M7011E
Design of Dynamic Web Systems
Project report

by

Tenta Noobs

Martin Terneborg — 960902-9138
`termar-5@student.ltu.se`
Hugo Wangler — 970818-1194
`hugwan-6@student.ltu.se`

LULEÅ
UNIVERSITY
OF TECHNOLOGY

January 18, 2020

Contents

1 Introduction

During this project a web system for controlling the production and consumption of electricity on a smaller scaled market has been developed. The users of the system consists of prosumers, who can both consume and produce electricity with their wind turbines, and managers operating a coal power plant which also supplies the market with electricity. Prosumers can also choose to direct part of their produced electricity to the market but may also store all of it in their batteries. In order to populate the system with data a simulation has been developed. The simulation simulates data for the registered prosumers based on data model for metrics such as windspeed and power consumption.

2 System

The web system developed consists of two different systems a simulation- and a prosumer- and manager system which together create the overall system. In fig. 1 the system architecure can be seen, here the prosumer- and manager system resides on the server.

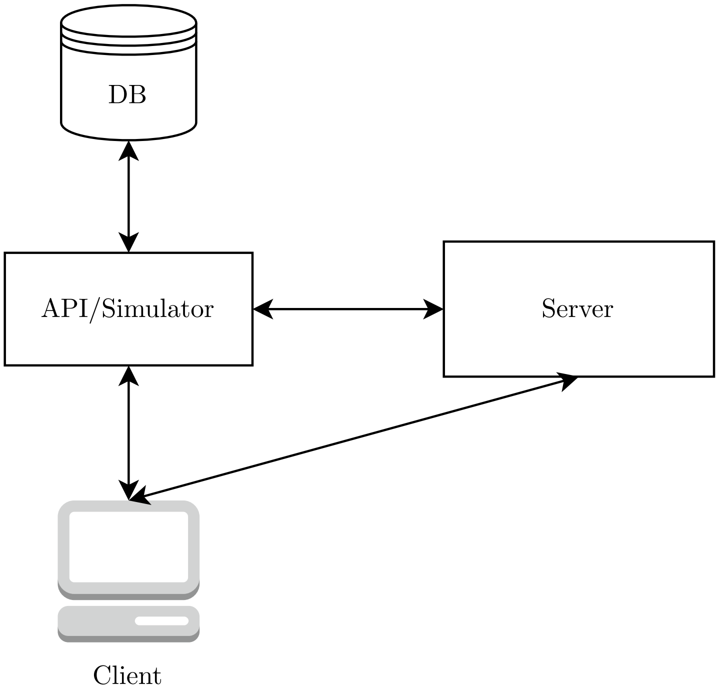

Figure 1: The architecture of the web system.

2.1 Simulation

The simulation system makes it possible to try out the web system without requiring actual wind speed and household consumption data. The simulation is responsible for calculating and governing the production and consumption variables of the system's users. To achieve this, various simulation variables were modelled, for example the prosumer's current wind speed and the wind turbine's power output. These simulation variables are accessible via an API running on the same server.

2.1.1 API

The API was built using GraphQL which is a query language for the API, and a server-side runtime for executing queries according to a schema. The schema itself is a type system defined for the data (see fig. 2). Apart from the simulation variables a lot of other functionality is provided with the API which can also be seen in fig. 2. A small, separate, REST API is also provided on the simulation server to handle image up-/downloads. The data modelled in the schema all originates from an object-relational database using the PostgreSQL database system.

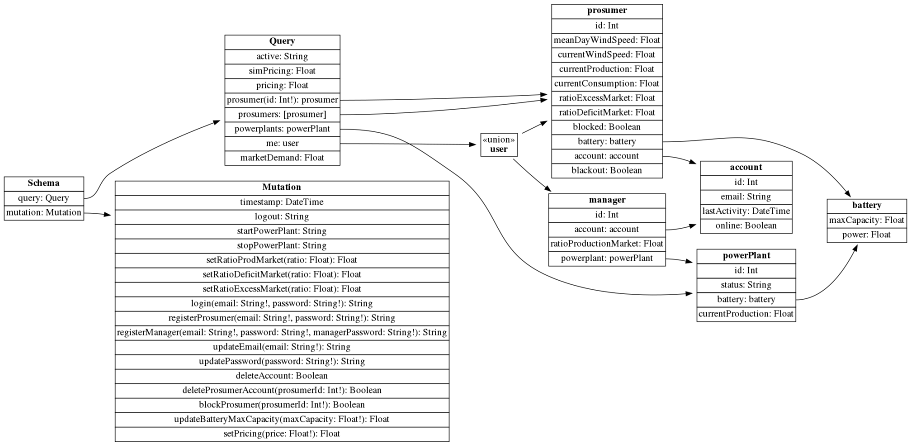

Figure 2: Diagram of the systems main API and its GraphQL schema. Both queries and mutations can be performed.

To secure the API, authentication and authorization is performed to verify that the sender of the request are who they claim to be and has sufficient authorization to perform the requested query/mutation. For example only the manager is able to start his power plant and request data of all the prosumers in the database. This is discussed and explained further in section 4.1.

2.2 Prosumer- and manager system

The prosumer- and manager system exists on a separate server where a web server with Node.js is running to which clients can connect. Using the web application the users can make changes to their systems and perform various functionality. The system itself uses the API provided by the simulation server to perform any changes connected to the data stored in the database i.e. account information. The routing on the web server also authenticates the users and checks for authorization when required.

2.3 Design choices

The overall system design can be seen in fig. 1. The design is influenced by the micro-service architecture, in which different types of functionality of the system are split into small, discrete and loosely coupled components. One might argue that the current architecture follows the micro-service architecture, however it is possible to identify some ways in which the system could be separated into more discrete units. For example, the API and simulator server could be split into two separate components where one simply serves as the API and the other the simulator, this would have a couple of advantages and drawbacks in comparsion to the current architecture.

The advantages would be that the simulator would then have to access the data of the system through the API (since the database only listens to the API's address), like the rest of the components of the system, which means that across the system, all data would be accessed and modified in a uniform way.

Another advantage would be to offload some computational work from the API. Since the simulator runs the entire simulation on regular intervals and sends queries to the database, it might have a negative performance impact on the API, and splitting the API and simulator into two different components could potentially alleviate some of this negative performance impact, which in turn would increase the scalability of the system.

A drawback of splitting them into two components would be that, since the simulator would have to access and modify data through the API, more network requests would have to be done as an additional request to and from the simulator and the API would have to be sent, instead of just a request to and from the simulator and database. This could negatively impact the scalability of the system. Despite the drawback this probably would have been the preferred solution, but it was chosen not to do so as the task-description was interpreted otherwise.

When designing the prosumer and manager system the choice of not separating the two systems was made. The main advantages and goal of doing this was to increase the reusabillity of functionality and existing code when developing the system.

Another improvement to the system would be to split the GraphQL-API and the REST-API into separate components. This would make sense from a micro-services point of view as they have somewhat different functionalities and are queried in different ways, and it could potentially increase the performance of the REST-API as it could possibly be tuned to serve images only.

3 Scalability analysis

As the system is deployed using docker and docker-compose it should be fairly easy to manage scaling it using swarm services such as docker-swarm or kubernetes. These services would allow for deploying several instances of each component of the system.

As the server is stateless (i.e. does not store data about or maintains connections to users) it should be able to handle quite a lot of requests. Since the pages are server-rendered as well, a potential possibility to increase performance would be to enable server-side caching, which means the server could store part of the, or even the entire, rendered pages in memory. As the requests to the server would increase one might also consider using a load-balancer, such as Nginx in order to additionally increase performance when serving requests, but the system does not provide any such service at the moment.

It is likely that the bottleneck of the system as the amount of users increased would be the GraphQL-API as the system does not provide a solution to the well-known N+1 problem, which states that with GraphQL the number of database-requests between the API and the database will increase linearly with the amount of results, N, instead of fetching all results using one query. Some other issues with scaling could arise from the flaws in the design choices discussed in section 2.3.

Arguably, the system uses quite a low amount of network requests so it would not have a large scalability impact on the network. This is the result of using GraphQL, as GraphQL uses a single endpoint and is able to express every relation in the data schema in a single query/network request.

Also, since the system uses an SQL-database, which are known to get quite a lot slower with large amounts of data, it might also have been advantageous to use a NoSQL-database, such as mongodb, which are able to handle larger amounts of datasets faster but do not provide the same safety features as SQL-databases. This might especially have been true if the system had used historic data and a lot of users would have joined the system as it quickly would have resulted in large datasets.

4 Security

4.1 Architecture

The security of the system is implemented using JSON Web Tokens, which are small encrypted/signed chunks of information.

When the user registers on the page and all the checks (e.g. there does not exist a user with the same email address) have passed, the password is salted and then hashed (using the well-known library bcrypt) before being stored in the database. The API then returns the new "auth-token" (authentication/authorization-token) to the client which stores it in a cookie. The auth-token is created by signing, using the simulator's private key and the HMAC algorithm with SHA256, an object which contains the user's account id and a boolean flag which indicates whether or not the user is a manager.

Similarly when logging in to the system the user will pass their email and password to the API which will compare the password to the hashed version of the password for that email. If they match, the API will return the same type of auth-token as when registering, and if not it will just respond with a 401-status code (not authorized).

After this token has been stored in the browser it will be passed around to the different components of the system to provide authorization/authentication, and for every component of the system it will be the only method of authentication/authorization.

4.2 Benefits and drawbacks

Some benefits from this system is that the entire system (except for the client and the database) can be stateless, i.e. it does not hold any information about or maintain connections to the users currently using the system. This means a malicious user will not be able to steal any of information about the user from the server.

Another benefit is that the authentication system follows the principle of not storing more data than what's needed for authentication/authorization, which is why the only info stored in the cookie is the account id and the manager-flag. This means that even if the attacker manages to somehow decrypt the auth-token, they would only get non-personal information. Also, since the auth-token contains the manager-flag it can also be used for authorization and not just authentication.

Since the server and API both hold the same private key, used to decrypt the auth-tokens, it is of utmost importance to protect those keys, since if the key were to be leaked, malicious users could decrypt stolen tokens and forge new ones.

Another security risk that arises with this system is cookie stealing. If a malicious user was able to steal a cookie they could sent requests to the API which would accept the token as valid and the malicious user would be granted access to all the user's data. It is therefore important that cookies are stored securely and that cookie stealing techniques, such as cross-site scripting (XSS) attacks are prevented. Thus, tokens in the system are stored in cookies instead of local storage, which are more vulnerable to XSS attacks. Also cookies should be stored as httpOnly-cookies, meaning they cannot be accessed through javascript at all. At the moment they are not however, which inflicts a large security-concern, but would be in the future if time was available. Another desired security feature which has not been implemented due to a lack of time would be to require the cookies to be sent over HTTPS only, i.e. the client would not send the auth-token over unencrypted protocols, such as HTTP.

Since the system uses an SQL-database (namely PostgreSQL), it is theoretically susceptible to SQL-injection attacks. This could be solved by escaping the query-strings. A few manual tests have been performed to test for such attacks, all of which showed the system was not susceptible to them. However, the escaping of the query-strings is all handled by the third-party PostgreSQL-client and nothing has actively been implemented in the system to prevent it, so it is unclear as to how immune the system is, or will be in the future, to SQL-injection attacks.

At the moment the database is not encrypted which should definitely be the case in the future, however, as the password stored in the database are salted and hashed, it should not be possible in the case of a possible leak to deduce the password via either brute-force or rainbow table attacks.

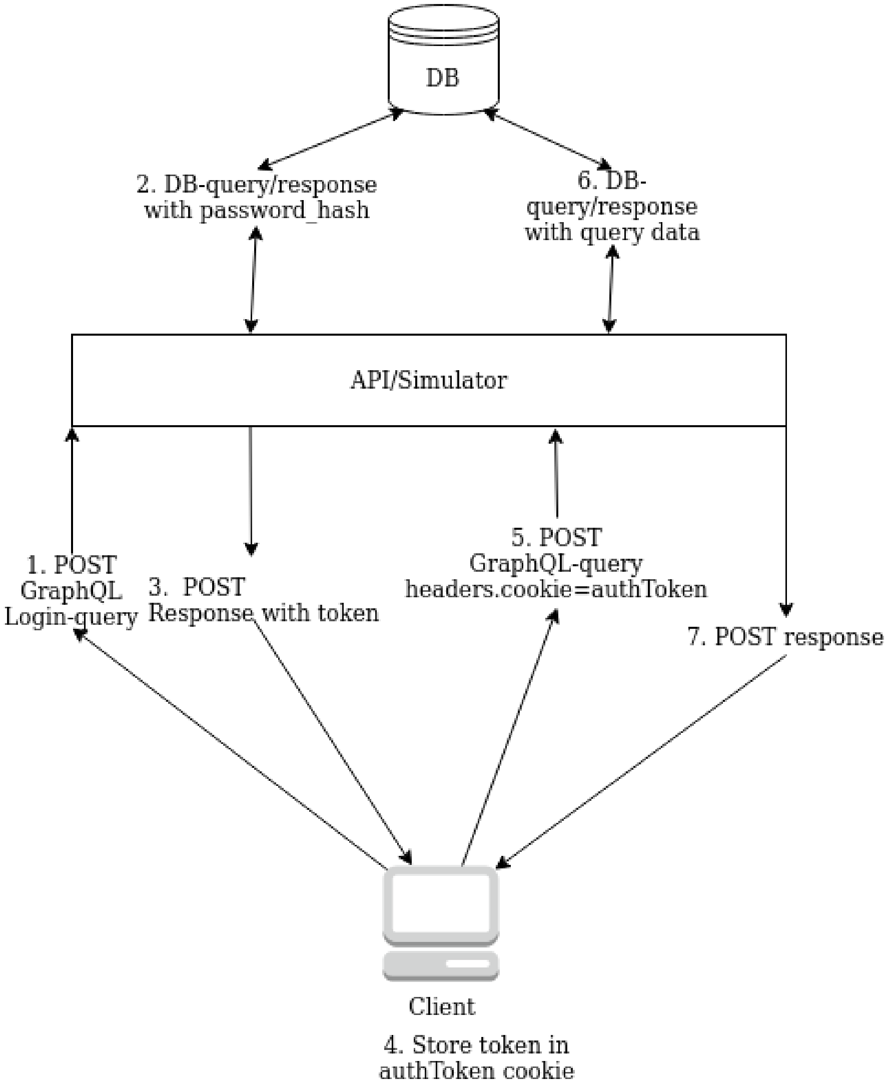
DB
2. DB-query/response with password_hash
6. DB-query/response with query data
API/Simulator
1. POST GraphQL Login-query
3. POST Response with token
5. POST GraphQL-query headers.cookie=authToken
7. POST response
Client
4. Store token in authToken cookie

Figure 3: An example highlighting the requests/responses when successfully logging in and querying the API

5 Implemented functionality

All of the basic functionality requirements of the project has been implemented in the group's system. Apart from this the advanced functionalities implemented are sliders to control the ratio from/to the market and battery for all users. This includes the prosumer's ratios in their dashboard as well as the manager's ratio in their dashboard.

The prosumers and managers can also update their credentials and delete their account from a profile page.

Another advanced feature that has been implemented is the possibility to be logged in and use the system with multiple devices on the same account. The system is also responsive and usable on different devices and screen sizes.

6 Challenges

One challenge was to manage dependencies on the client-side in a sane way. As no build-system or bundler, such as grunt or webpack, was used, dependencies had to be included via script tags which made managing dependencies difficult, and the code not easily debuggable as the source of a function was not obvious from the code.

Another challenge was to manage the flow of data through all of the components while developing. Whenever some new form of data was to be developed one would have to implement it on the client, API, database and occasionally the server. This meant that doing something as seemingly trivial as renaming a variable or function, could cause a breaking effect in some other component in a non obvious way. This could have been mitigated by writing more tests.

Yet another challenge was to find a good testing library for the GraphQL-API. No testing-library with the desired features could be found, and therefore large amounts of the GraphQL schema (which arguably is one of the more important parts of the system) remain untested. To mitigate this Insomnia was used to verify the APIs functionality but it requires a lot of manual work while developing.

During the development of web application it was also a challenge to understand how the time flow in the simulation would be synced with usage of the website in real time. This was desirable in order to achieve a natural flow of e.g. wind speed changes and not have an unnatural behaviour of simulation variables when used by a real user.

7 Future work

In the future the first things to fix would be to address the security issues addressed in section 4.2, most important of which probably are the httpOnly-cookies, https and SQL-injection issues, as these pose a security risk for the entire system.

It would also be desirable to separate the simulation system from the API. As the project grew in size and the simulation API started handling a lot of different functionality, not connected to the simulation e.g. account registration, it made less sense that the entire API was coupled with the simulation. Also if the electrical grid were to be implemented in reality the high coupling would make the connection and integration of a real data stream, instead of a simulation, require more effort. This would also have to be done in case of replacing the polling mechanism with a subscription-based mechanism for updating the data on the client, since the simulation modifies the data directly via the database, but all the other components modify via the API, which means subscriptions to the API would not cover all modification of the data.

A test suite for the prosumer and manager system and the API is also needed. Currently only a test suite for the simulation system have been established (apart from the GraphQL-API) and to further verify the functionalities of the system this would be required. Established test suites would also improve the ease of development and implementations of new features in the future.

The first new feature to implement would likely be historic data, which could be shown in a graph for an improved user experience. Historic data is also important specifically for the manager in order to draw conclusions of the power grid's performance in its current state.

To further improve the user experience and functionality of the system a notification system would be needed. Since the system is used to operate an entire power grid, important information such as how many prosumers are experiencing blackouts needs to be reportable to the manager both via the website but also over email and telephone.

Functionality for buying and selling electricity to and from the market is currently implemented but no balance system exists. It would be desirable to keep track of the users balance and connect it to a invoice system to issue monthly payments.

A Time management

The work was divided by making Github issues for the features to be implemented, and each group member then assigned themselves to whatever issue they wanted to implement. This let the group work at distances and individually. During the development of the web system the group divided the work into three major milestones: the simulation, prosumer- and manager system, each with separate deadlines. While working towards each milestone the group members worked on separate features of the system.

Analyzing the time reports a total of 421 man-hours has been spent on the project. With Martin Terneborg and Hugo Wangler spending 220 and 201 hours respectively. It can also be seen that not a lot of time was spent on bug fixing which kept the development going at a good pace. This is was mainly due to the test suite established for the simulation system and the use of Insomnia to confirm the functionality of the GraphQL-API.

B Contributions and grading

Below follows descriptions of each member'ss contributions to the project as well as grades.

B.1 Contributions

B.1.1 Martin Terneborg

- Docker images and services
- Profile page
- Authentication system
- Database schema
- Login and registration page
- Simulation
- Front-end design (bootstrap)
- GraphQL schema
- Manager dashboard
- Testing frameworks and tests
- Starting/stopping powerplant and powerplant status

B.1.2 Hugo Wangler

- Wind speed model
- Wind turbine model
- Battery model and usage operations
- Simulation logic for prosumer selling/buying/blackout
- Prosumer and manager ratios
- Prosumer dashboard
- Manager dashboard
- Tracking and determining of user online status
- Misc front-end design
- GraphQL schema
- Database schema and queries
- Tests

B.2 Self assessment and grading

B.2.1 Martin Terneborg

I feel as though i deserve a grade of 4, as all basic functionalities and some advanced have been implemented. Another reason was that i implemented the authentication system almost from the ground up (with exception for the JSON Web Token library for signing/unsigning the cookies).

B.2.2 Hugo Wangler

For this project I feel like I deserve a grade of 4. Although not a lot of advanced functionality has been implemented I feel like the resulting project, and my contributions has a good software quality.

C Project release, deployment and installation instructions

A running production environment can be found at http://ec2-34-198-184-223.compute-1.amazonaws.com. The latest Github release of the system can be found at https://github.com/MTBorg/M7011E_lab/releases see the README for installation and deployment instructions.