M7011E

Luleå University of Technology

Rustislife
Aron Strandberg - arostr-5@student.ltu.se
Mark Håkansson - marhak-6@student.ltu.se

January 21, 2020

Figure 1: https://skitterphoto.com/photos/skitterphoto-3856-default.jpg

Contents

Authors
Aron Strandberg - arostr-5
Mark Håkansson - marhak-6

1. Introduction

This dynamic web application is a project designed and built for the purpose of simulating, modelling and predicting a small scale electricity market. The web application is built on top of *NodeJS* a *Javascript* execution engine which executes Javascript on a web server. The server runs a simulator which generates data such as market price, wind speed and all households electricity consumption and wind turbines' production. The server also handles the connection with the database and the templating of dynamic web pages before they are sent to the front-end or client. To present the data that's generated an API, *GraphQL* has been introduced to abstract away complexity around data querying and manipulation. The data generated by the simulator is stored directly to a document driven database, *MongoDB* with the help of *mongoose*, an object document mapper for translating the documents between the code and database. The web application has implemented some security aspects, especially some of OWASP top ten security concerns as SQL injections, sensitive data, authentication and authorization. The way the system architecture has been designed have allowed for system scalability and some modularity where the API (GraphQL), web server (Nodejs), database (mongoDB) and simulator can respectively run on different machines. This is demonstrated in fig.2

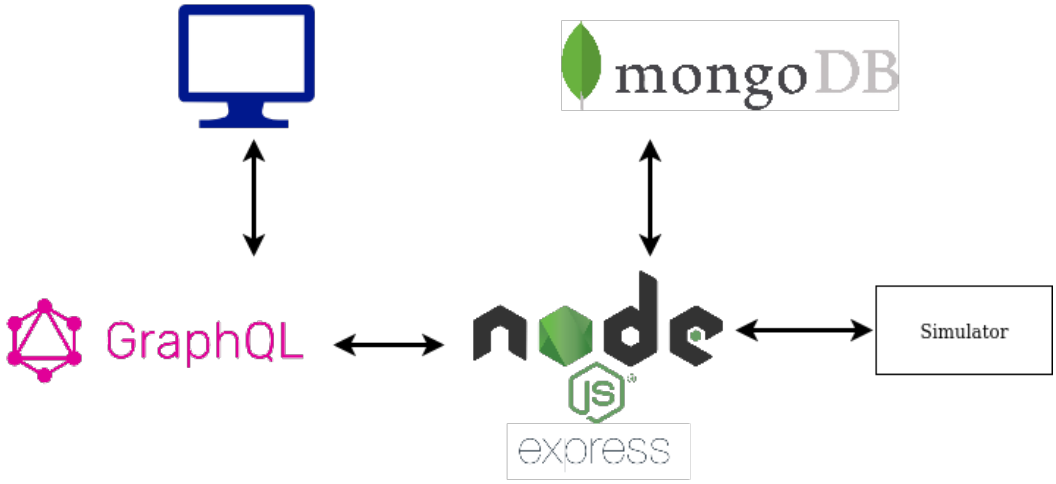

Figure 2: Overall system architecture of the web application

2. Design choices

2.1 Overall architecture

2.2 API

The API was built with GraphQL. It was chosen because it's an alternative to REST that was thought to be interesting to look further into. As well as an educational purpose as the knowledge for REST already existed in within the team. There were a few things that were found to be great when compared to a regular REST API. When querying on the client-side it allows for queries to return exactly the data that's requested. The type system is very useful to specify what type queries and mutations return not to mention what types all arguments and parameters

Authors
Aron Strandberg - arostr-5
Mark Håkansson - marhak-6

should be. The included interactive tool GraphiQL allowed the team to easily test and debug the queries and mutations that was created. When an error occurred the response from the GraphQL query/mutation would return some information of what went wrong. It was also very easy to create and throwing custom errors from the API back with th query whenever something went wrong.

Another benefit of GraphQL is that it's less complex than REST API described below, see fig.3. All queries as e.g. fetching simulation data, login, sign up and image uploads are handled by the GraphQL API. On the other hand a few functionalities were handled by REST APIs as e.g. fetching online users because of it's simplicity. The online users functionality is based on active express-sessions on the server and therefore it was easier to use REST in this case. Therefore both GraphQL and REST API has been used in the application but mainly GraphQL. Mixing the both APIs seems to have been a common case when searching about it on the web but may not be ideal but it's an option if the developers are not willing to rewrite endpoints in the application and save themselves some hassle.

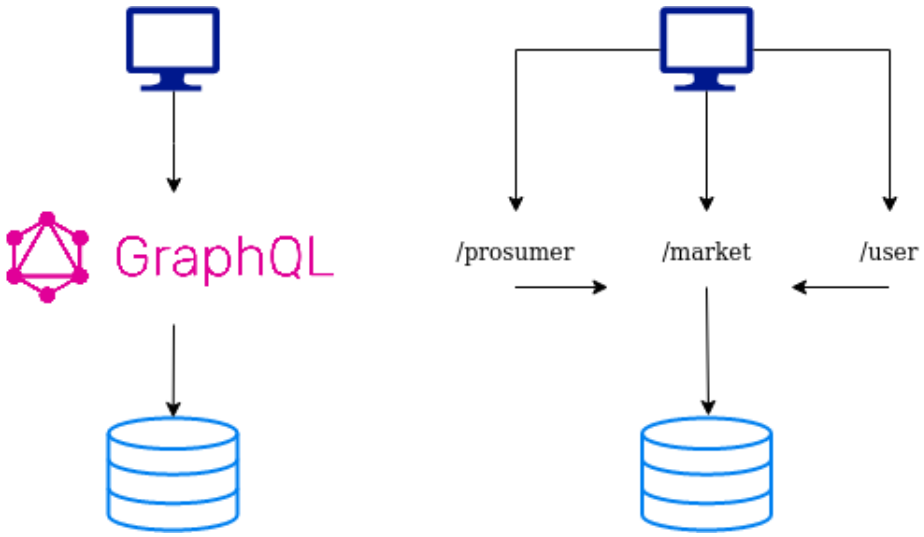

Figure 3: Demonstrating the difference between a GraphQL API and REST API during a client request to the server

Unfortunately there were some drawbacks with using GraphQL as well. As GraphQL is a relatively new technology it was difficult to search for solutions on specific problems. The team also only used the bare minimum tools available for GraphQL to work with NodeJS which required much longer time to get basic features implemented, then if the team had chosen to use GraphQL modules such as Apollo GraphQL that comes with many quality of life tools and improvements.

2.3 Simulator

The simulator is written purely in JavaScript and with an object orientated view in mind. The idea was to make the simulator as scalable and modifiable as possible by dividing the simulator into models of prosumer, consumer, weather and market where each model would handle itself (simulation rules). The controller would then connect all of the models and where new models can

Authors
Aron Strandberg - arostr-5
Mark Håkansson - marhak-6

be added to and removed from the simulation with ease. This would also make way for the choice of database used in the application as the models would be identical to the models stored on the database, MongoDB. The data which the simulator generates and stores to the database is time critical according to the project requirements. Therefore each model is stored in steps and all data is updated synchronously to the database. Time is defined only by the data (models). There is no connection between the web page and the simulator. This correlates to that the data displayed on the web page will always be truthful, up to date and there is no possibility of a race-condition. In the case of a simulator failure which would not affect the overall system, the simulator would try to restart and delete the latest entries in the database.

2.4 MongoDB

The reason why a document object-driven database were chosen were one, both project members already had experience of SQL and no experience of NoSQL and two, mainly because the data generated by the simulator were thought to be almost non-relational. Therefore a non-relation database were chosen. There is some coupling between the models e.g. prosumer and market but this can still be handled in a NoSQL database. Benefits of using MongoDB has been the simplicity when storing simulated data, the database schema or models could be identical to the simulators models. Thus, making the system more scalable and modifiable. Lastly, NoSQL is generally performing queries faster than SQL.

A drawback is that SQL provides a much more powerful querying language, that MongoDB lacks. Where an advanced and easy to understand SQL query might only require a single line of code, the comparable MongoDB query could require several lines and be more difficult to understand and unintuitive.

2.5 NGINX

For running the web application on the web the web server NGINX were used as a reverse proxy. NGINX also packs other functionality as load balancing and other utilities for web development which would or could become handy. NGINX were used instead of e.g. apache since knowledge of running NGINX already existed and for time related concerns.

2.6 Utility libraries

The NodeJS engine and package managers as npm allows for adding and handling libraries to the project with ease. Libraries added to the project were Express which is a minimal web framework for NodeJS and is a necessity when developing on the NodeJS execution engine because it abstracts away unnecessary complexity in this case. Since the simulator was written towards object orientation the database documents had to be stored as models in the database, therefore mongoose were used to build the models, queries etc. For handling user authentication and authorization *express-session* has been used to handle and store user sessions on the server. For storing user credentials as the password bcrypt has been used to hash the users password in the database.

Bootstrap has been used for front-end development to simplify the development process in HTML, CSS etc. and to make the application more responsive on both web and mobile devices.

jQuery is a JavaScript library that has been used for wrapping DOM (Document Object Model) related manipulations. AJAX (Asynchronous JavaScript + XML) allowed for asynchronous requests to the API which would make the application faster and improve overall performance.

2.7 Code style

For code linting the npm module *eslint* was used. This stopped certain code from being compiled if it did not follow the linting configuration and kept easily missable errors to a minimum and also gave good warnings when certain bad code was written. It also helped to automatically format the code, which was useful so that both team members could keep the code stylistically the same.

As in any case of software development each team member has his own coding personality and preferred code style. By using Github and the tools it provides as branches, issue and piping tests removes some development errors and meanwhile making development more fluent.

3. Scalability analysis

Steps taking to the application scalable has already been mentioned in sections 2.3 and 2.4. Listed below are further steps to make the application more scalable.

The way that documents is stored in the database could be improved. As of right now whenever a change is made to a document, that document is cloned with a new timestamp and then stored to the database. This makes it possible to retrieve the history of a certain prosumer or user etc. However it will consume more space as duplicate data e.g. the user's name will be stored many times over and older revisions will be stored on disk as they are not used frequently. Therefore it's believed that if many users were to use this application in its current configuration, database queries would bottleneck the simulation and a lot of extra space will be needed. A solution to improve performance would be to store all old documents in a separate collection and the latest entries of all users in its own collection. Thus a query on the current user's state will be performed in constant time as there is only one document of it in the collection.

A NoSQL database allows for horizontal scalability i.e. adding more machines for storing data to the system which is suitable for expanding the application when e.g. user usage grows. Another benefit is that data and users can be stored on machines specified for regions and meanwhile keeping the database intact. Horizontal scaling also allows load balancing where users can be directed to different machines to avoid overloading of the system.

Currently the web server (NodeJS + NGINX), database (MongoDB) and simulator runs on the same machine. This is not a good idea for different reaasons, bad performance in one part will affect the others and if one part goes down, it will take the whole system down. But the application has been built such that all parts can all easyily be made to run on seperate machines on e.g. AWS. If e.g. the simulator would crash this would not affect the other machines which prohibits total system failure and allows for implementation of system back-up functionality. Each machines could have a back-ups and if any machine would fail another would take over.

Authors
Aron Strandberg - arostr-5
Mark Håkansson - marhak-6

4. Security analysis

The server is connected to the MongoDB database with the object-data mapper *mongoose*. This uses a pre-defined schema for the document types that needs to be stored and all properties in those documents needs to be given a type. Mongoose will validate the documents to make sure they follow the schemas set previously, before saving them to the database. The API, GraphQL is also strongly-typed and verification is done on both client and server side. Authentication and authorization are implemented for all API queries where unauthenticated or unauthorized users should not be able to query or mutate data or access unauthorized URLs. Therefore it should not be possible to store wrongly typed data into the database as either mongoose, MongoDB or GraphQL will stop it and throw an error should any validation break.

One flaw of the current implementation is that user input is not sanitized before saving it to the database. A user could send a string with embedded code in a query which would be stored on the database. It seems unlikely (would be difficult to do) that such code would be able to run on the server-side, as the only code on server side that handles the data is the simulator and it should not interpet the data in any way such that a string could be executed. But if the code could be stored in the database, users could be served that code on the client-side if they were able to fetch that specific data.

An example that's applicable to this web application would be when a malicious user creates a new account, then they insert code into the username field that steals the current cookie. The backend will interpret the username as a string and should allow the user to be created (the following code is very loosely based on the current web application).

```
let username = '<script>malicious code here to steal cookies...</script>';
let password = '..';
Database.createUser(username, password);
```

On the manager's dashboard page there is a display of all the active users and their usernames.

```
<span class="activeUsers"></span>
```

When the username of the malicious user is loaded it will be interpreted as a script instead of a simple text field and the malicious user has stolen the manager's cookie.

```
<span class="activeUsers">
   <script>malicious code here to steal cookies..</script>
</span>
```

XSS is therefore deemed as a possible attack vector on the client-side. The best way to deal with this would be to sanitize the input in the mongoose schemas and/or the GraphQL API. Mongoose can use custom validation rules (and supports white- and blacklisting) and these will be run with the other validation checks when a document is saved to the database. There are many packages available on the node package manager that can be used to escape and HTML-encode input strings that could be used there. Which is the most important thing to sanitize in this web application's case.

Another problem is that some errors might be unhandled in the code. A user might send a query with bad input, intentionally or not. Since it's not properly sanitized, it might result in errors being thrown when storing. These errors can be thrown when storing the document or it can be thrown from the simulator when using that document. It is very likely that not all cases where errors can be thrown are caught appropriately. Unfortunately the team did not find any good tools to find cases where errors are not handled.

The web system is also susceptible to session hijacking. The connection between clients and servers is not encrypted as HTTPS is not used because the team did not have access to any certificate. Therefore requests are insecure and any sensitive data sent between the server and client can be read by any eavesdropper. Also since there is no fingerprinting added to the cookies, a cookie containing the user's session id can be stolen from a user and used by someone else to commit a spoofing attack. Thus gaining access to their account. To mitigate the problems, HTTPS should be enabled and cookies should be fingerprinted with the user's consent. If the spoofer's location is in the USA whilst the cookie's location was in Sweden the server can decide to close the session. Another good idea is to regenerate a new session id frequently on the server side, such that if the cookie is stolen it will contain the old session id of the user and they will not be able to access the user's account.

Some security vulnerabilities as brute-forcing e.g. doing multiple request of queries on the login page to GraphQL would instead be handled by the proxy server, NGINX where limiting the number of http requests from an IP-address.

5. Advanced features

Logging has been used in parts of the application using the npm module *winston*. It has mostly been used to log errors and warnings to file in the simulator. But can easily be used anywhere in the application as there are various logging levels, and can be used to log non-critical messages such as connection requests etc.

5.1 Simulator

In the simulator, wind turbines are able to break for the prosumer households. After every simulator tick there is a fixed probability of a turbine to break, when that happens a repair will be scheduled for a set amount of ticks in the future.

5.2 Prosumer

The prosumer page is completely dynamic and does not redirect users when navigating and does not refresh the page at all. There is a profile page where the user can update password and delete the account. Sliders for controlling ratio from/to the market. Because Bootstrap was used to design the page, the page can be scaled to different screen sizes and is responsive.

5.3 Manager

As the page design is based on the prosumer page, the manager page is responsive and supports different screen sizes.

6. Challenges

One of the big challenges with working with the API was dealing with GraphQL. There are many tools out there to build a GraphQL API and the one that was chosen was *express-graphql* for this system. Unfortunately there are many different ways to structure the API. One can choose to follow the GraphQL specification which is to declare types, queries and mutations in text form using GraphQL's schema language. The other way is to do build schemas by creating Javascript objects, then linking the types, queries and mutations together. When it comes to the latter, there seems to be no correct way to structure it. The way the current API is structured was deemed satisfactory but never felt like it was truly perfect. As previously mentioned the application could not be built on purely the GraphQL API because REST API is very embedded into the NodeJS and Express environment and were in some cases more straightforward to use.

Another challenge was writing tests for a web system. Certain things were easy to test, such as trying to test initial values in the simulator. Whereas the API was difficult to test due to most of the queries requiring an authenticated and authorized user. To do that a MongoDB instance needs to be run, and initiated with a test user and then the GraphQL queries can be tested. Because of this the team did not manage to write any tests for the API.

Debugging has been challenging because of JavaScript. E.g. the application can be running perfectly but functionality might not be working on the client side and system continues running meanwhile there might be bugs on functionality on the server side and when does functions are executed on users action the entire system crashes. Examples of these challenges can be presented on client side when an AJAX requests gets no data for a field on the dashboard and the application would still be running. On the other hand on server side if some component of the simulator would crash the entire application would crash. Therefore error handling for some components of the application were more critical than others.

Other challenges has been implementation of basic functionality as e.g. that the manager should be able to view the prosumers system or data. Here the decision was made to implement a variable user to the session and thus providing the functionality to change the sessions user (only if you are a manager) and by doing so the manager technically becomes the prosumer and are able to view the prosumer data aswell as manipulating it (controlling ratio). Some functionality as deleting the prosumer does not work since the manager don't know the prosumers password. The decision was made on minimizing complexity of the application and the need to build new specific component for the functionality.

Above could be an example of a challenge during development were as in any software development project some criteria can be misinterpreted or not fulfilling the project owners requirements. Therefore it's been listed a challenge since the communication between developer and product owner is of most importance.

Separating development- and production environment. NodeJS has multiple tools for separating the both environments and this could have been done better.

Also writing good and clean code is difficult in a larger project. The team used the Javascript linter *eslint* together with Github's own configuration to keep the code written between team members visually the same. But a linter does not comment on the code and the simplicity of the written code. It was easier in the beginning to write coherent, modifiable and easy to understand code when the application was small. But as the development progressed and things started to become

more complex and the project deadline came closer, the team were no longer able to spend a lot of time on small things. Thus because of time constraints in the end of the project many changes and fixes to the code were quick fixes and there was not a big focus on writing the best code possible. Which is a little bit irritating as both team members wanted to write as good and clean code as possible.

7. Future work

The current user models are very barebones and only contains a username, password and an optional profile picture. The prosumer is related to a user via the username. In the future it might be nice to add more information to the user such as a real name, email address which could be used for e.g. password resetting or giving a more personalized dashboard for the user. For security reasons it might also be good to store the last time the user logged in and the IP address that the user logged in from. Then the user could see all sessions that they've logged in from and maybe even log out certain devices.

As written in 2.4 it might be beneficial in a performance standpoint to improve the way documents are stored. Perhaps by either storing historical data inside an array in each document to reduce disk storage or separate old documents to a separate collection such that the latest document can always be queried in constant time.

The GraphQL API implementation could need some cleanup. There's a lot of duplicated code and there could be more abstracted and modifiable code/functions. As in any project, development can leave a lot unnecessary code all over. Therefore, all functionality could have a second look-over in the future.

Some advance functionality as streaming the data to the dashboard using sockets instead of AJAX, better implementation of online users and chat functionality. As mentioned, online users are shown depending on active sessions which is not fully accurate since e.g. if the user closes the web browser the session would still exist and the users would be shown as online. Here sockets would be a better alternative for streaming data, showing online users and chat functionality.

Moving the application into a *"wrapper"* as Docker to make the application both more scalable and reliable. Docker is a platform for running the application on almost any operative system. It could have been introduced in the beginning of the project which would allow developers to develop in their preferred environment while not risking incompatibility. Then at production allowing the web application to run on different operative systems. Thus, making the application more flexible and more secure.

Apache JMeter which is a load testing software in javascript which allow the developer to test the application for heavy usage (multiple users) to make the application more robust and benchmark it's performance.

Sessions should be stored on the database. The current session store is in memory and will leak memory after running for a while with many active sessions. Thus it is recommended to swap to a MongoDB session store instead. The best alternative that was found was the npm package connect-mongo which seems pretty much straightforward to use. The team did not choose to implement this as it would take some extra time to debug and would delay the release.

User data should be anonymised when a user wants to delete their account. Currently when a

user deletes their account their user and houehold information are all deleted from the database. It might be good to anonymise the data instead such that no private information can be linked to the data. Because it might be good for the manager to still have the historic data in the future.

As mentioned in the security vulnerabilites chapter, data sanitation should be done on the backend for user inputs. HTTPS should also be enabled.

References

SQL vs NoSQL,
https://www.geeksforgeeks.org/difference-between-sql-and-nosql/

OWASP,
https://www.owasp.org/index.php/Main_Page/

NodeJS,
https://nodejs.org/en/

npmJS,
https://www.npmjs.com/

JavaScript,
https://developer.mozilla.org/en-US/docs/Web/JavaScript

GraphQL,
https://graphql.org/

MongoDB,
https://www.mongodb.com/

MongooseJS,
https://mongoosejs.com/

Bcrypt,
https://github.com/kelektiv/node.bcrypt.js

Bootstrap,
https://getbootstrap.com/

JQuery,
https://jquery.com/

AJAX,
https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

NGINX,
https://www.nginx.com/

Amazon Web Services,
https://aws.amazon.com/

Authors
Aron Strandberg - arostr-5
Mark Håkansson - marhak-6

A. Time reports and analysis

A.1 Aron

By registering the time and summarizing what has been done on the project has provided a foundation for self assessment. Approximately 170 hours have been registered which is less than the requirement of 200 hours but more hours has been spent on the project which has not been registered as communication between project member, research around web security (OWASP) and more. Most time spent has been on the simulator, security, API and front-end development. This is somewhat similar to the task performed by the other project member and therefor one can say both members have been working on all components of the application. In the beginning of the project both members worked together on the foundation of the application to get a basic understanding, to discuss and to set pathways concerning database, API and overall project focus.

At the end of the project both members were working independently and focusing on different functionalities of the application. Thus, the importance of good communication between both members has been essential otherwise working independently can result in that team members would be working on the same functionality. This did happened and therefor in future, setting up e.g. issues in the Github environment would minimize this from happening.

Both members did a good job and had developed a dynamic web application that fulfilled basic functionality as well as some advance functionality mentioned in the requirements. The project has been fun, engaging and educational which has supplied fundamental tools for future software development.

A.2 Mark

The time spent according to the time report on this project for me was 175 hours. This was excluding the time spent on the report and some days I forgot to log my time spent. The time was spent on various things, mostly the API and simulator as well as the prosumer page. We did split the responsibilites of the different parts of the projects within the team, but rather worked on the parts that each member wanted to work on. I think that after all, we both spent time on all parts of the project but at different points in time. I.e. we never worked on the same thing simultaneously. It worked very good as we could do whatever we felt like doing. But in hindsight it might have been good to divide the tasks into issues on Github and then each team member could've picked an issue they wanted to work on. It would have made it easier to understand what the other team member was doing at all times.

Even though we did not do many of the advanced functionalities I personally think that I've been challenging myself and worked as hard as possible during the whole project. I did not have much previous experience with web development in Javascript or NodeJS, or creation of APIs, so much of it was all new to me. Some of the time were spent working on learning how to deal with bigger projects with tools that are used in the industry such as continuous integration, good git behaviours, linting rules, loggers, finding security flaws etc., that was not part of the basic nor the advanced functionalities. I think a grade 4 would be fair for the work that I contributed with.

Authors
Aron Strandberg - arostr-5
Mark Håkansson - marhak-6

A. Project link

`https://github.com/markhakansson/m7011e-dynamic-web-systems`