ENERGY MONITORING SYSTEM

Introduction

The project proposes an energy monitoring dashboard where the user can retrieve data from a server and view devices and their consumption. In our highly connected world, users need to keep track of their Internet of Things Devices. This project proposes a solution for managing a database of such devices. This system has two components:

- Front-end user interface
- Back-end server and database

Besides the two components, the system must also accommodate two types of users:

- Admin users
- Normal users

Front end

The front-end part of the application consists of a web app. Users should be able to create an account and log in. Once logged in, each user will be directed to a particular page depending on their role. If the user is an admin, he will be directed to the admin dashboard where he can view all users. The admin should be able to perform crud operations on users and devices. He must be able to manage both devices and users as well as to create associations between the two groups. When a device is associated with a user, the user can see the device in his device list. Most importantly, the admin should be able to add and remove devices, as well as users. He is also able to modify the email address of a user.

The user on the other hand will be redirected to the user page. He is more constrained in the web application, being able to only read data. Once he logs in, he should be taken to the user page where he should be able to view his device list. He should be able to select one of such devices to view details about it, such as id, description, address and maximum hourly consumption.

Besides the data about the device, he should also be allowed to view a graph of energy consumption. He should have the option of choosing the date he wants to view the consumption on, and once he does so, he will be shown each level of energy consumed every hour on that day.

Back end

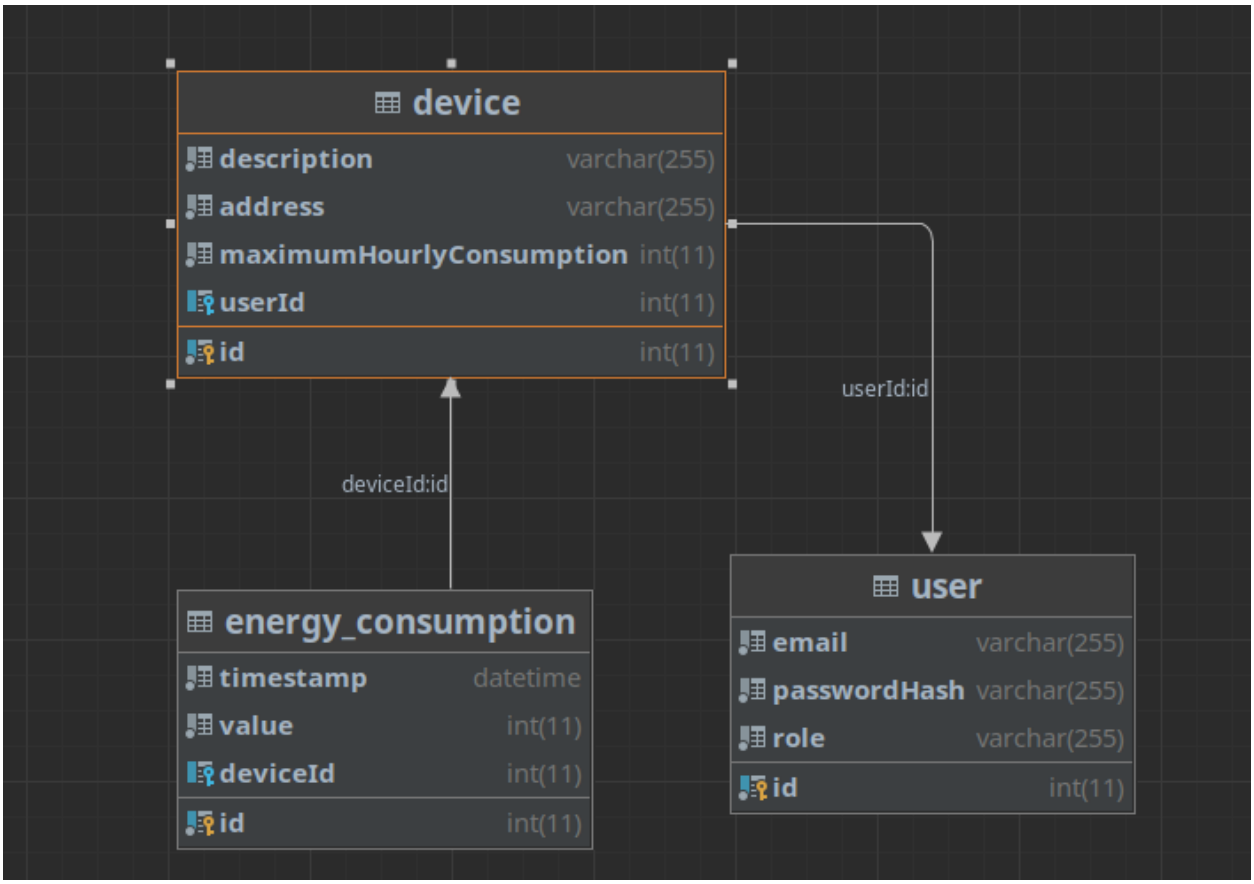
The back end of the application is a web server that should expose a number of endpoints to allow both user groups to perform their desired actions. These endpoints should provide functionality through web requests. There should be a set of CRUD endpoints for each resource in the backend, i.e. user, device and energy consumption. The endpoints should perform queries on the application database to retrieve the requested data or make changes to already existing data. Some endpoints should be accessed only by users, while others only by

administrators, therefore there are special endpoints for each group, like retrieving all users for administrators or retrieving device energy consumption for a specific device for user.

Implementation

Database

The database of the project was implemented using a MySQL database server. In it, three tables are created to hold information on our users, devices and energy consumption. Between these entities, there are relations. Each device has associated an user or no users. The devices without any users are referred to as orphans. Each user, therefore, has multiple devices. To each device in turn there is a set of energy consumption objects associated, representing the data gathered from the device. Each energy consumption entry has a timestamp and a value representing the consumption in kilowatts, therefore each device has multiple energy consumption records.
Each entity in the database has also a unique id associated to it. The database schema looks as follows:



Back-end

The backend server of the application was realized using Nest JS. Nest JS is a modern framework for creating REST applications, microservices, graphQL APIs and many more. It is powered by Typescript and has many very well-implemented plugins for a number of Typescript/Javascript packages to achieve multiple common tasks. Its design is a three-tierd architectural pattern, consisting of a controller, service and data access layers.
In the implementation, multiple NestJs plugins were used:

- Passport

Passport is a javascript library used for authentication and authorization. It implements the strategy pattern in order to assess the access of users. Among these strategies were local and jwt authentication.

- TypeORM

TypeORM is a fully featured ORM for typescript. It works by mapping each entity to a typescript object and combines the powerful typescript feature of decorators to achieve ease of use when defining columns, keys and relations. It can be used in both active entity pattern mode and repository pattern mode. For this project, the repository pattern was used. TypeORM can infer relations and features of entities in your app and automatically generate repositories that simplify common queries like filtering, pagination, creating, updating and deleting records. This was the main ORM in the application and serves as the entirety of the data access layer.

The service layer is facilitated by the powerful dependency injection engine of NestJS, which compartmentalizes each resource as a module which exposes controllers, and providers and imports resources from other modules. Each resource in turn has a Controller, Service and Repository. When needed, modules can import resources from other modules. This way, the class coupling is kept low and the code is very orderly.

The controller layer exposes multiple endpoints that can be accessed from the outside. This tier has the task of mapping requests to their appropriate handlers. Again, using decorators, NestJS can automatically generate these mappings. A controller is just a class that contains a set of functions that are mapped to the router of the web server.

Front-end

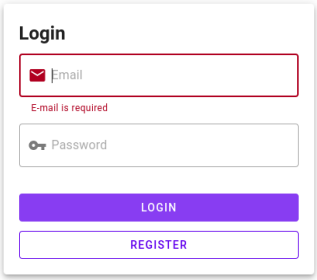
The front end of the application was realized in Vue.js 3 using composition API. The app consists of four pages managed by the main router, allowing for the app to be a single-page application. The state management was done using Pinia stores and the interaction with the backend was achieved using the Axios package. The routing was done with the official Vue router package, which provided many features like re-routing, route guards and parameter extraction.

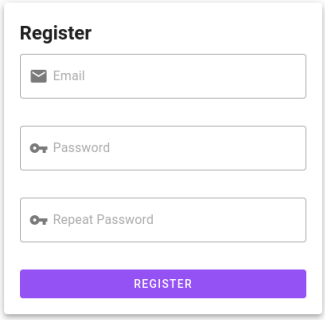
The four pages are:

- User page
- Admin page
- Register page
- Login page

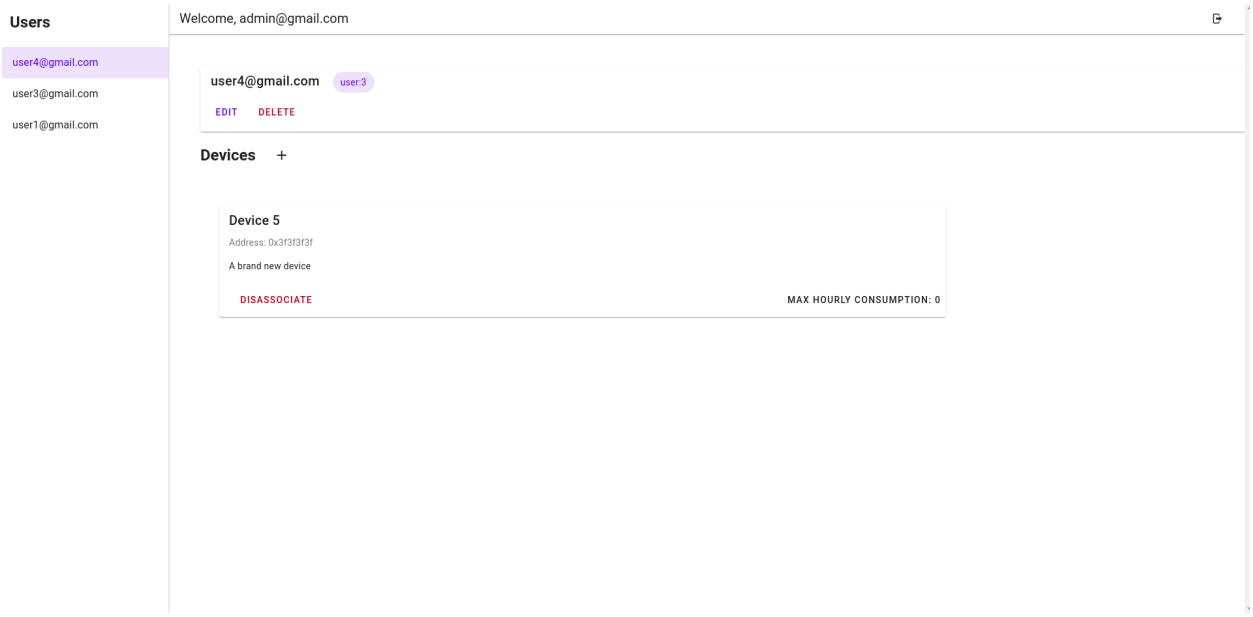
If the user is not logged in, he will be redirected to the login page from anywhere in the app. From the login page, he can access the registration page to create an account. If logging in with an administrator account, he is taken to the admin page where he can manage devices and other users. If he is a normal user, he is taken to his own page to view the devices associated to him.

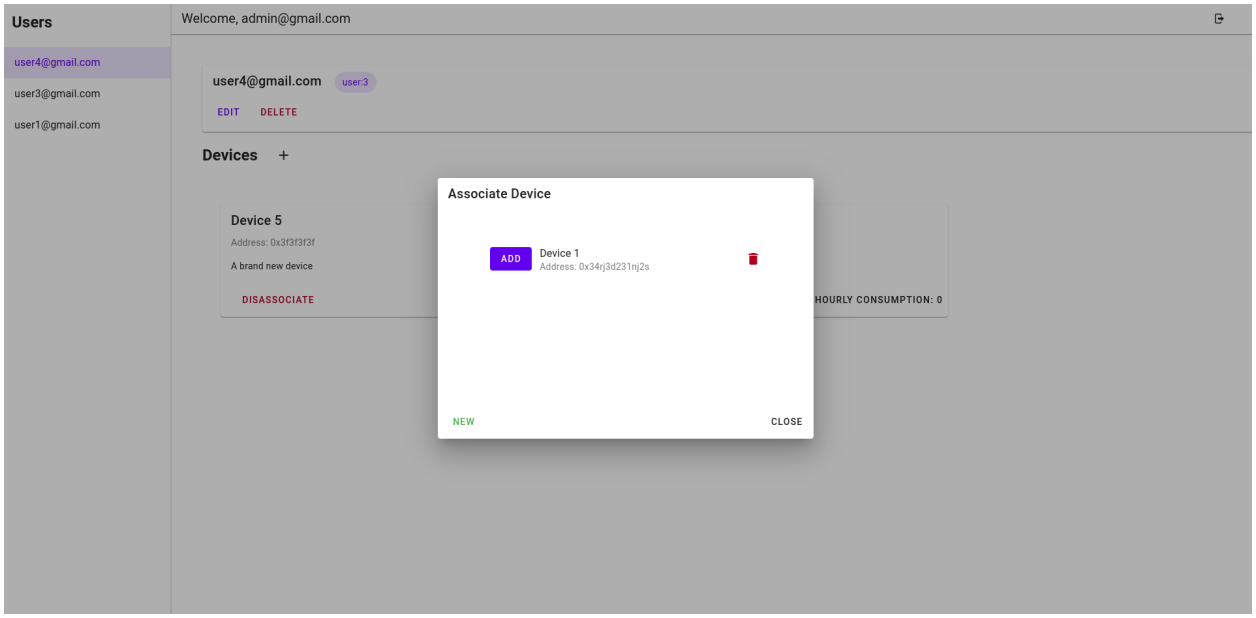
Login Pages:

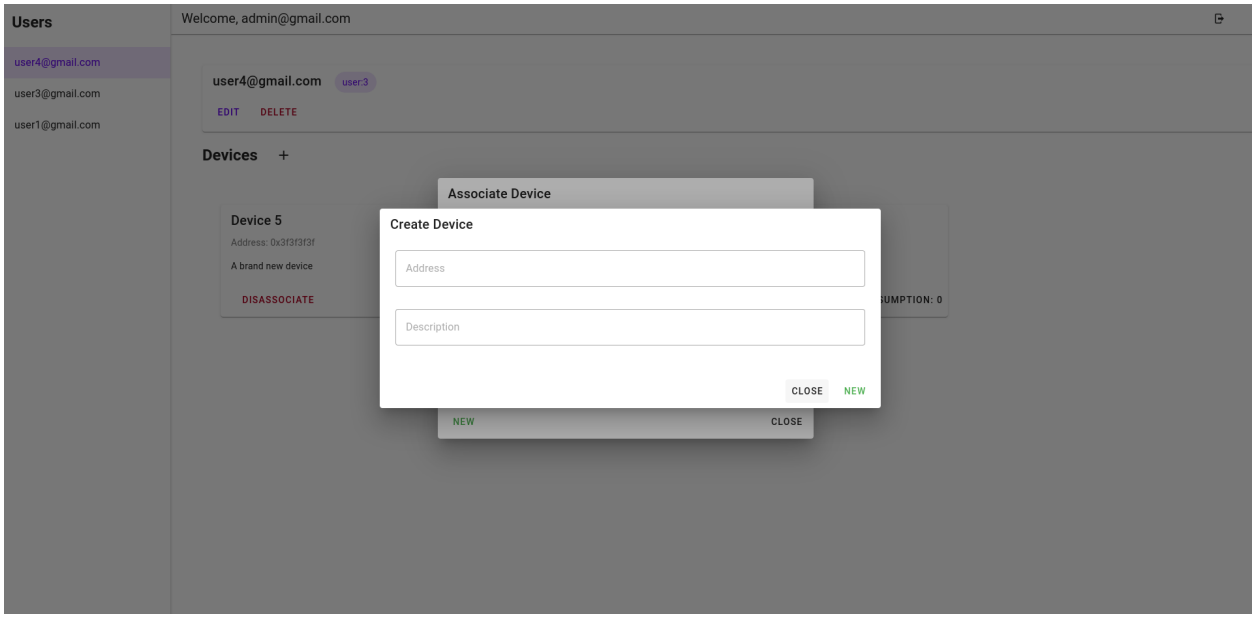




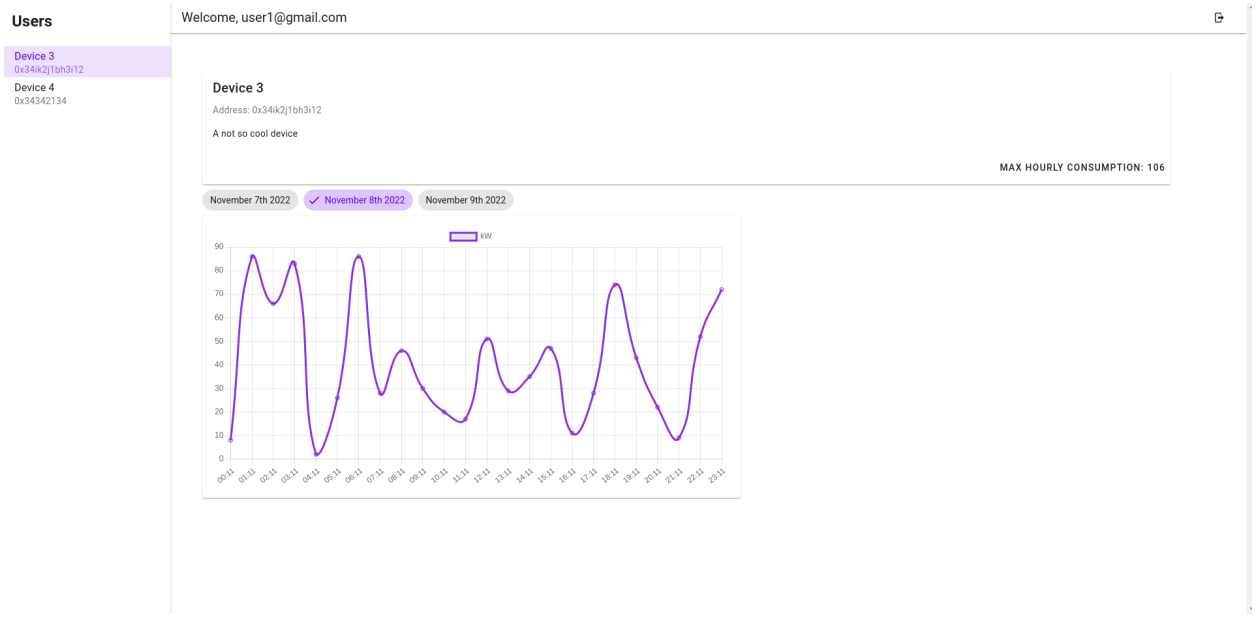
Admin Page:


Users
user4@gmail.com
user3@gmail.com
user1@gmail.com
Welcome, admin@gmail.com
user4@gmail.com
user:3
EDIT
DELETE
Devices
+
Device 5
Address: 0x3f3f3f3f
A brand new device
DISASSOCIATE
MAX HOURLY CONSUMPTION: 0


Users
user4@gmail.com
user3@gmail.com
user1@gmail.com
Welcome, admin@gmail.com
user4@gmail.com
user:3
EDIT
DELETE
Devices
+
Device 5
Address: 0x3f3f3f3f
A brand new device
DISASSOCIATE
HOURLY CONSUMPTION: 0
Associate Device
ADD
Device 1
Address: 0x34rj3d231nj2s
NEW
CLOSE


Users
user4@gmail.com
user3@gmail.com
user1@gmail.com
Welcome, admin@gmail.com
user4@gmail.com
user:3
EDIT
DELETE
Devices
Device 5
Address: 0x3f3f3f3f
A brand new device
DISASSOCIATE
Associate Device
Create Device
Address
Description
CLOSE
NEW
NEW
CLOSE

User Page:


Users
Device 3
0x34ik2j1bh3i12
Device 4
0x34342134
Welcome, user1@gmail.com
Device 3
Address: 0x34ik2j1bh3i12
A not so cool device
MAX HOURLY CONSUMPTION: 106
November 7th 2022
November 8th 2022
November 9th 2022
kW

Sensor asynchronous communication

Front-end to Back-end Communication

The front-end and back-end communication in our application is implemented using two methods. The first method is simple web requests, where the front-end requests resources such as information about users, devices, and device consumption. This method is sufficient for most applications, but it is not suitable for our monitoring system because we need to have bi-directional communication between the client and the server.

To enable bi-directional communication, we also integrated WebSockets into the application on both the front-end and the back-end. We used the Socket.io javascript library to implement this integration. This allows us to use request-response patterns while still enabling bi-directional communication between the client and the server. This is important for our monitoring system because it allows us to receive updates from the server in real-time, without needing to constantly poll for new information.

To integrate it in our front-end, we used the already defined Socket.io client npm module and we subscribe to two main events: the notify:userId event where userId is the id of the currently connected user, and device:deviceId where deviceId is the currently open device in the application. This allows us to listen to notifications from our server and also ensures that only the currently logged-in user with his current device can listen for updates.
On our back-end server, we imported the Socket.io server module and defined a gateway class which will handle all communication through the WebSocket. This gateway can later be injected into other parts of the application and it will expose functions for sending notifications to users, as well as sending device updates.

Now, instead of having to refresh the page to get new data, the front end will listen for notifications for the currently logged-in user, as well as for updates for the current device. These updates will be broadcasted the moment a new value is recorded for the device from the back-end and will be immediately plotted on the user’s energy consumption graph. This way, the application is more interactive and the user can see in real-time the consumption of his devices.

Smart energy metering device to Back-end Communication

Inside our application, we need to concurrently gather data from several smart devices running independently from the server. To aggregate and register all the energy readings from these devices, we used asynchronous communication, a message-oriented middleware service.

For the implementation of this message-oriented middleware, we chose rabbitMQ. RabbitMQ is a queue-based messaging middleware in which there can exist two types of users: producers and consumers. The producers will post messages to exchanges which will get redirected to queues inside the rabbitMQ service. After these messages are published, the consumers who

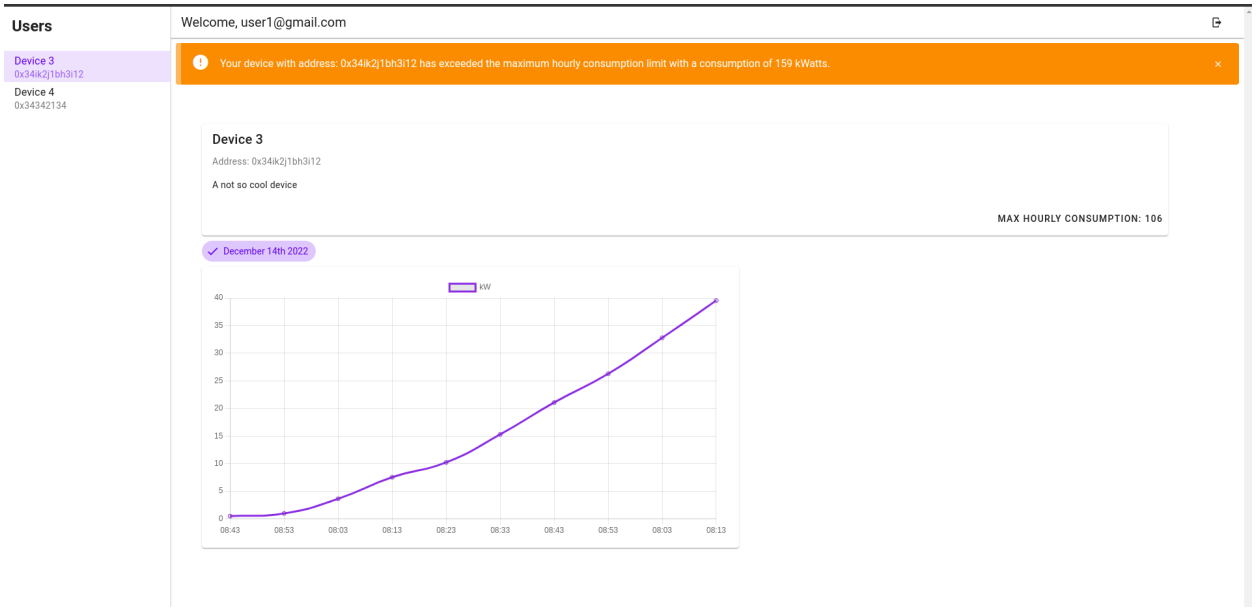
listen to these queues will get notified and will receive the message, thus consuming it and dequeuing it.

In our particular implementation, we used a single queue, called device_consumption. Multiple devices will be able to publish to this queue. The publishers are the smart metering devices, which are given a particular device ID. With this device ID, they can publish readings in the queue, where they get handled and processed by our consumer, in our case the back-end server, which will register them in the database and will then notify any front-end application that listens for updates on that particular device.

To model this behaviour, the smart metering devices are simulated using a python script that reads values from a CSV and published them to the rabbitMQ queue for processing. This mock device will be started and will receive as a parameter, either from the command line or from a dedicated environment file, the device ID with which it can be identified by our server.

Each metering device will publish a tuple containing the current Unix timestamp and a floating point value representing the energy consumption in kilowatts, as well as the device id used to identify the device. This JSON object will be published in the default exchange which will get automatically routed to our device_consumption queue inside the rabbitMQ message broker.

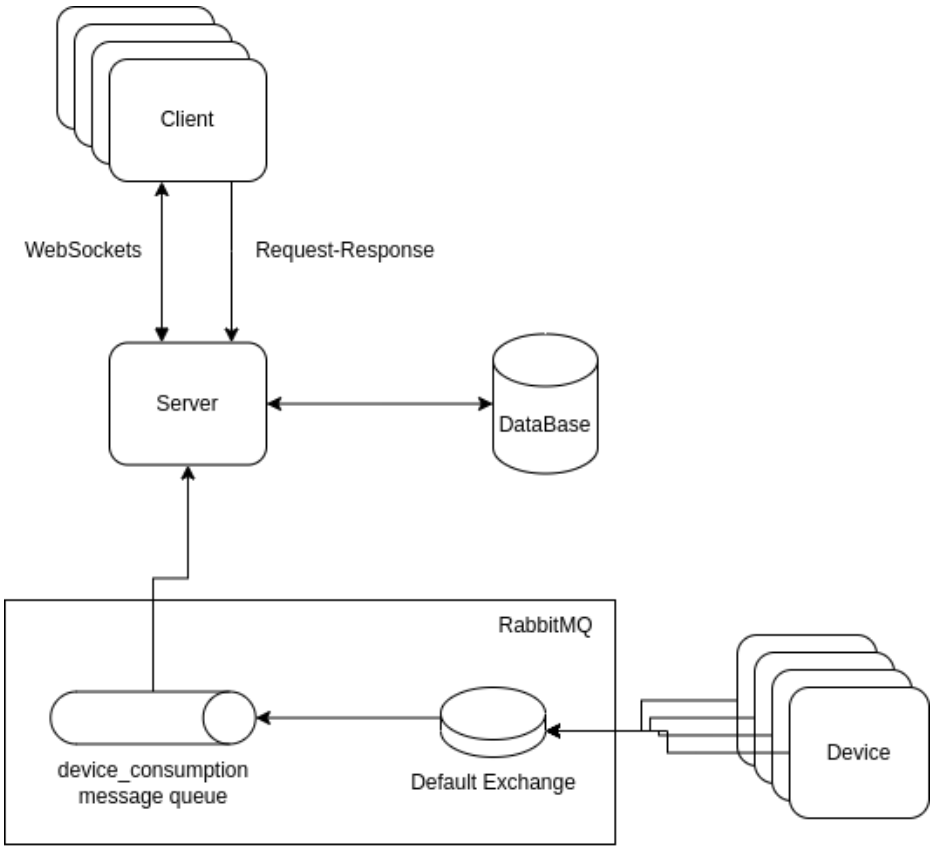
From this queue, the messages will be read and recorded in the database. For each new message, a query will be performed to count the total power consumed in the last hour of functioning of the given device. If the consumption exceeds the set maximum hourly consumption, then the user will be notified of this through a yellow banner appearing inside the web app, like so:



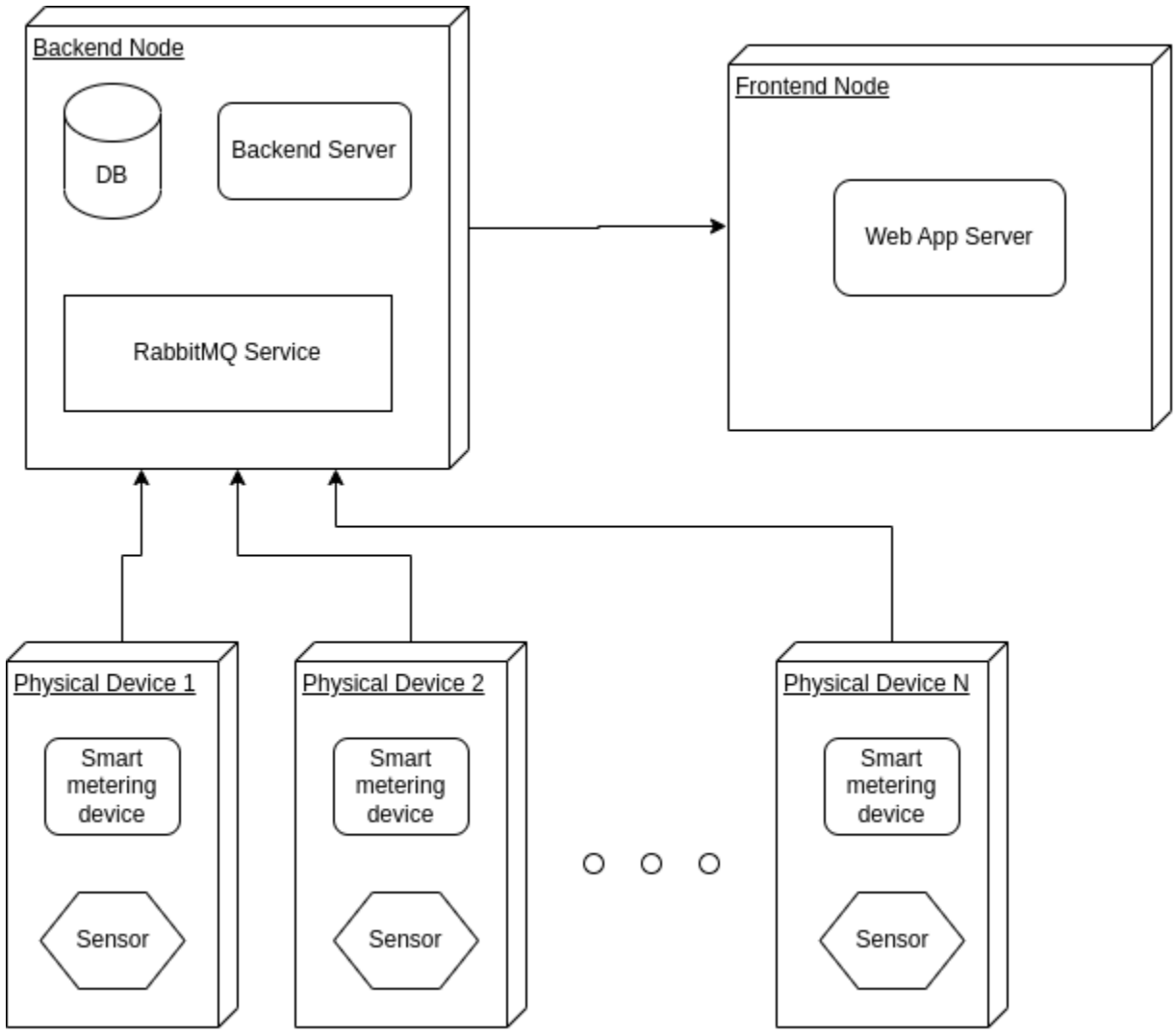
The architecture of the system

Here, we can see the overall architecture of the system. The arrows between elements represent ways of communication between the services and processes of our application:

- Clients (Front end website)
- Server (Back end server)
- Database (MySQL database)
- RabbitMQ middleware (message queue)
- Devices (Python clients publishing messages to the message queue)



UML Deployment diagram



gRPC powered chat service

We are to implement a chat service that makes it possible for two users to communicate on the platform. The functional requirements for this service are

- A user account to be able to communicate to an admin account.
- The admin account to be able to communicate with multiple users
- When typing, a message should be displayed to the conversation partner.
- When a message was opened, it should be indicated to the one who has sent it
- When a new message appears, both the user and the admin should be notified visually.

For the implementation of this system, a gRPC communication scheme was used. A common stub defining the functions and data types of the system is given to both the client and the server. The two entities than can make use of the API to send messages to each other.

Design

The interface between the client and the server present three main functions that which used in conjunction with each other will implement the full functionality of the server:

- **postMessage() -** which will take a **TextMessage** message and will return a **Result** message.
- **getMessages() -** which will take a **Client** message and will return a **Result** message.
- **getUpdateStream() -** which will take a **Client** message and will return a **Stream of TextMessage** back to the client once called.

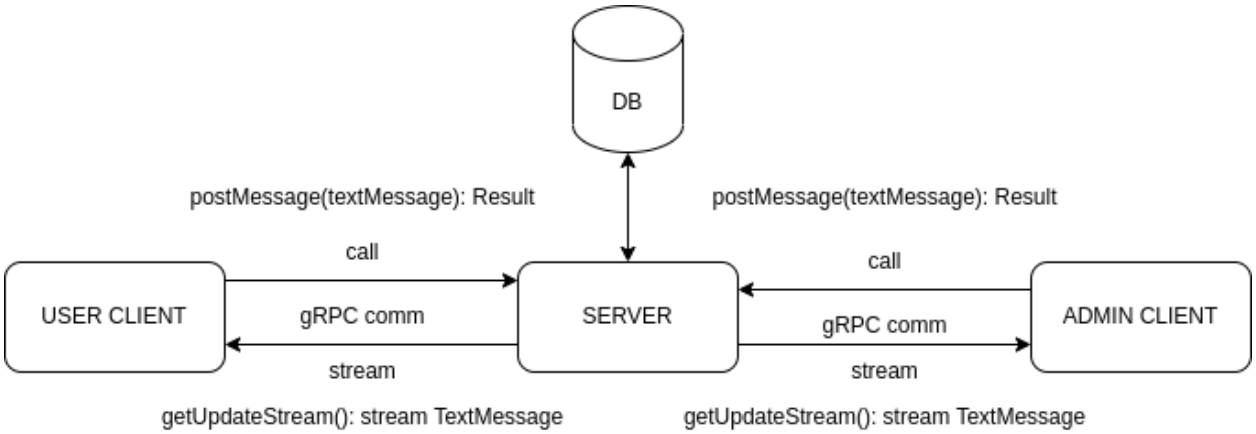
The three functions work together in the following flow. The user will log into the application. Once logged in, we can now make use of his id and embed it in a Client message which has only an id attribute. When the app is being initialized, it will call the getUpdateStream() method which will return a stream of TextMessages. On this stream, we can retrieve all messages from the server, as well as receive a new message in case one of the other users texts us. Once we have this stream up and running, we will listen for any updates coming down from it and we wll add them to the appropriate collection.

After establishing the stream, we will then call the getMessages() method which will return a result message telling us if the function ran correctly or if an error happened. If the function ran correctly, then from the server the client will start receiving a stream of messages. These messages are all the user messages, i.e. the messages sent by him to anyone else, or those which were meant for him. These messages get sorted into their appropriate bucket, identified by the id of the sender/receiver of the message which is different from our client.

However, the getMessages() function will not directly return the messages concerning the client, but in fact will only return the status of the function. Those messages will actually be returned down the previously opened stream.

In this architecture, the stream opened in the initialization of the app will act as a one-way communication bus between the server and the client. Any updates or history of conversations will be distributed to their recipients through this only stream of TextMessages. That includes the acknowledgement messages when the user reads a message, as well as the typing and stopped-typing messages.

To publish to other users' streams, however, a user should call the postMessage() method with the message they want to publish. Once called, the message will be recorded in the database and will be then distributed to the two streams of the users involved in the conversation. This is done in order to tag each message with a special id.
When initializing the client, a server will receive the id of the currently logged-in user who is trying to get his update stream. With this id, it will keep an internal record of all currently connected users. In this record, the reference to the stream which was created to communicate to the client will be kept and then later used to broadcast updates.



When talking about the acknowledge read messages as well as the start typing and stop typing messages, the TextMessage has a field called type, which is an enum denoting 4 possible message types:

- CONTENT
- TYPING_START
- TYPING_END
- ACK

The content type will contain a message to be recorded in the database. Once recorded, it will be broadcasted to the two conversation partners who own it.

The typing start message will only be broadcasted directly to the recipient of the said message. No typing-type messages will be recorded in the database, as they are only ephemeral messages that indicate an action which will end soon and do not create a new resource on the client, thus not changing the server state.

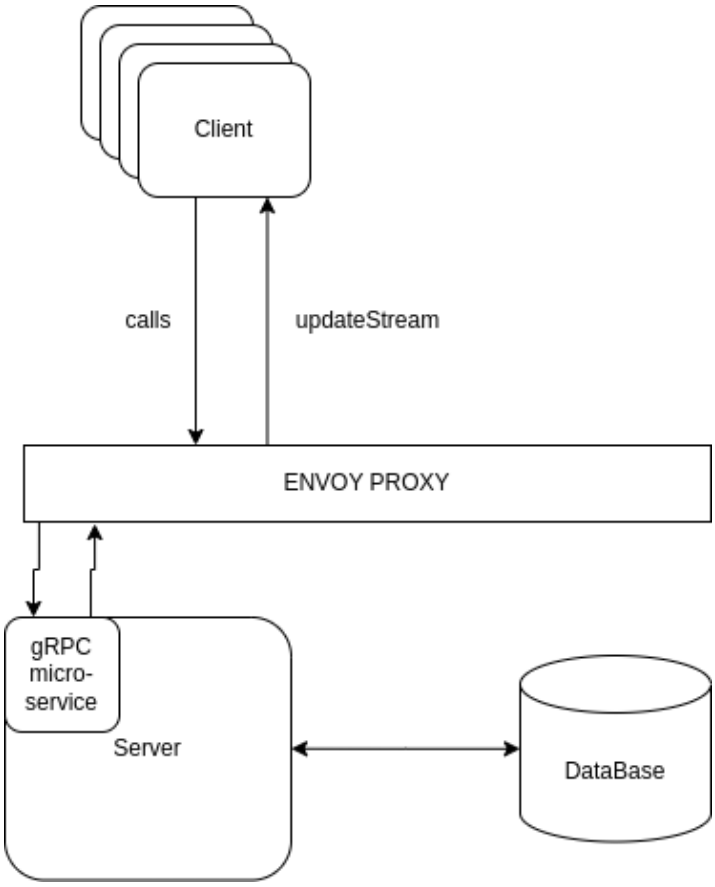
The acknowledgement message will be broadcast to the server and will change the state of the database. The acknowledgement message comes with a message-id which is the id of the message being acknowledged. Once received, this message will cause the server to update the message record in the database in order to reflect the fact that the recipient has actually received the message and can be shown as seen. This also publishes the received message down on the stream of the author of the message, thus he can notice changes in acknowledgement and update the view accordingly.

Implementation

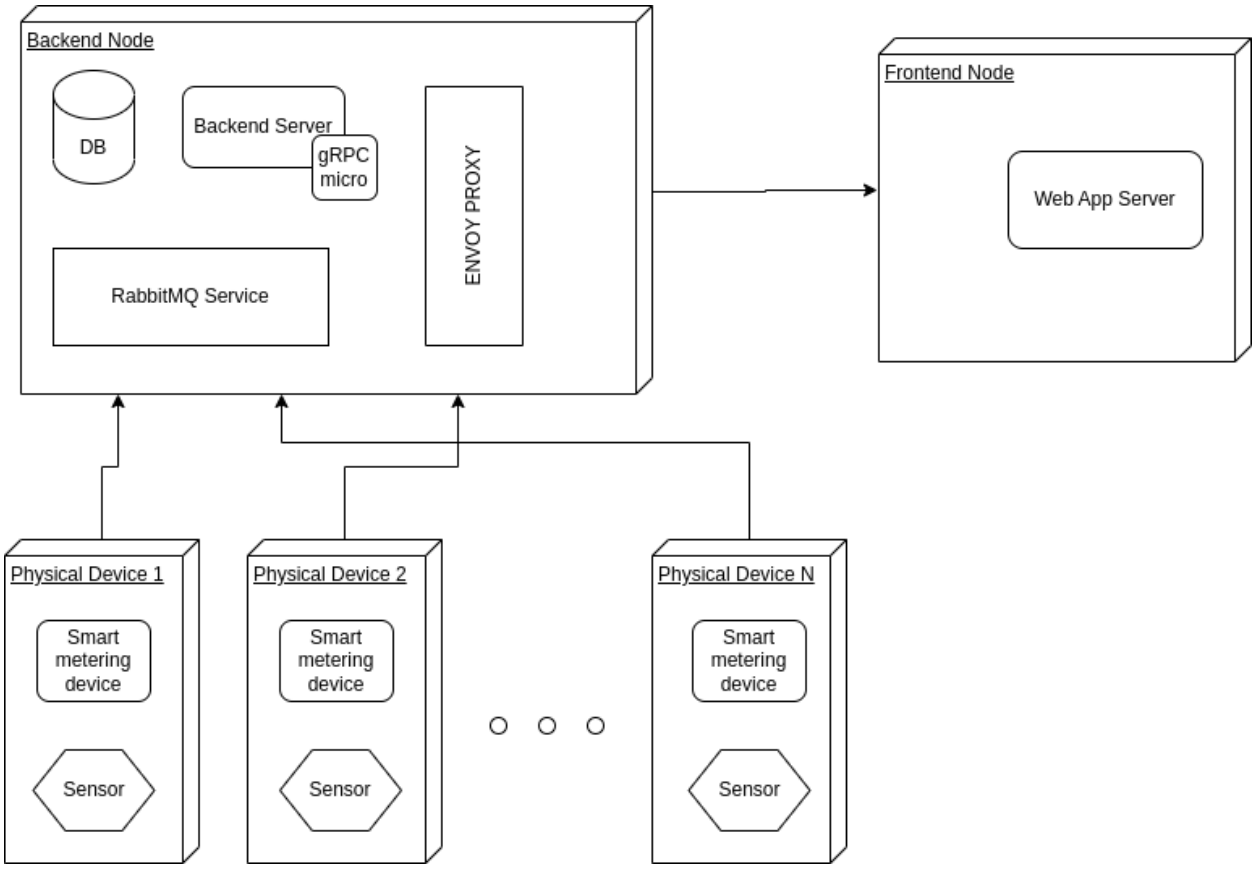
The connection of the client to the server was realized using the gRPC-web package. To have the stub and then generate typescript code which performs the grpc calls, the ts-proto package was utilized. Upon running a script, a file containing all the data classes and a grpc client which has all the functions of the messaging service is generated. However, javascript that runs in the browser does not support all control features of HTTP/2 required to run gRPC from the browser and for it to directly call the services in the backend. To evade this limitation, a proxy server was set up to take the client messages and then forward them to the server serving the gRPC protocol.

The proxy server used is called Envoy, and it will be hosted alongside the backend in the docker-compose file.

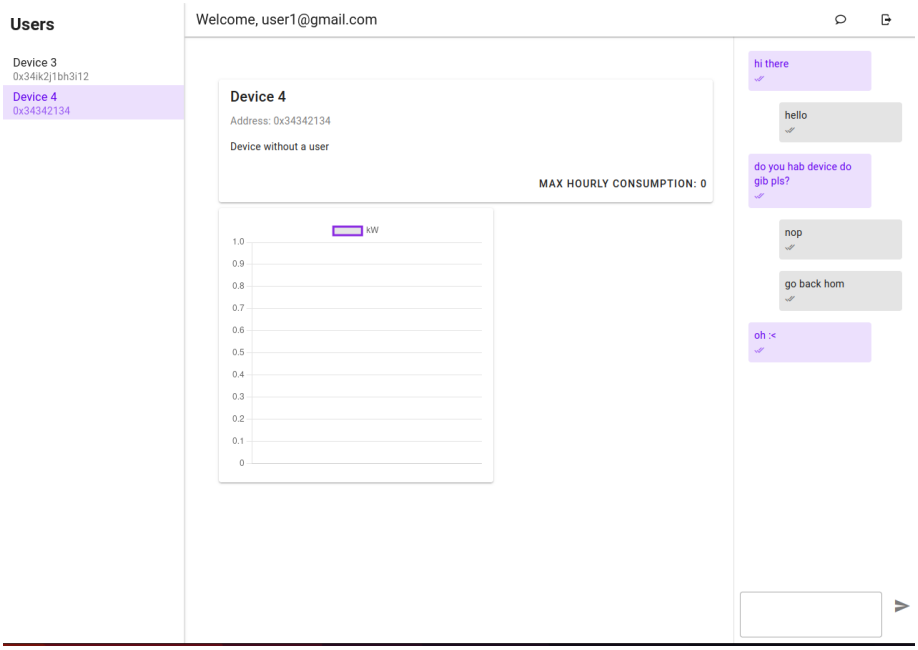
On the back end side, NestJS offers microservice support by default, and that includes dedicated packages for receiving and calling gRPC-ready microservices. To enable our back-end server to run with gRPC, we follow a pattern made by the creators of the package and spawn a child microservice which acts as the entry-point for all gRPC calls. This microservice is served alongside the server, however, it receives messages on a different port. The microservice however has access to all the environments of our server application and is under the dependency injection umbrella of NestJS. Therefore from its services, we can make calls to other parts of our code as well as to the connected components of our backend, like the database and the queue.



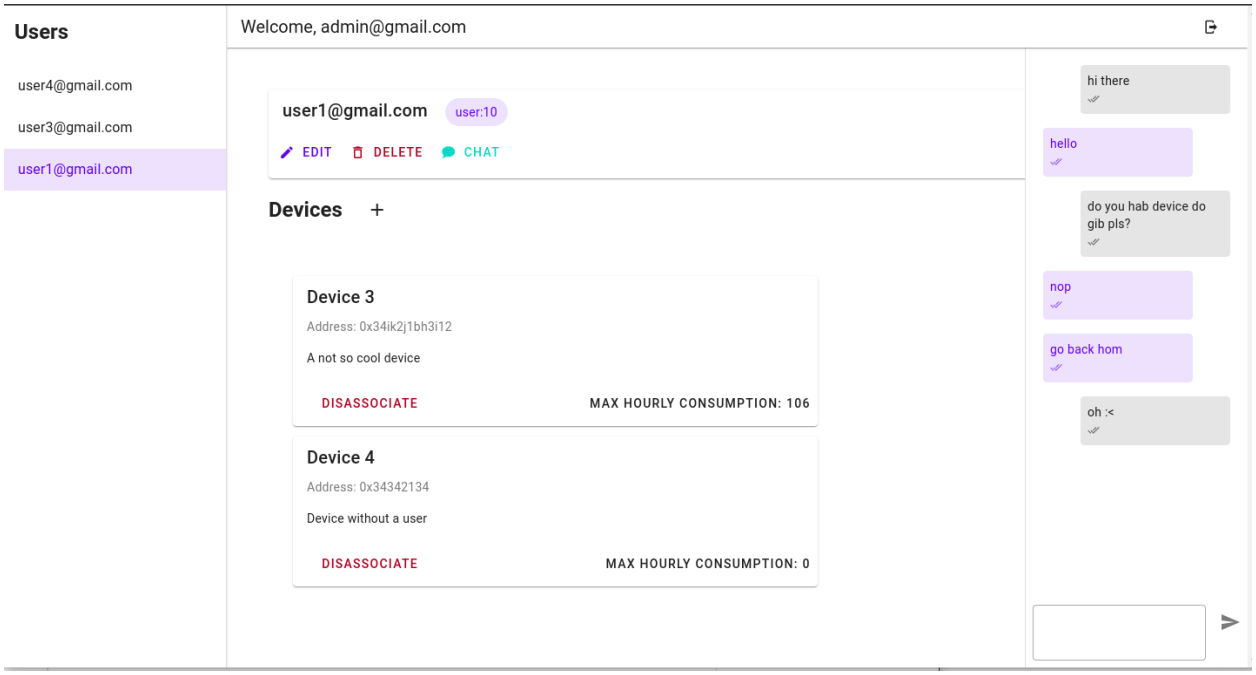
The deployment diagram is as follows:



The interface of the user will be unmodified except for a button that will open a side menu with the messages sent to the admin:



The interface of the admin will be also not majorly modified. Every user card that shows when selecting a user will have a chat button that will open the side chat. When a user sends an admin a message, a red notification shows next to the user.



The chat window changes depending on the selected user.