges

OpenRemote had advantages in built-in functionality and ease of integration, but limited customization in several areas. As it was mentioned earlier, OpenRemote does not support inserting historical data points, so implementation of the DAO service was required to process past and future data entries. Furthermore, while OpenRemote supports forecasting, there was little flexibility in implementation and customization, so forecasting was implemented by creating child forecasting assets and inserting the forecast data into the future data points for that attribute.

Future improvements

As for future improvements, the development of an email server to notify important personnel of critical alarms via email and remove all asset types that aren't part of the physical twin, that is, when the system starts it only loads asset types that exist in the database could be considered.

Additionally, the performance of the software may be a factor to consider. It takes around 15-17 minutes to update 5m+24h data and there are only 20 devices currently in the system. There are a couple of ways to make the update process more efficient:

- Insert data points in bulk. Currently, the system inserts the Datapoints in the DB using `insert(List<AssetDatapoint> assetDataList)` method one by one. This can be modified to insert the values in parallel, for example by utilising `PgBulkInsert` functionality.
- More efficient handling of null values. The null values are discarded late in the update process, a bit before inserting them in the DB. By discarding them early, we could remove the overhead, improving performance of the system
- More efficient Http calls. Http calls take the majority of update time and most of them are inefficient. For example, the minimum lag_time is 7 days, but the system only actually updates last 5 minutes and the rest is rewriting the same data. This process can be improved by making the calls in parallel or removing unnecessary overhead. However, that will require additional changes on the Digital Twin part.

Appendices

Appendix A: Project proposal

Project Proposal

User Interface for a Digital Twin of a Business Park using OpenRemote

Group 8: Daniar Baialiev Sekai Ariji Hristo Bizhev Jordan Sberlo Machiel Luning Yasin Omidi
Supervisor: Juan López Amézquita; j.c.lopezamezquita@utwente.nl

Introduction

This document outlines the project requirements approved by our client, Juan López Amézquita, and provides a detailed plan for the development of a web-based user interface (UI) for a digital twin of a business park in the Netherlands known as "Ecofactorij." The digital twin is designed to visualize and monitor real-time energy data and improve energy efficiency through smart energy systems.

The goal of the project is to develop a user-friendly and intuitive interface using OpenRemote, an open-source platform for Internet of Things (IoT) applications. The UI will allow users to easily monitor energy production and usage, battery storage efficiency, and other critical metrics. Additionally, it will enable users to configure the system, adjust backend settings, and set alerts for specific events, such as when energy consumption exceeds a defined threshold.

Functional requirements

- The system must display parameters of Ecofactorij devices such as batteries, circuits, heat storage loads, and PV(Photovoltaic)
    - As a user, I want to see the amount of energy generated by PV
    - As a user, I want to see the efficiency of each PV
    - As a user, I want to see the parameters of a battery such as discharge or efficiency
    - As a user, I want to see heat storage efficiency
    - As a user, I want to see active and reactive loads (consumption patterns)
- The system must implement role-based access control, where roles are defined as the companies located at Ecofactorij. In OpenRemote terminology, these roles are referred to as “Realms”.
- As a user, I want to see a map with the locations of all the connected devices

- As an admin, I want to set global static parameters, such as lag_time or a threshold for firing a notification
- As a user, I want to receive an email when certain parameters break through a threshold set by a system administrator
    - As a user, I want to get an alarm when PV efficiency gets lower than the threshold
    - As a user, I want to get an alarm when battery efficiency gets lower than the threshold
- As a user, I want to be able to visualize data in graphs with standard deviation values.
    - As a user, I want to see the graph of active loads in autumn during weekdays.
- As an admin, I want to be able to create accounts with different permissions and access levels.
- As a user, I want to fetch and update the API data manually.

Note: The list of functional requirements will be extended with an addition of modules currently in development, such as State Estimator and Forecaster

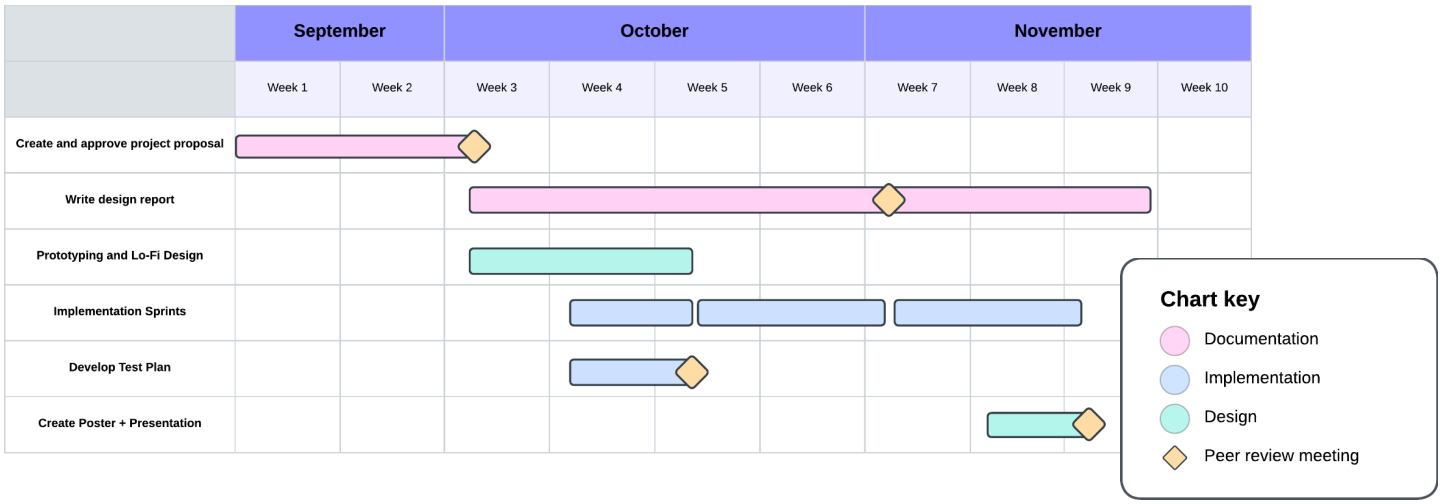
Qualitative requirements

- The UI should be understandable and actionable, even for users without a technical background.
- The UI should be displayed on Mobile and Desktop displays.
- The system must integrate with external digital twin API
- The system must be deployable on any operating system by using Docker
- The system must fetch data from other APIs once in 24 hours
- The system color palette and design should align with Ecofactorij's design

Planning

The deliverables of the project are as follows: Project proposal, Design and testing report, OpenRemote application, Usage Manual and Documentation, and finally a poster and presentation.

The following timeline was made of tasks required to complete these deliverables.



Testing strategy

Continuous unit testing will be performed during the whole development cycle. Additionally, there will be a final system and integration testing stage that will take place during the last week of implementation. This stage is crucial to provide a good quality product for the client.

The primary **objectives** of the testing phase are:

- Functional validation: Identifying and fixing issues related to incorrect system behavior.
- Usability testing: providing better usability by ensuring that the interface is intuitive and easy to navigate.
- Integration Testing: Ensure seamless integration between the UI and backend systems such as the OpenRemote platform and digital twin components.

Testing methods:

- Unit testing: Automated JUnit tests.
- API unit tests: Since the API is still in active development, all used API routes will be put under test to validate that the API response matches the expected structure.
- Integration testing: Continuous manual integration testing by developers.
- User acceptance testing with the client. Since user-friendliness of the interface is the main aim of the project, the development team will iteratively perform user acceptance testing with the client.

Risk analysis

Failure to understand system requirements:

Risk: Failure to understand system requirements fully, leading to incomplete systems.
Mitigation: Interview clients at an early stage and prepare a document that clearly defines system requirements.

OpenRemote platform dependency:

Risk: The platform may have limitations, bugs, or unforeseen constraints that could affect the development of the DT.

Mitigation: Early research into the usage of OpenRemote to discover potential limitations and develop solutions to work around the limitation.

Data Integrity and Accuracy:

Risk: Inaccuracies in energy data or failure to update data in real time could lead to faulty visualizations based on incorrect information.

Mitigation: Regularly test the API integration with the digital twin to ensure real-time data is being accurately processed and displayed. Implement error handling to identify and resolve any inaccuracies in data.

Stakeholders

Ecofactorij: https://ecofactorij.nl/
Client: Juan López Amézquita
Current companies located at Ecofactorij: Ameco, Breads Chemistry, Campus Woudhuis, Greenferm, Grolleman Coldstore, ID Logistics, Harbers Trucks, ITB2 Data centers, Royal Oosterberg, Preco, SILS, Sparkling Projects, VDL Weweeler, Wasco B.V.

Limitations and constraints

Obligation to use Open Remote
Dependence on Digital Twin API
Digital twin for monitoring purposes only

Project organization

Communication

Communication with the client is set up through Microsoft Teams and emails. Weekly meetings will be scheduled primarily on Mondays to provide updates, get feedback, and discuss progress. For internal team communications, we will use Discord. Additionally, internal meetings will be scheduled 2-3 times per week to review tasks, discuss issues, and keep everyone updated and on track with the project.

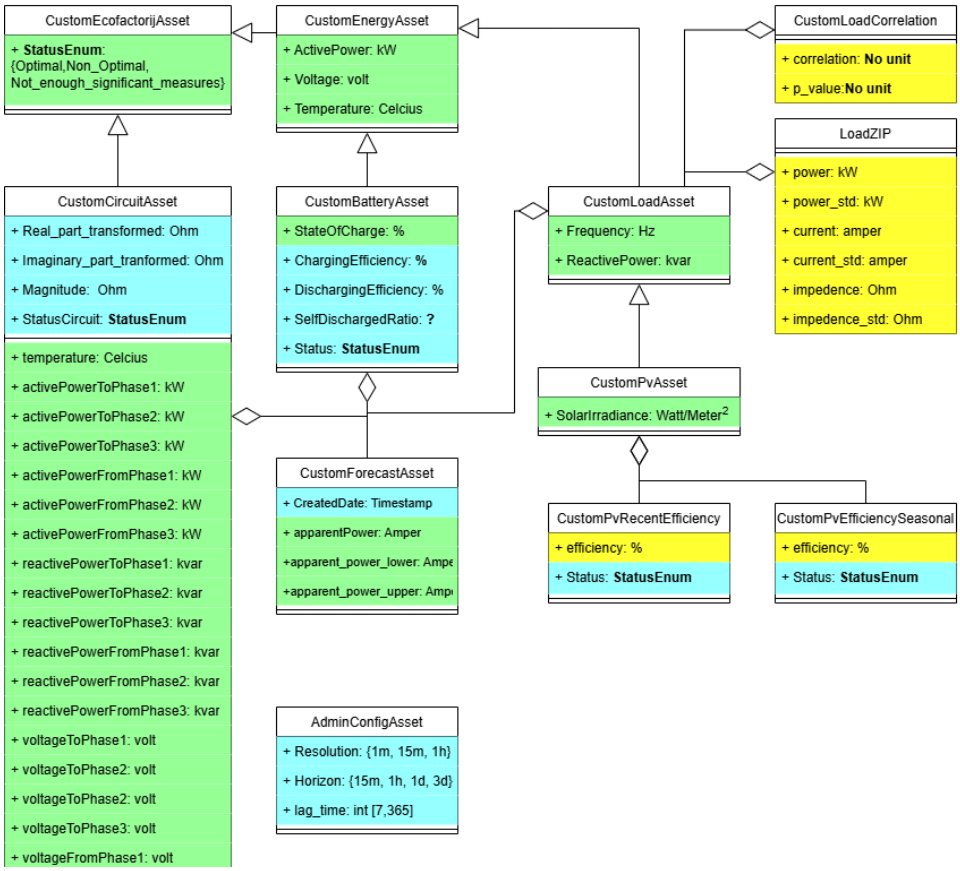
Responsibilities

We will adopt a flexible and collaborative approach to distributing responsibilities within the team for this project. The responsibilities are divided among UI design, implementation, documentation, testing, and other project components. We take into account each team member's technical skills and experience with the relevant tools as well as their interest in particular aspects of the digital twin development.

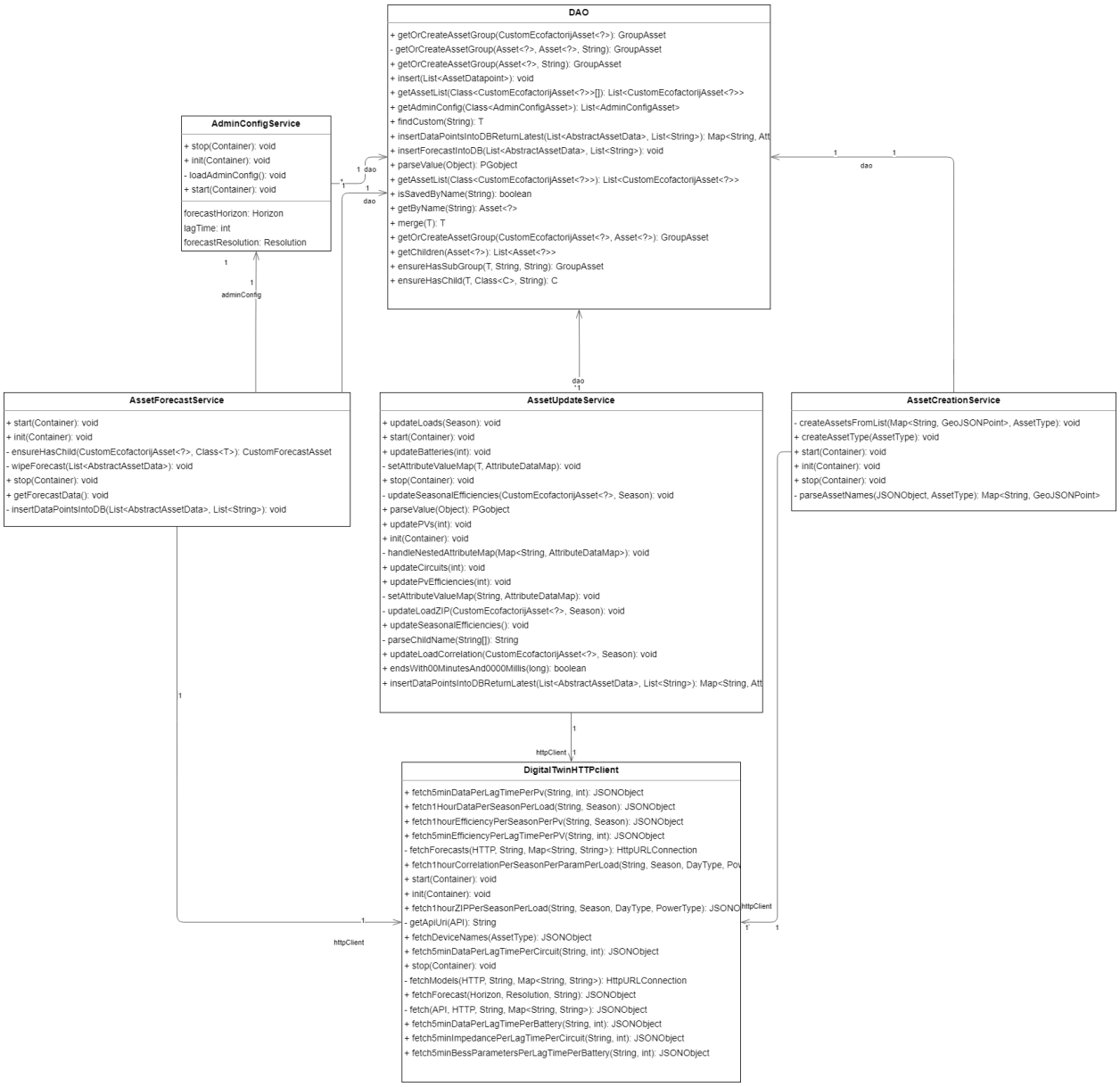
To maintain a balanced workload and ensure that everyone is contributing fairly, we hold regular meetings where tasks are reassessed and adjusted as needed. During these meetings, we provide updates on our progress, address any challenges, and redistribute responsibilities if certain tasks require additional support or focus. This collaborative process ensures the team remains aligned with the project's goals.

Appendix B: Class diagrams

B1



B2


DAO
+ getOrCreateAssetGroup(CustomEcofactorijAsset<?>): GroupAsset
- getOrCreateAssetGroup(Asset<?>, Asset<?>, String): GroupAsset
+ getOrCreateAssetGroup(Asset<?>, String): GroupAsset
+ insert(List<AssetDatapoint>): void
+ getAssetList(Class<CustomEcofactorijAsset<?>>[]): List<CustomEcofactorijAsset<?>>
+ getAdminConfig(Class<AdminConfigAsset>): List<AdminConfigAsset>
+ findCustom(String): T
+ insertDataPointsIntoDBReturnLatest(List<AbstractAssetData>, List<String>): Map<String, Att
+ insertForecastIntoDB(List<AbstractAssetData>, List<String>): void
+ parseValue(Object): PGobject
+ getAssetList(Class<CustomEcofactorijAsset<?>>): List<CustomEcofactorijAsset<?>>
+ isSavedByName(String): boolean
+ getByName(String): Asset<?>
+ merge(T): T
+ getOrCreateAssetGroup(CustomEcofactorijAsset<?>, Asset<?>): GroupAsset
+ getChildren(Asset<?>): List<Asset<?>>
+ ensureHasSubGroup(T, String, String): GroupAsset
+ ensureHasChild(T, Class<C>, String): C
AdminConfigService
+ stop(Container): void
+ init(Container): void
- loadAdminConfig(): void
+ start(Container): void
forecastHorizon: Horizon
lagTime: int
forecastResolution: Resolution
AssetForecastService
+ start(Container): void
+ init(Container): void
- ensureHasChild(CustomEcofactorijAsset<?>, Class<T>): CustomForecastAsset
- wipeForecast(List<AbstractAssetData>): void
+ stop(Container): void
+ getForecastData(): void
- insertDataPointsIntoDB(List<AbstractAssetData>, List<String>): void
AssetUpdateService
+ updateLoads(Season): void
+ start(Container): void
+ updateBatteries(int): void
- setAttributeValueMap(T, AttributeDataMap): void
+ stop(Container): void
- updateSeasonalEfficiencies(CustomEcofactorijAsset<?>, Season): void
+ parseValue(Object): PGobject
+ updatePVs(int): void
+ init(Container): void
- handleNestedAttributeMap(Map<String, AttributeDataMap>): void
+ updateCircuits(int): void
+ updatePvEfficiencies(int): void
- setAttributeValueMap(String, AttributeDataMap): void
- updateLoadZIP(CustomEcofactorijAsset<?>, Season): void
+ updateSeasonalEfficiencies(): void
- parseChildName(String[]): String
+ updateLoadCorrelation(CustomEcofactorijAsset<?>, Season): void
+ endsWith00MinutesAnd0000Millis(long): boolean
+ insertDataPointsIntoDBReturnLatest(List<AbstractAssetData>, List<String>): Map<String, Att
AssetCreationService
- createAssetsFromList(Map<String, GeoJSONPoint>, AssetType): void
+ createAssetType(AssetType): void
+ start(Container): void
+ init(Container): void
+ stop(Container): void
- parseAssetNames(JSONObject, AssetType): Map<String, GeoJSONPoint>
DigitalTwinHTTPclient
+ fetch5minDataPerLagTimePerPv(String, int): JSONObject
+ fetch1HourDataPerSeasonPerLoad(String, Season): JSONObject
+ fetch1hourEfficiencyPerSeasonPerPv(String, Season): JSONObject
+ fetch5minEfficiencyPerLagTimePerPV(String, int): JSONObject
- fetchForecasts(HTTP, String, Map<String, String>): HttpURLConnection
+ fetch1hourCorrelationPerSeasonPerParamPerLoad(String, Season, DayType, Pow
+ start(Container): void
+ init(Container): void
+ fetch1hourZIPPerSeasonPerLoad(String, Season, DayType, PowerType): JSONO
- getApiUri(API): String
+ fetchDeviceNames(AssetType): JSONObject
+ fetch5minDataPerLagTimePerCircuit(String, int): JSONObject
+ stop(Container): void
- fetchModels(HTTP, String, Map<String, String>): HttpURLConnection
+ fetchForecast(Horizon, Resolution, String): JSONObject
- fetch(API, HTTP, String, Map<String, String>): JSONObject
+ fetch5minDataPerLagTimePerBattery(String, int): JSONObject
+ fetch5minImpedancePerLagTimePerCircuit(String, int): JSONObject
+ fetch5minBessParametersPerLagTimePerBattery(String, int): JSONObject
dao
adminConfig
httpClient
httpClient
httpClient
1
*

Appendix C: API endpoints

C1

1hour_data_per_season_per_load


```
"kW": 13.19,
"volts": 398.62,
"Celsius": 26.1,
"Hz": 50.01,
"kvar": 2.75
```


5min_data_per_lag_time_per_pv


```
"kW": 0.05,
"volts": 414.48,
"Celsius": 24.7,
"W_per_m2": 178,
"kvar": 3.79
```


1hour_efficiency_per_season_per_pv


```
"Efficiencies": [24],
"Hours": [24],
"Status": "optimal"
```


5min_data_per_lag_time_per_battery


```
"kW": 0.05,
"volts": 414.48,
"SoC": 24.7,
"Celsius": 178
```


5min_data_per_lag_time_per_circuit


```
"kW_1_L1": 16.19,
"kW_1_L2": 15.57,
"kW_1_L3": 16.55,
"kvar_1_L1": 1.07,
"kvar_1_L2": 1.03,
"kvar_1_L3": 1.1,
```


1hour_correlation_per_season_per_param_per_load


```
"Correlations": [24],
"p-values": [24],
"hours": [24]
```


1hour_data_per_season_per_pv


```
"kW": 0.05,
"volts": 414.48,
"Celsius": 24.7,
"W_per_m2": 178,
"kvar": 3.79
```


5min_efficiency_per_lag_time_per_pv


```
"Efficiencies": [24],
"Hours": [24],
"Status": "optimal"
```


5min_bess_parameters_per_lag_time_per_battery


```
"Charging_efficiency": 1,
"Discharging_efficiency": 1,
"Self-discharged_ratio":
0.00463,
"Status": "non-optimal"
```


5min_impedance_per_lag_time_per_circuit


```
"Real_part_transformed": null,
"Imaginary_part_transformed": null,
"Magnitude": null,
"Status": "Not enough significant
measurements"
```


1hour_ZIP_per_season_per_load


```
"hours": [24].
"P_std": [24],
"I_std": [24],
"Z_std": [24],
"P_cte": [24],
"I_cte": [24],
"Z_cte": [24]
```


1hour_ZIP_per_season_per_load


```
size = horizon/resolution
"ds": [size],
"unit": text,
"apparent_power": [size],
"apparent_power_lower": [size],
"apparent_power_upper": [size]
```


C2

1hour_ZIP_per_season_per_load


```
size = horizon/resolution
"ds": [size],
"unit": text,
"apparent_power": [size],
"apparent_power_lower": [size],
"apparent_power_upper": [size]
```

Appendix D: Detailed testing breakdown

#	Test name	Purpose	Expected result	Test file
1	Check creation of assets in OpenRemote	Verify that for each device an asset has been created in OpenRemote	Pass; For all devices, there exists a corresponding asset	TestDT_Models_to_OpenRemote.groovy
2	Check 5 min data per lag time PVs	Verify per PV that its 5-minute data has been correctly put into OpenRemote	Pass; For all PVs, the 5-minute data is correct	TestDT_Models_to_OpenRemote.groovy
3	Check 5 min data per lag time Batteries	Verify per battery that its 5-minute data has been correctly put into OpenRemote	Pass; For all batteries, the 5-minute data is correct*	TestDT_Models_to_OpenRemote.groovy
4	Check 5 min data per lag time Circuits	Verify per circuit that its 5-minute data has been correctly put into OpenRemote	Pass; For all circuits, the 5-minute data is correct	TestDT_Models_to_OpenRemote.groovy
5	Check creation of configuration asset	Verify that the admin configuration asset has been created in OpenRemote	Pass; The admin configuration asset has been created	TestDT_Models_to_OpenRemote.groovy
6	Check Bess parameters for batteries	Verify per battery that its BESS parameters have been correctly put into OpenRemote	Pass; For all batteries, the BESS parameters are correct	TestDT_Models_to_OpenRemote.groovy
7	Check impedance parameters for circuits	Verify per circuit that its impedance parameters have been correctly put into OpenRemote	Pass; For all circuits, the impedance parameters are correct	TestDT_Models_to_OpenRemote.groovy
8	Check 5 min efficiency per lag time PVs	Verify per PV that its 5-minute efficiency data has been correctly put into OpenRemote	Pass; For all PVs, the 5-minute efficiency data is correct	TestDT_Models_to_OpenRemote.groovy
9	Check 1 hour efficiency per season PVs	Verify per PV that its 1-hour efficiency data has been correctly put into OpenRemote	Pass; For all PVs, the 1-hour efficiency data is correct	TestDT_Models_to_OpenRemote.groovy

10	Check 1 hour ZIP per season Loads	Verify per load that its ZIP data has been correctly put into OpenRemote	Pass; For all loads, the ZIP data is correct	TestDT_Models_to_OpenRemote.groovy
11	Check 1 hour data per season Loads	Verify per load that its 1-hour data has been correctly put into OpenRemote	Pass; For all loads, the 1-hour data is correct	TestDT_Models_to_OpenRemote.groovy
12	Check 1 hour data per season Loads (only Hz for PV)	Verify per PV that its 1-hour Hz data has been correctly put into OpenRemmote	Pass; For all PVs, the 1-hour Hz data is correct	TestDT_Models_to_OpenRemote.groovy
13	Check 1 hour correlation per season per parameter Loads	Verify per load that its correlation data has been correctly put into OpenRemote	Pass; For all loads, the correlation data is correct	TestDT_Models_to_OpenRemote.groovy
14	Check creation of forecasts in OpenRemote	Verify that a forecast asset is created if it exists in the digital twin and that the forecast consists of the correct amount of data points	Pass; For all available forecasts, there exists a corresponding asset with the correct amount of data points.	TestDT_Forecasts_to_OpenRemote.groovy
15	Get names of devices - Batteries	Verify that the response for batteries from the digital twin contains a key “batteries_list” and its value is not null	Pass; The response to get the batteries from the digital twin is as expected	TestAPI.groovy
16	Get names of devices - Circuits	Verify that the response for circuits from the digital twin contains a key “circuits_list” and its value is not null	Pass; The response to get the circuits from the digital twin is as expected	TestAPI.groovy
17	Get names of devices - Loads	Verify that the response for loads from the digital twin contains a key “loads_list” and its value is not null	Pass; The response to get the loads from the digital twin is as expected	TestAPI.groovy
18	Get names devices - PVs	Verify that the response for PVs from the digital twin contains a key “pvs_list” and its value is not null	Pass; The response to get the PVs from the digital twin is as expected	TestAPI.groovy

19	Data per load per season	Verify that the response for 1-hour data per season per load contains the right parameters and their value is the correct type	Pass; The response to get the 1-hour data per season per load from the digital twin is as expected	TestAPI.groovy
20	Verify no null values in ZIP coefficient lists	Verify that the response for the ZIP data contains no null values	Pass; The response to get the ZIP data from the digital twin is as expected	TestAPI.groovy
21	Hourly correlation per season, param, load	Verify that the response for the 1-hour correlation data is of the correct type and size	Pass; The response to get the 1-hour correlation data from the digital twin is as expected	TestAPI.groovy
22	Hourly data per season per PV	Verify that the response for 1-hour data per season per PV contains the right parameters and their value is the correct type	Pass; The response to get the 1-hour data per season per PV from the digital twin is as expected	TestAPI.groovy
23	Hourly efficiency per season per PV	Verify that the response for 1-hour efficiency data contains the right parameters and is of the correct size	Pass; The response to get the 1-hour efficiency data from the digital twin is as expected	TestAPI.goovy
24	5min data per pv - lag time	Verify that the response for 5-minute data per pv contains the right parameters and their value is the correct type	Pass; The response to get the 5-minute data per pv from the digital twin is as expected	TestAPI.groovy
25	5min efficiency per pv - lag time	Verify that the response for 5-minute efficiency data contains the right parameters and is of the correct size	Pass; The response to get the 5-minute efficiency data from the digital twin is as expected	TestAPI.groovy
26	5min data per battery - lag time	Verify that the response for 5-minute data per battery contains the right parameters and their value is the correct type	Pass; The response to get the 5-minute date per battery from the digital twin is as expected	TestAPI.groovy
27	5min bess param per battery - lag	Verify that the response for BESS parameters	Pass; The response to get the BESS	TestAPI.groovy

	time	contains the right parameters and their value is the correct type	parameters from the digital twin is as expected	
28	5min data per circuit - lag time	Verify that the response for 5-minute data per circuit contains the right parameters and their value is the correct type	Pass; The response to get the 5-minute data per circuit from the digital twin is as expected	TestAPI.groovy
29	5min impedance per circuit - lag time	Verify that the response for the impedance parameters contains the right parameters	Pass; The response to get the impedance parameters from the digital twin is as expected	TestAPI.groovy

If a test from TestDT_Models_to_OpenRemote.groovy or TestDT_Forecasts_to_OpenRemote.groovy fails, OpenRemote may not have been updated with the latest data from the digital twin yet. Trying again later after the data has been updated might resolve it.

Appendix E: User testing protocol

User Testing

Introduction

In part of delivering a complete product, user testing is invaluable in evaluating the functionality and user-experience of our system. In the case of our product, user tests are conducted with the direct client, with an emphasis on qualitative analysis.

Goals

In these tests our aim is to assess the following aspects of our platform:

- Usability - How long does it take users to complete certain actions?
- Experience - Are users satisfied with the product? How can it be improved?
- Features - uncovering usability issues and or identifying missing features

Method

Interact

Users are given a series of tasks to be completed:

1. As a user, locate BATTERY_1 in the interactive map page and view its data, then press view.
2. As a user, locate Circuit M13_to_M300 in the assets page and generate a graph showing the active power to phase 1. Then change the time-frame.
3. As an admin, add a when-then rule that creates an alarm whenever the voltage of PV drops below 3.5V.
    For ‘When’ select PV and set parameter
    For ‘Then’ select alarm
4. As an admin, create a user with only read permission.
5. As an admin, create a dashboard in the insights page, create a line chart and add attributes: current from phase 1,2 and 3 of circuit M13_to_M300.
6. As an admin, freely interact with the platform for 60 seconds.

Record

User interactions are recorded using OBS with Input Overlay plugin (https://obsproject.com/forum/resources/input-overlay.552/). These recordings are then analyzed for time and error rate of each individual task. For error detection, the recordings are reviewed by comparing user input behavior, and system response. In cases where user input results in an unexpected response, it is flagged as an error.

Interview

This section will focus on evaluating the user-experience and features of the system. Our users are interviewed immediately upon completing the interaction. The questions are as follows:

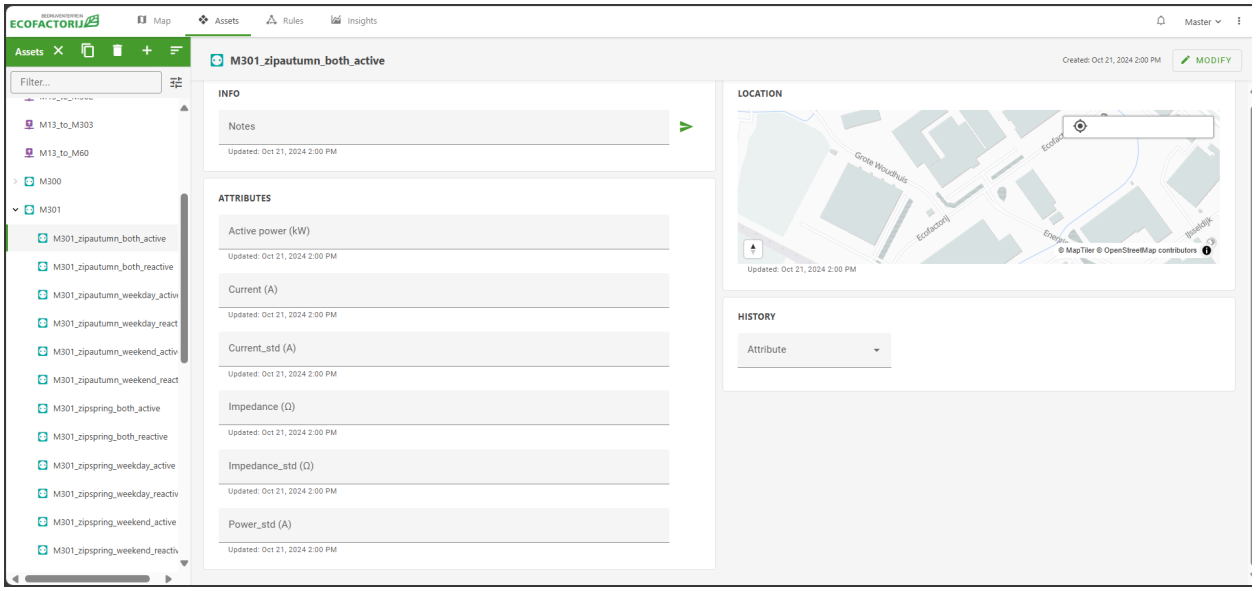
1. How easy was it to find the specific assets or data you were looking for?
2. Were there any points where you felt lost or confused about where to go next
3. How would you rate the overall design and layout of the platform? Did it feel intuitive?
4. Were there any issues you encountered with existing features?
5. What improvements or additional features would enhance your experience using the platform?

Prototype testing

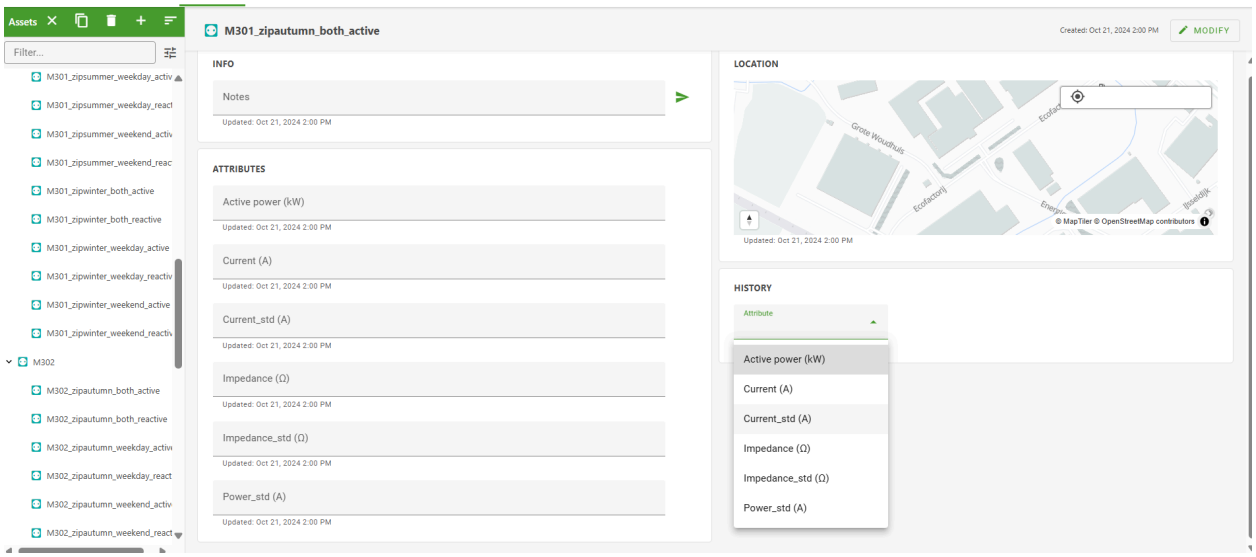
OUTCOMES: PUT EVERYTHING IN THE ASSET NAMES
M13

In this part of the user testing, we will evaluate three different frontend design prototypes for displaying asset data (load_zip, pv_efficiencies_seasonal, and load_correlation). Each prototype presents a different approach to organizing asset parameters between asset names and attribute names. This part of the test will assess which approach users find more intuitive and easy to navigate.

Prototypes Description:
Prototype 1 - All Parameters in Attribute Names:



In this prototype, all relevant parameters are appended to the attribute names. The asset names remain as the core identifiers, while the specific parameters are distinguished through the attributes associated with each asset.
Prototype 2 - All Parameters in Asset Names:



This prototype places all parameters within the asset names themselves. Here, asset names are more detailed, containing both the asset identifier and its associated parameters. The attribute names remain basic and unchanged.
Prototype 3 - Combined Method:

In the combined method, some parameters are assigned to the asset names, while others are appended to the attribute names. This approach aims to strike a balance, potentially making it easier to filter or search for assets, while keeping detailed parameter information accessible in the attributes.
Task:
Users will be asked to interact with each of the three prototypes and complete a set of predefined tasks (e.g., locating specific assets, generating graphs, setting parameters). During the interaction, they will be asked to provide feedback on the usability, clarity, and efficiency of each prototype.

Question:
Customization:
After testing the three prototypes, users will be asked: If you could choose how to distribute parameters between asset names and attribute names, how would you arrange them? Which parameters would you prefer to see in the asset names, and which ones in the attribute names? This question will help us understand user preferences for organizing information, providing valuable insights into how we can allow for more flexible, user-defined customization in future designs.

Appendix F: User guide

Below is the user guide for our application.

User Guide

OpenRemote User-Interface for Ecofactorij

Authors:
Daniar Baialiev
Hristo Bizhev
Jordan Sberlo
Machiel Luning
Sekai Ariji
Yasin Omidi

Supervisor:
Juan López Amézquita

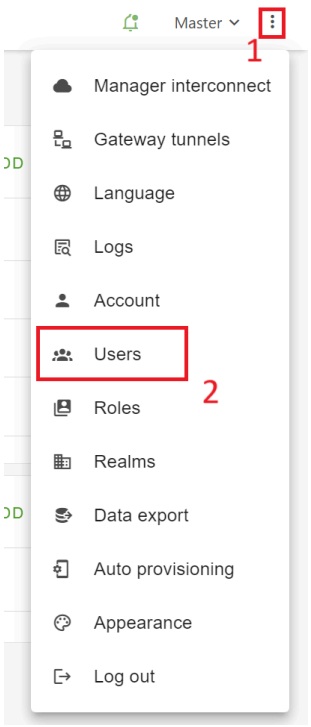
Introduction

The user interface shares many of its functionalities with the OpenRemote default interface and will therefore behave similarly in most cases. However, some features were changed and new ones were added specifically for the application requested by Ecofactorij. This manual provides a step-by-step guide for each of the functionalities offered by the system, including pictures and examples.

If you are looking to perform a specific task please refer to the table of contents below:

Users

By default the system is provided with an admin user account, with access to view, create, modify, and delete additional users. Only the admin account has the ability to manage other accounts and should therefore be secured appropriately. To navigate to the Users page, click on the three dots on the top right of the page, then select Users.

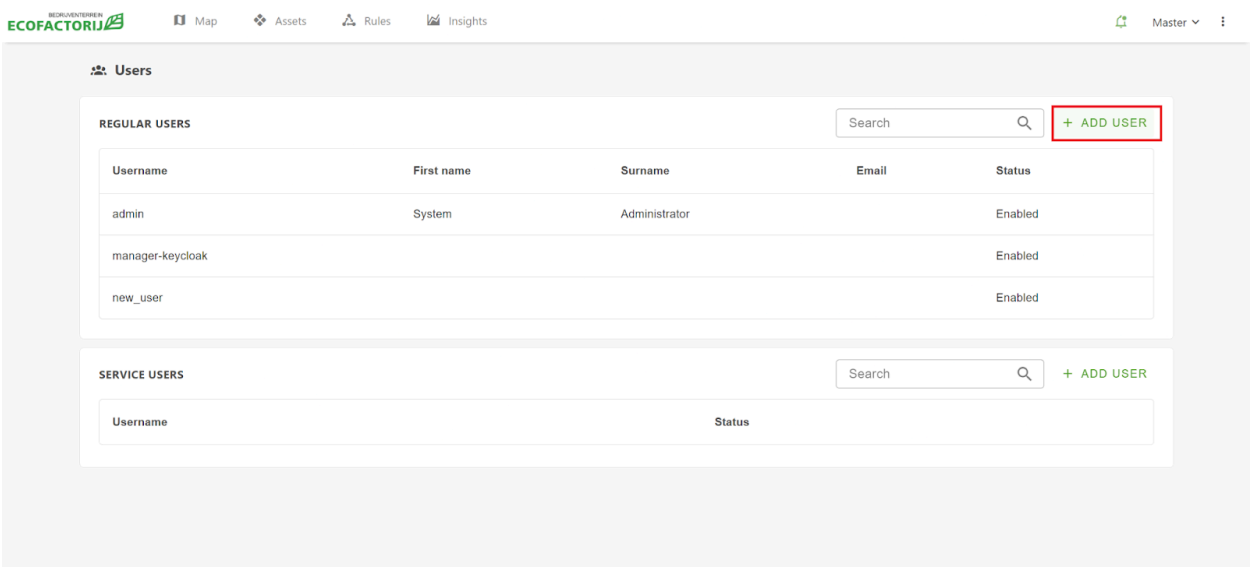


Regular Users & Service Users

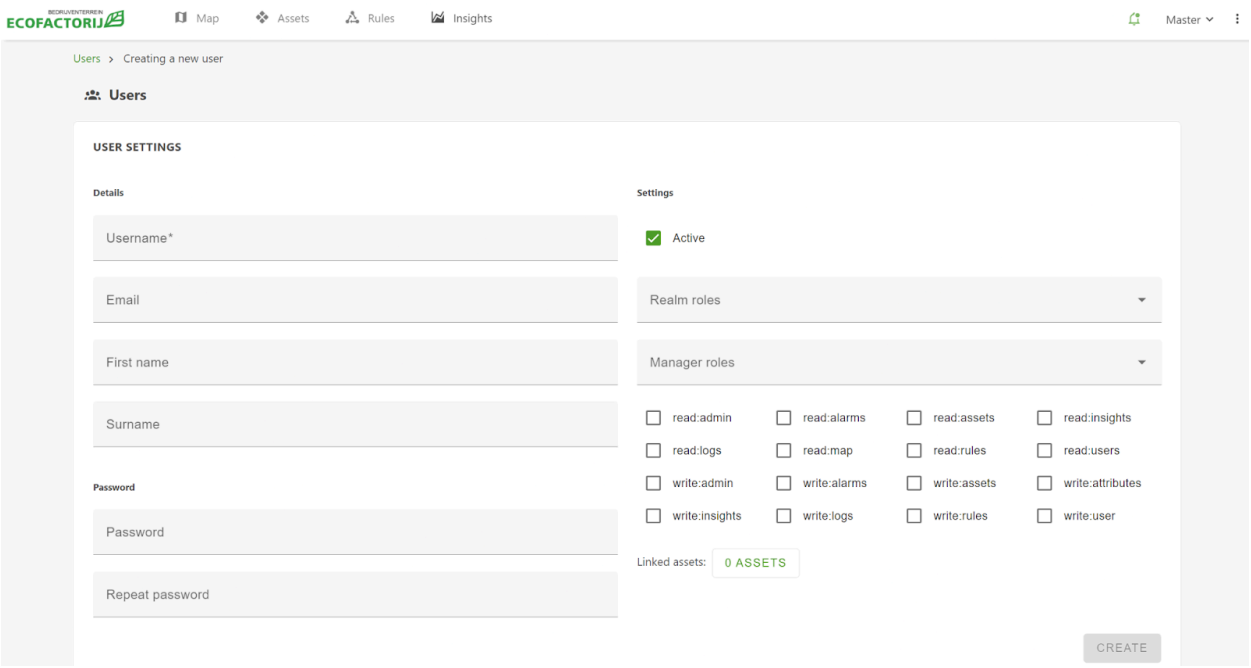
OpenRemote allows the creation of two types of users, Regular Users and Service Users. Within the context of our application, Service Users have no purpose. All new users should be created as Regular Users to avoid issues.

Creating New Users

To create a new user simply press + ADD USER



You will be forwarded to the following page where you will have to insert the user credentials. The system only requires a username and password for an account to be functional.



Permissions

Before creating a user it is important to set their permissions accordingly. If the user is intended as an administrator or manager, perhaps giving them write permissions to certain functions is necessary. This is where you decide how much control to give the user.

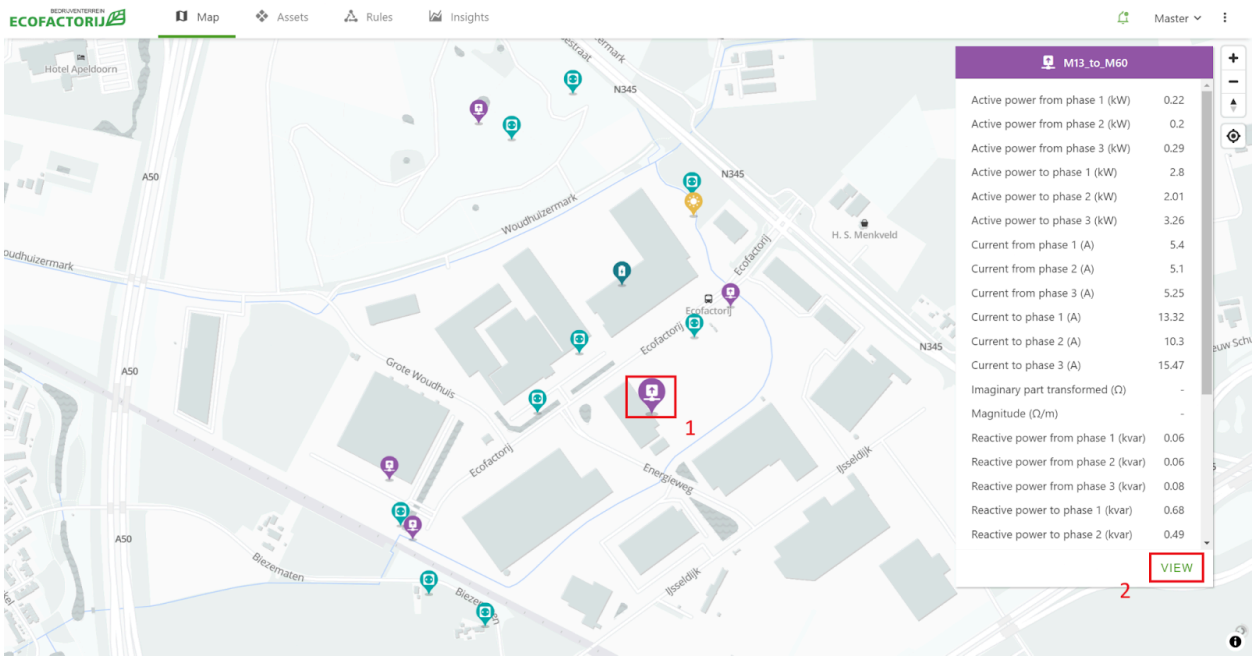
When you're done, simply press CREATE at the bottom right to save that user in the database.

Map

The user interface features a map of the Ecofactorij area. It provides an aerial representation of the property with coordinates resembling the different meters on the premise. Users may press on their desired meter and a window will appear on the right side, showing an overview of its attributes.

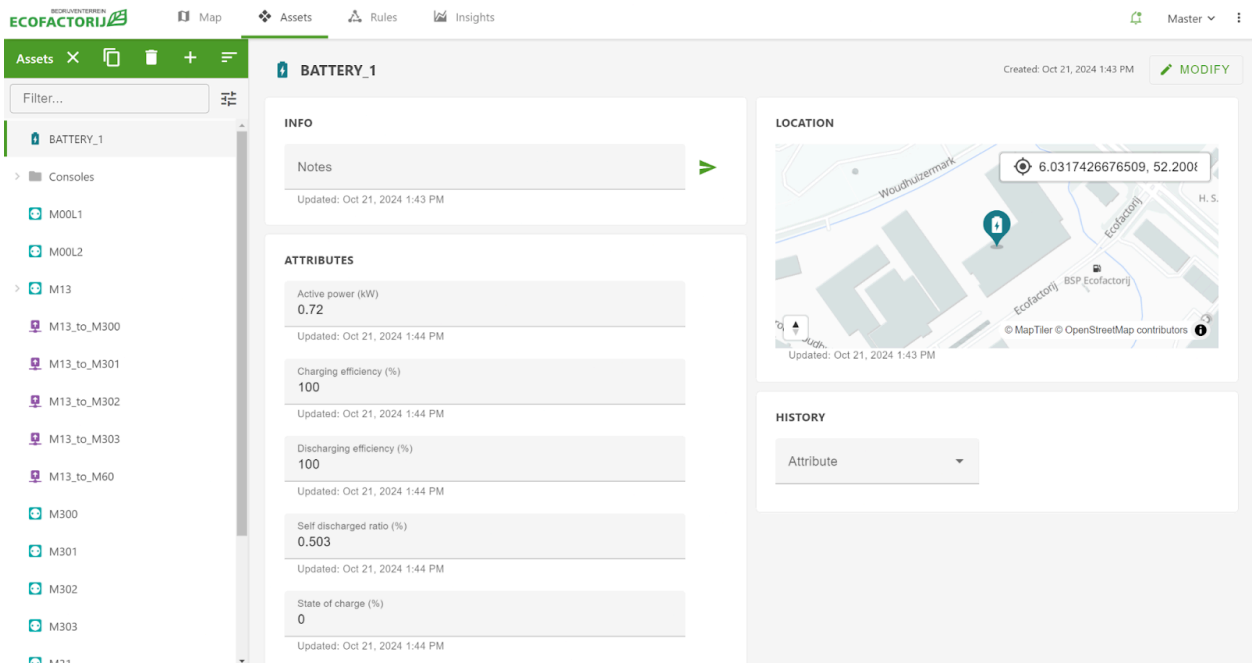
Viewing Assets

To get more information about that meter, users can press on VIEW which will direct them to the assets page. (Refer to the screenshot below)



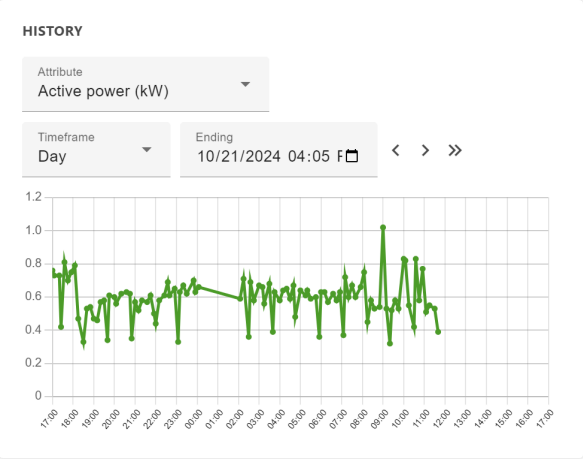
Assets

The Assets page allows users to view a more detailed overview of the asset's attributes. It also allows users to generate a line chart, visualizing the historical data provided by each meter.



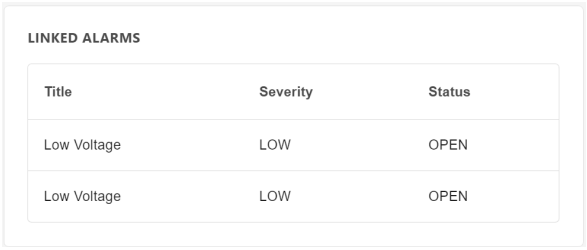
Generating History Graphs

To view historical data, select the attribute that you'd like to visualize. An interface will appear that will allow you to choose specific timeframes and dates for which to provide visualized data. Please note that the date only sets the end date. Meaning if for example, you'd like to see the data from the last week, select today's date and change the timeframe to WEEKLY.



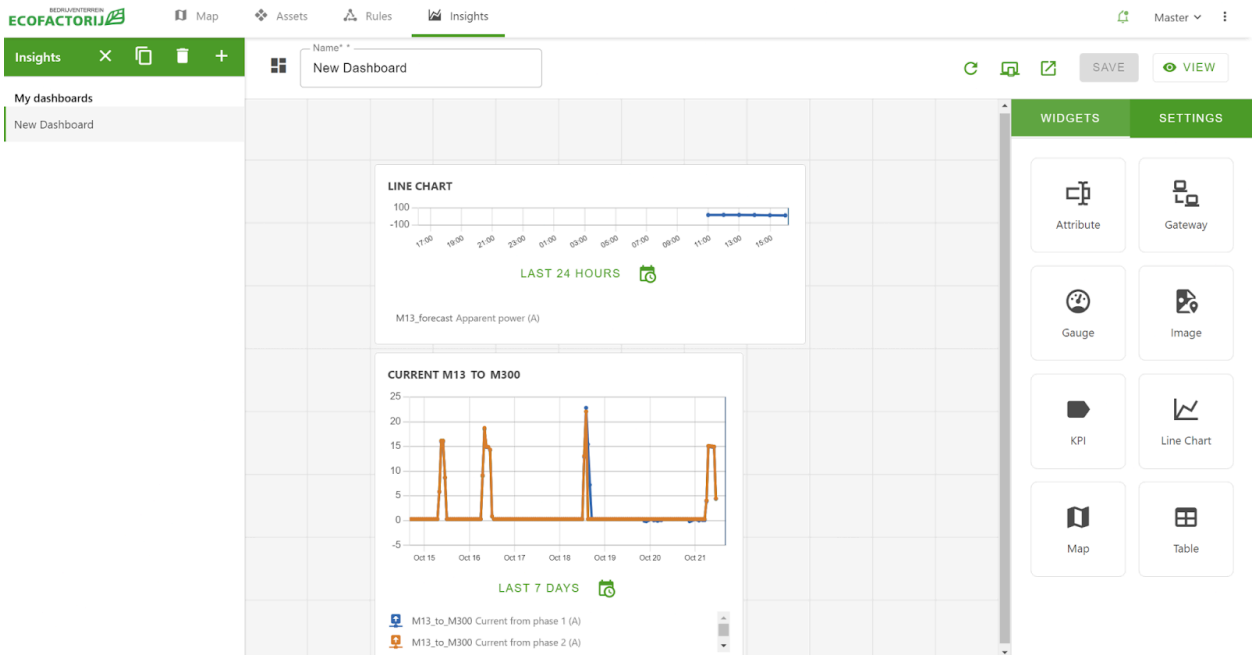
Linked Alarms

Under History, some assets will have a section titled LINKED ALARMS. In case any alarms have been triggered by this asset, they will show up in this section, ranked by severity.



Insights

This page allows users to create their own dashboards by adding widgets.



Setting up a Dashboard

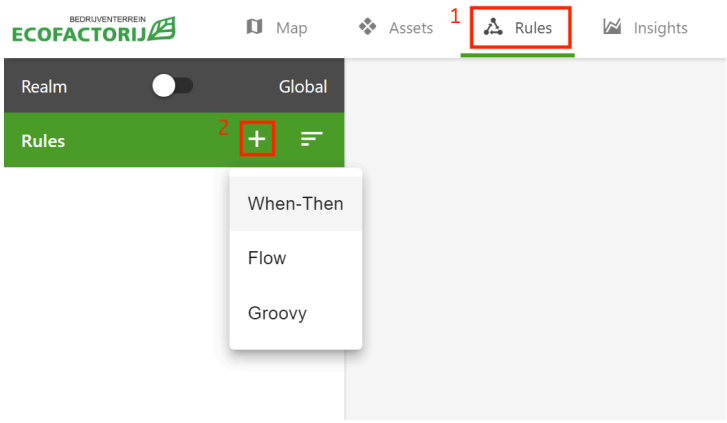
Users may press + to create their own dashboard. In the dashboard, widgets can be added by dragging them onto the dashboard.

Forecast

Users may see the forecast for an asset by following the process below:
Drag the Line chart widget onto the dashboard -> Select attribute forecast for specific asset -> Click show error band -> Set time to next 3 days

Rules

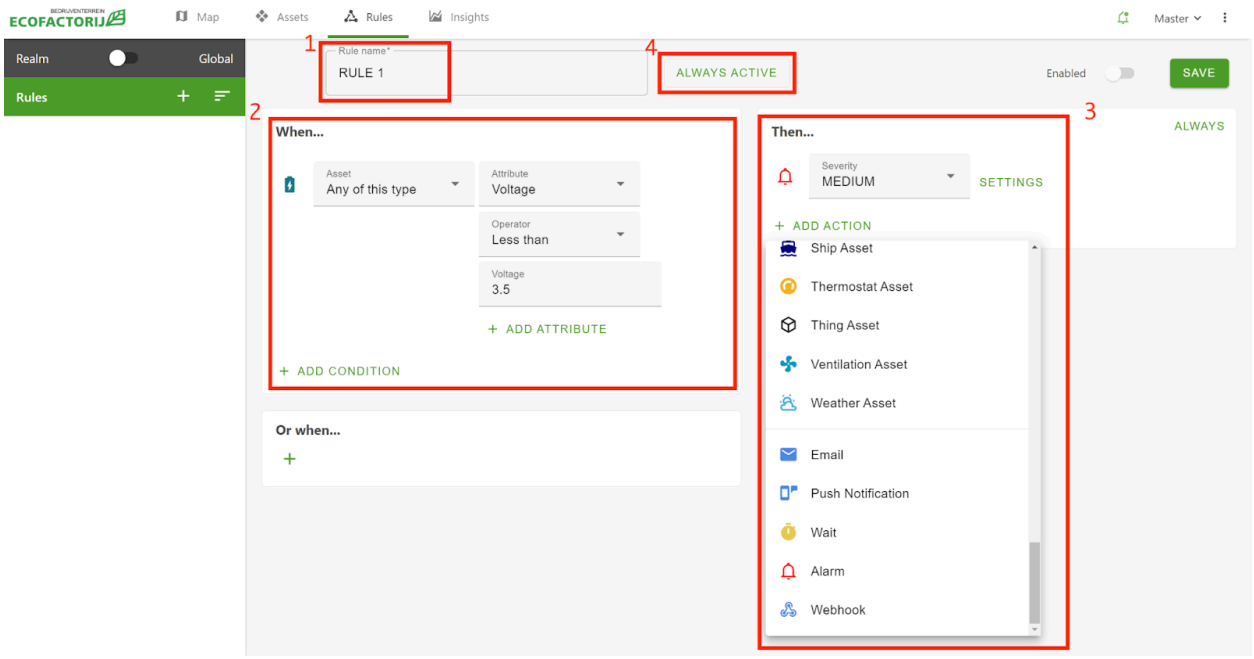
The rules page allows users to view/create rules. Users with read:rules or write:rules permission may view/create rules. To navigate to the rule page, click on the rules at the top of the page.



Create Rules

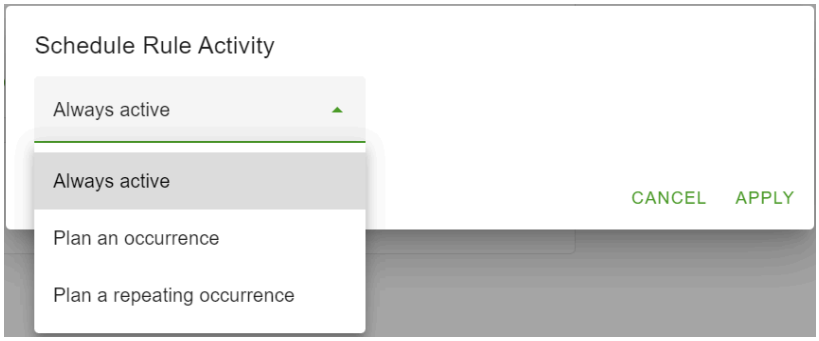
To create a new rule, select + and select the type of rule that the user wants to create.

When-Then Rules



In the When-Then rule, users may set a rule with When (the condition) and Then (the action when the condition is met) attributes. In the When section, the user may select an asset, attribute, operator, or voltage that the user wants to have as a condition. In the Then section, the user may select the action the user wants to perform when the condition in the When section is met.

By selecting the ALWAYS ACTIVE section, users may choose the rule activity from always active, plan an occurrence, or plan a repeating occurrence.



Flow Rules

Flow Rules are intended for application users to perform attribute value conversions. The main purpose is to enable the linking of attributes (e.g. a Battery with a PV), or to process attributes to generate new 'virtual' attributes (e.g. calculating energy consumption as the sum of three individual meters). For more details, see Flow Rules.

Groovy Rules

The Groovy Rules scripting editor can be used for complex rules. They have the most flexibility, but also need a clear understanding of the Groovy language, especially to avoid mistakes. For more details, see Groovy Rules.