Green Lean Electrics Report

Design of Dynamic Web Systems, **M7011E**

Alexander Mennborg
alemen-6@student.ltu.se

Niklas Lundberg
inaule-6@student.ltu.se

February 10, 2020

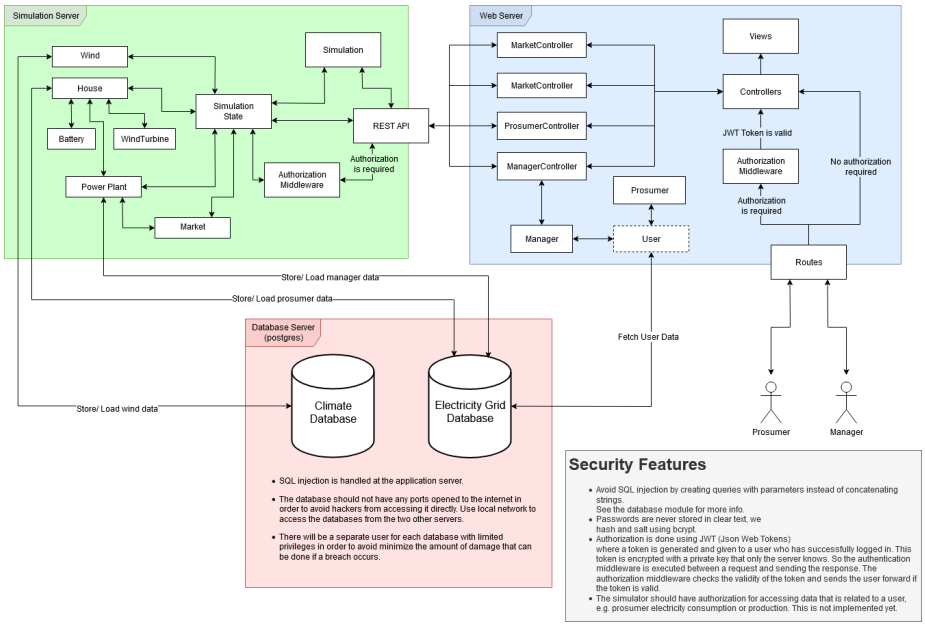
LULEÅ
UNIVERSITY
OF TECHNOLOGY

1 Introduction

The aim of the project is to make a dynamic web server where users called prosumers can monitor their power consumption and production. There should also be users called managers that can control and monitor their coal power plant. The production and consumption of power is simulated for each prosumer and power plant.

Contents

2 System Architecture



2.1 Database

In this project there is two databases hosted on the same PostgreSQL server. One of the databases holds all climate related data like wind speed, it is called Climate database. The other database is called electricity grid Database it holds all the data that is related to the electricity grid, like the power plant and prosumers data. The reason for having separate databases is to simulate that the climate data is from a third party. It also provides some security because the climate database user can't get to the more sensitive data in the electricity grid database.

2.2 Simulator

The simulator is its own separate server and its main functionality is to generate simulated data and provide that data to the web server through a restful API. The main reason for having it be a separate server is to make it easier to replace the simulator with a server with real data. The simulation mainly consist of a simulation class which job it is to handle the simulations state by periodically

calling the step event and checkpoint event. The step event is for updating the state of the simulation and the checkpoint event is for storing the state in the databases.

The state of the simulation is contained in a class called *SimulationState* which keeps track of all the simulated objects. All of the simulated objects have classes that also act as states but the difference is that they also have functions for simulating data. One of the simulated objects are houses which simulate the electricity consumption of a prosumer. Houses contains a battery object for keeping track of how much electricity they have stored. They also contain a wind turbine object that simulate the amount of electricity a house produces from the wind. Then there is the power plant class which simulated electricity production and keeps information on the state of the power plant. It also contains a market object which keeps track of demand and price. Houses can buy and sell to the market that they are connected to by getting the reference to the market through the power plant object.

The simulator runs on a Nodejs server where the only way of getting communication is through a restful API. The API gets or sets the simulations state by searching in the state of the simulation object, it can also get data stored in the databases. But the API uses a authentication middleware which means that only users with a valid authentication token that they have received form the web server can access the simulation data. The token works because the two servers have the same private key which makes it possible for both of them to decode the token. This is to make sure that the users first contact the web server which job it is to handle the user requests.

2.3 Web Server

The web server is an Nodejs server, its main functionality is to handle the presumes and mangers requests. The reason for having it separate from the simulator is to make is easier to swap the simulator server for a server with real data. The web server consist mainly of authentication, views and controllers. Where views are the different web pages and controllers are classes that handle requests from the REST API.

The authentication uses tokens that are encrypted on the web server using a private key that only the web server and simulator has. The token is created when a user signs in or up to the application and is only valid for a period of time. When the user then send a request they also send there token that the server authenticates using a authentication middleware. The token also contains the user id so the server knows who is accessing the server. The fact that the token is a encrypted with a private key and that it contains a user id, makes it almost impossible to forge a fake token.

There is two separate groups of web pages for the two different kinds of users. The first group of pages is only accessible by the prosumers, those pages are the prosumers dashboard and overview page. The dashboard contain some general information about the prosumer that is fetched from the server in a set time interval using a JavaScript procedure that is loaded into the users browser. The overview page shows the prosumers status and production data, it also shows wind speed and price. The overview also have a JavaScript procedure that fetches data but it also contains some input fields, used for updating some of the prosumers house settings.

The second group of web pages are only accessible by the mangers. Managers has a control panel page for controlling the power plant, it works similarly to the prosumers overview page. They also have a page for seeing a list of all the prosumers and their status. The manger can also block the prosumer from selling and delete their accounts form this page. The last page displays more information about just one selected prosumer and is accessed through a button on the page where all prosumers are listed. The data shown on the manager pages are also from JavaScript code that fetch data from the server in set intervals.

All requests to the web server are routed to different controllers after they have been verified. The controllers then preforms the requests and returns a response to the user. It is here the web server fetches data from the simulator if it is requested by the user. It is also here user account data is crated, stored, updated and fetch with the help of models for the prosumers and managers.

3 Design

3.1 Structure

The project was structured into three different servers described above. The PostgreSQL database is required to run on its own server. The simulator module could be put on the same server as the two other modules namely the prosumer and manager. Since the simulator has very different functionality from the other modules it was later decided to put the simulator on a separate server to improve code readability. The main drawback of this approach is that it was harder to implement the REST API calls from the prosumer and manager modules to the simulator since some requests require the authorization token which is located on the web servers session storage. The simplest solution was to duplicate the REST API on the web server and implement a simple fetch call to the simulator with authorization token provided. Since the prosumer and manager has similar functionality it was better to keep those sub systems on one server to allow code reuse. The use of Docker and docker-compose made

this effortless to setup, develop and deploy to EC2.

3.2 Performance

There has not been any performance tests done throughout the project but since there is only one server running for each part there are reasons to believe that high network traffic will degrade performance and might even crash the server. There is no support for caching so some repetitive requests that require the same memory allocations are not optimized. An example of where caching can improve performance is during the authorization middleware where the users data is fetched from the database to compare passwords and then removed. But since many pages requires user data it would be more efficient to cache the user data to save up to two database accesses at every request in the web server.

3.3 Scalability

The simulator and web server is not designed with server scalability as a focus and SQL databases also faces challenges with scalability e.g. sharding. Since we are using Docker we could use Kubernetes for setting up load balancing but since the design lacks support for this refactoring will be needed.

3.3.1 Scaling up the Simulation Server

In order to scale up the Simulation Server there needs to be a global state amongst the servers. When running multiple servers starting at the same checkpoint it will start out with the same simulation state. This however does not work since these state objects wont be synchronized and random variables e.g. wind generation will output different wind speeds for the same time on each server replica. This could be solved by only storing the state on a single server and everyone communicates with that server to access the state variables. This design might suffer from race conditions since different servers are simultaneously trying to mutate the state. Consider using a database instead for storing the state as this provides mutex locks by default. Another problem is that the simulator also needs to update only a subset of the users otherwise the same computation is done for each server, some kind of internal load balancer for separating tasks. The only scaling that will be easy is vertical scaling i.e. add more power CPU and RAM.

3.3.2 Scaling up the Web Server

This is fortunately much simpler since the web server only handles requests sends responses with predictable behavior. There is no states other than sessions that will remain after the response is sent. So this could easily be scaled both horizontally and vertically.

3.4 Security

From the start of the project it was designed with security in mind. For example when dealing with SQL queries it was important to avoid the ability to do SQL injection attacks from the start rather than fix it later since it could easily be forgotten. When designing the simulator REST API it was important that also that the parameter and queries were sanitized even though users never directly query the simulators REST API.

When designing the authentication system it was decided to use the library *bcript* for hashing and salting passwords. Hashing makes sure that passwords cannot be reverse engineered and salting makes it so that two exactly the same passwords does not generate the same hash. This makes it very difficult to hack into someones account even if the hashed password is revealed and can therefore only be guessed so users has to pick safe passwords. When logging in a authorization JWT (JSON Web Token) token is generated and kept in the servers session storage, this token holds the users UUID (Universally unique identifier) and is hashed with the *HS256* algorithm using a private key that only the web- and simulation server knows. The users never actually gets to see this token but even if this happens it should be difficult to reverse engineer the token if a strong private key is chosen.

4 Advanced features

The simulator uses a database for storing historical data which is used in different graphs to show how the data have changed over a period of time. The database is also used for storing the state of the simulation which makes it possible to restore the simulation to a previous state if something were to happen to the server. The user accounts are also stored on a different database which means they also don't disappear when the server shuts down.

The simulator is on a separate server which makes it possible to have it run on a different computer and it makes the application more modular. Having it on a separate computers relieves some of the computing load. It also allows for having multiple web server working with the simulator. The web server could

also be reused for another project were the data isn't simulated.

There is also docker files for the servers and a compose file. This allows the servers to be run on separate containers which adds a layer of protection to the host computer and makes it very easy to set up a new instance of the project.

5 Challenges

In the beginning of the project it was very challenging to get a clear picture of the project from the instructions. That made it difficult to start coding and to setup a workflow. It was also very difficult to set up a project structure, the reason being lack of experience with dynamic web applications.

Networking between the web server and simulator was challenging because the errors are very vague. The most common one being ECONNREFUSED, which can be caused by many different problems. It also didn't help that the servers where running in docker containers because the IP addresses from the error were of the docker containers. Those IP addresses are not trivial to access from the host computer which added another layer of difficulty. There was also the CORS (Cross-Origin Resource Sharing) protocol stopping communication between the servers which was a little challenging to understand at first.

Debugging was challenging in JavaScript because it is not a typed language which made it hard to have a good structure when coding. The bad structure caused spaghetti code which is very difficult to debug. That is why the simulator got refactored in Typescript. Another reason debugging was challenging is that the other persons code could sometimes be misinterpreted and that would cause the new code to not work as intended. This was not a big problem because the code was reviewed before being pushed to the master branch.

Working together was a little challenging because the skill level and experience with dynamic web development varied. It made it challenging because it was sometimes difficulty to understand the other persons code and work with it, especially when a new languages was introduced into the code. Communication was also a little challenging because there were sometimes misunderstandings of how a feature should be implemented, but it got better as the project progressed.

6 Future work

The user interface is the most important part of this applications, because if it is not easy to use no one will use it. That makes the most important improvement to improve the usability of the web pages by adding better visual representation

of the data. Like having more graphs and customizable views of the data. It is also important that the improved pages have a more consistent design, to make it easier to navigate and use.

Another important part of web applications is the traffic over the internet. In this case it is very important because some of the data needs to be fetch at a frequency of 0.1 Hz which generate a lot of traffic. That is why web sockets would be the second thing to implement, this will reduce the amount of data sent by removing unnecessary calls and overhead. Web sockets should be used in all the instances where the data is fetch multiple times on a single web page, like fetching the wind speed in real-time.

Security is also very important when dealing with personal data, that is why the third improvement would be to have the simulator have its own private key. This will add another layer of protection to the simulator and the sensitive data stored there. Also there should be another database user added which only can access the user table in the electricity grid database. That is because the web server at the moment has access to all the data in electricity grid. And by restricting the privileges of the web pages database user would make the simulator the only access point for that sensitive data.

The last improvement is to add automated tests for the API:s and authentications. This will make it easier to find bugs and other faults in the applications. It will also make it easier to develop for, because the tests can be used to test so the new code doesn't break any thing. These two benefits will increase the quality of the application because there will be fewer mistakes.

References

Khalil Stemmler. How to setup a typescript + node.js project. `https://khalilstemmler.com/blogs/typescript/node-starter-project/`.

w3schools. Javascript tutorial. `https://www.w3schools.com/js/`.

w3schools. Html tutorial. `https://www.w3schools.com/html/`.

w3schools. Css tutorial. `https://www.w3schools.com/css/`.

Chartjs team. Chart.js. `https://www.chartjs.org/docs/latest/`.

Mozilla. Cross-origin resource sharing (cors). `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`.

Postgresql. Postgresql 12.1 documentation. `https://www.postgresql.org/docs/12/index.html`.

7 Appendix

7.1 Niklas Lundberg

In the beginning we had a hard time understanding the project which made it very difficult to start programming. Another reason is that I have almost no experience with JavaScript which made it more difficult to get motivated. The lack of motivation caused me not to work the 20 hour per week. But that changed when my understanding of JavaScript and dynamic web apps quickly grew. That gave me the motivation to work extra hard reach the 200 hour goal for this project.

I spent a lot of time on the simulator in the beginning and in the middle of the project because I have experience with simulators. But I did not only work on the simulation parts of the simulator, I also worked on the restful API and docker for the servers. When the simulator was thought to be complete I stated on making web pages with JavaScript's that fetch data from the API:s. A big part of the time I spent on the web pages and API:s was learning how to do it. Then the last week I did a little of every thing.

We worked together by having a Trello board where we planned together all the tasks that needed to be made for each assignment of the project. Then we chose a task with the highest priority first and did it. This caused the work load to be very equal because we roughly worked the same amount of hours each week except over the holidays, where I didn't work much.

Both of us did a little of every thing on this project and the work load was pretty even between us. But Alexander did more work then me and did more of the advanced work like the authentication system. This was because I have less experience in dynamic web development which made me a little slower at finishing my tasks.

I think we both deserve a four because we worked very hard on this project and both of us worked on every part of this project but in varying degree. The project also has some cool advanced features like a database that is used in many different ways. But the most impressive advanced feature we implemented was to have the simulator in a separate server. I also think Alexander deserves more credit then me because he worked more on the project and taught me a lot by giving good feedback on my code.

7.2 Alexander Mennborg

I had some previous experience working on Node.js with a simple online game using socket.io but that was over 4 years ago. Due to some overlap with another

course project I could not spend that much time the first weeks but I quickly realized that I had to pause that project in order to make time for this project and the other course this study period. I personally do not like JavaScript but I stuck with it and once I remembered that TypeScript exists It was already too late to redo the entire web-server.

I have a problem with spending too much time design graphical user interfaces. I have also spend a lot of time refactoring code but always failed to make the JavaScript code to look good, going back and forth between different designs never finding myself satisfied. I rewrote the entire simulator server in just under 10 days using TypeScript which improved the quality of the code. I don't like putting SQL code inside the source code so I build a QueryBuilder that generates SQL code and provided simple CRUD operations. This is where most of my time went but I have of course worked on other features in web-server.

I think we both deserve the grade 4 since we have delivered a project with good quality but less quantity or features. We could have skipped rewriting the simulator but the fact that we decided to rewrite the entire simulator proves that we care about code quality. Personally I also care about the quality of the user experience and prefer something simplistic and visually pleasing over something bloated and messy even if it has less features. If I had more time I would rewrite the entire web-server as well.

7.3 Link to github

https://github.com/Aleman778/m7011e-project