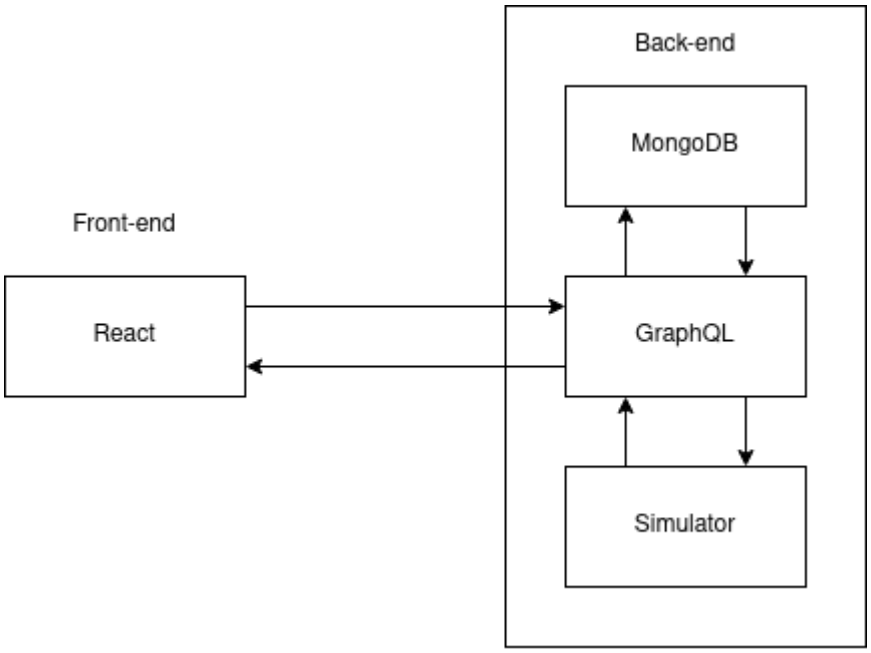
Design of Dynamic Web Systems

Thomas Schmidt, tohsic-7@student.ltu.se
Hector Nyblom, hecnyb-7@student.ltu.se

Introduction to the system:

Our system is built on an express server, running a graphQL API and using REACT as our front end. The communication through the system can be seen in the image below. Our GraphQL API is the one communicating with our database. So in order to fetch and update values in the database we have to access our resolvers in the api.



.

Database:

Our mongoDB database contains 3 different collections. Prosumers, Consumers, Managers. All of these contain relevant information for our different users.

GraphQL:

For every collection in our database we have divided our resolver functions into files that correspond to the specified user type. In every file we have functionality that allows us to insert, get one, get all, update and delete values from the database.

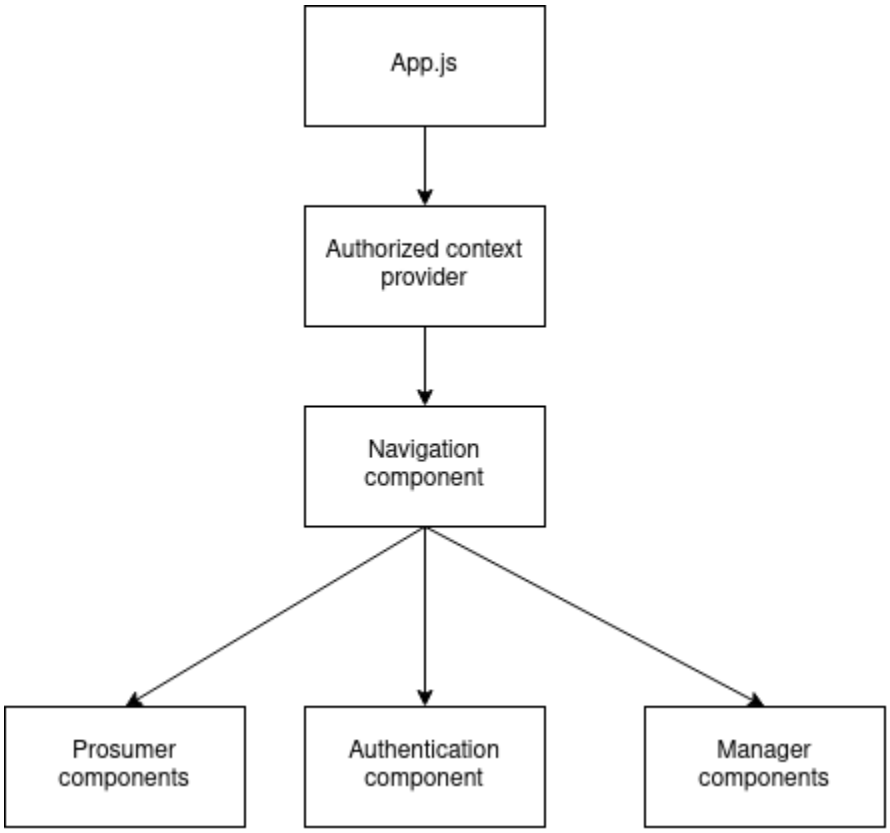
Simulator:

The simulator is a loop that every third second updates the values as if 1 hour had passed in real time. Each loop we calculate new unique values for wind for each prosumer using gaussian

distribution with the help of a box-muller transform. The wind is then used to calculate the production of the prosumers. Consumption for the prosumers is also calculated by using gaussian distribution. When we know these values the buffer of the prosumer is updated. Then these values are sent to our api resolver that updates them in our database. The consumers are then fetched to update all of their consumptions. The manager is updated with the demand from all the users and depending on the status it may also create some production. Lastly when we know the entire production and consumption of the system blackouts are calculated.

Front-end

This image shows how the flow of components in react are connected. It maintains this flow so that a context component always is able to pass context state variables down to it's child components so that they can be used there. The navigation component is always present for every page in our system, but changes depending on the context, so it is always rendered. Then depending on what the url and authorization is the other components may be loaded. The authentication component is the login and signup page where users can create a user or log in to their user if they are authorized. The prosumer components can only be loaded if the user contains an authorized token in their local storage, with a prosumer user type. The same goes for the manager page but they need a manager user type. Otherwise the user will be redirected to their user page. Or if there is an unauthorized token, it will be removed from local storage and will be redirected to the authentication page.



Design choices:

GraphQL - For our api we chose to go with graphQL and run it on an express server. From what we could tell when comparing REST and graphQL it seemed like graphQL is more modern and efficient. GraphQL is powerful as it is possible to obtain specific data with only one http request. But it took us a long time to implement the api because it is very type sensitive and the syntax was not really straightforward. One struggle we had with graphQL was that uploading image files to the server was not inherently supported.

MongoDB - For our database we went with mongoDB. By the information we found online regarding graphQL, a lot was based on a mongoDB database, so it would be easier for us to find examples combining these two. It is also very easy to understand syntax and very efficient. We have also not worked with mongoDB before so it was interesting to learn about a document based database. One issue with mongoDB is that it does not support transactions. So if we want to use sequential queries that are dependent on all succeeding data could be corrupted.

React - For our front-end we used the react javascript library. The library allows for easy rendering of changing state variables. And because it is component based it is easy to pass context variables down to components that need to use them.

JWT - To persist context data when refreshing a page we needed to store relevant data in the browser. We haven't done this before so we did not really know the best way. But it seemed like using web tokens and storing them in local storage on the client was efficient and one of the easier ways to do it.

Mutler - We choose to use mutler to upload files to our server because of the problem we had with graphql. From what we could tell this was one of the easiest methods to upload files but generally we would have preferred to use our graphql api to take care of the file uploads.

Graphql
Mutler
React
nånting

Scalability:

A good choice for our system was using graphQL. This is because to obtain specific data it only requires one http request instead of multiple http requests such as in REST. This means that our server generally gets less traffic and can therefore handle more users wanting to communicate with the server. But a problem in our system is that to fetch all the updated values from the simulator and present them on our frontend we make every user fetch their data every second. If a lot of users are logged in and observing their information this could lead to too much traffic.

MongoDB is an excellent choice for horizontal scalability. Which means that computation power can be sharded and handled with multiple machines. We have not enabled sharding for our

system at the moment so that would be a bottleneck, but if traffic were to increase this could be a solution to distribute the workload.

Grapghql good because less https requests.
Dont know how much traffic it can handle
Try to minimize http requests
Subscriptions istället för att uppdatera varenda sekund

Security analysis:

We have taken certain precautions in order to make our website as secure as we can with the limited time. To begin with all of the passwords that we put into our network are salted, hashed and verified using bcrypt. This is so that even if someone would be able to access information about a user through the backend their password would not be leaked.

In our system we are using tokens in order to verify that a user can access certain restricted pages and functionality. This is done so that if a user authenticates with the correct username and password they will be sent a token from our api. This token is then stored at the clients local storage. The token contains what type of user it is, user id and the tokens expiry date. By using PrivateRouting for prosumer and manager routes in react we can ensure that only an authorized user with a verified token and correct user type can access these pages. The tokens are only used so that users cannot access restricted pages, but we have not implemented it in requests to our api. This could result in someone not authorized changing values in the database through our api which we realize is a severe security issue but that we did not have time to implement. An issue with storing the token in local storage is that it is vulnerable to XSS attacks.

Our front-end web server and our express server uses https connections. What this means is that our website encrypts data that is sent to and from the server. This prevents other people from eavesdropping and seeing what is being sent between the server and the user. This is helpful to secure that no one can access sensitive data. Such as passwords or a user's unique token. This also helps users going to the url that the website is verified from a certificate authorizer. This is not the case in our website though as we self assigned our certificate, but if it were to be deployed for real we would go through one of these authorities.

tokens
Https
Hashing lösenord

Advanced features:

The only “advanced” features that we implemented in our system is that values like battery ratios can be manipulated by the use of sliders.

Challenges:

One of the major challenges we met was how to persist context for a user when refreshing the page they were currently on. With react context it worked perfectly until someone refreshed the page and then all context was lost and the user had to log in again. This is why we began using local storage to begin with to fetch user data from the client side storage when a page was refreshed. But since it wasn't secure we instead implemented tokens in local storage so that user data was encrypted in the storage.

Another challenge is that graphql doesn’t really support uploading files. This meant that we instead had to send a post request to our express server and use mutler to store the images to be displayed on the respective profile pages for prosumers and the manager.

It was also a challenge to understand the state and lifecycle of react components. It would be confusing and sometimes complicated to retrieve data before the page rendered. Another challenge we faced was when changing from one component to another, the previously visited one did not stop running. This was happening since we were fetching data periodically in every component and we never explicitly stopped fetching data when the user switched to another component. This would eventually lead to the browser freezing and not responding after some time.

Understanding the graphql framework
Persist context when refreshing page
Tokens
Image uploading
Understand react cycles.

Future work:

The most important work for our system is to implement authorization in our api. So that only a user with a verified token can request data from the api. This was something we realized too late to fix but is a major security flaw with our program because anyone can send a request to the api and receive and modify any information.

Something we would have liked to be using is graphql subscriptions. Here the server maintains a connection to the user via a socket. If a specific event happens in the backend, such as a prosumer’s data changes it sends this data directly to the client in real time.

We would also like to better structure our fetch requests. At the moment they are bunched up in our component files. Some components need multiple fetch functions so it would be easier if we made a general fetch request to the api that all components used and the only thing that changed is the request body that is sent to the fetch function and what parameters it should send back.

Improve the authentication page to send errors if user is not authenticated and also create a more distinctive sign up page.

References:

Academind tutorial:
https://www.youtube.com/playlist?list=PL55RiY5tL51rG1x02Yyj93iypUuHYXcB_

Jason web token for npm
https://www.npmjs.com/package/jsonwebtoken

Self signed certificates for https express server:
https://flaviocopes.com/express-https-self-signed-certificate/?fbclid=IwAR04j3ZMxWNXrkn2UObieKGXQWjQrO-7BtiHk1jDRgHb3BvU6QZU3Y88dKo

Mutler middleware for image uploads:
https://www.npmjs.com/package/multer

React documentation:
https://reactjs.org/

Graphql documentation:
https://graphql.org/learn/

NodeJS documentation:
https://nodejs.org/en/docs/

Bcrypt documentation:
https://www.npmjs.com/package/bcrypt

MongoDB documentation:
https://docs.mongodb.com/

Mongoose documentation:
https://mongoosejs.com/docs/api.html

Express documentation:
https://expressjs.com/

Appendix:

Time logs:

https://docs.google.com/spreadsheets/d/1iStLwIFMof4_2eU7BHD7HObCmhsrgj4h0bCxysPNitk/edit?usp=sharing

Time report Analysis:

Much of the development process went rather smooth but we were struggling initially when setting up the project as we had not used REACT, Mongodb or NODE JS before. As a result of our lack of knowledge much of the bugfixing we encountered were related to timing errors and trying to understand the asynchronous nature of NODE. Much effort also went into understanding how to design the graphQL API and writing the queries and resolvers that would be required to deliver a product that would be on par with the requirements of this course.

During the later part of the development we discovered that we had to figure out how to make the tokens persist on page refresh and also to make sure that the routing on the page worked correctly. We also discovered that file uploads were not supported by graphQL which led to a significant amount of time had to be spent on finding a work around.

Contribution for each member:

In the beginning of the program we worked together a lot. This was in order to get on the same page with what the structure of our project should be. Also neither of us had worked on an api before so we deemed it necessary that both of us worked on it together. This was so both of us gained knowledge for the future and so both understood how it could be used. After most of the api was finished we parted ways a bit. Thomas started with setting up a rudimentary front end and basic functionality and routing. Also restructuring and making our api easier to follow and modify for the future in case something was missing for the future. Meanwhile Hector worked on the simulator implementing most of the functionality so that we could fetch the incoming data to the front end. After this we mainly split up so that Thomas worked on the manager pages and Hector worked on the prosumer pages.

Grade:

I think we both put in a lot of effort and considering that most of the languages used was new to us it was a challenge to accomplish all the requirements. This in itself led to us gaining a lot of knowledge which we both appreciate. We both feel like both of us put in a lot of work and it resulted in a webpage that runs very smoothly. Therefore we feel like a fair grade would be a 4.

System instructions:

GitHub link to the project:
https://github.com/tohsic-7/DynamicWebSystems

In order to use our system it is necessary to first enter the following url:s and validates that our servers use self assigned certificates.
https://130.240.200.73:3000
https://130.240.200.73:4000

To log in as a manager to the system use the following credentials:
Username: manager
Password: manager

The following users have already been added to the system:
Username: Pelle
Password: Pelle

Username: hector
Password: hector

Username: thomas
Password: thomas

In order to create a user press the “switch to signup” button. Then write in a username and password. Then switch back to to login and log in as that user.

API documentation:

All of the documentation for our api can be found in the following link:
https://130.240.200.73:4000/graphql

Press the docs tab and it will list all of the available resolver functions. It is also possible to try out the resolvers here if it is desired.

Completion:

We have added the electricity price to the values that are available at the prosumer page. We have also added functionality that allows the manager to update the prosumer credentials. This can be done by following these steps:

1. Navigate to the users page (when logged in as manager).
2. Press the green “Visit” button next to one of the displayed prosumers.
3. Some of the selected prosumer’s values are now displayed, press the blue button “Update Credentials”
4. Enter the new credentials in the modal that pops up.

We have also continued to build upon the simulator. Now the energy consumption increases during the two first and last months of the year e.g. during the winter. Previously we also designed the simulator to randomly assign electricity to the households present in the network rather than iterating through a list of all households for example. This results in a more realistic

simulation of blackouts since the same household will not always be affected. If we were iterating through a list, the last house would always be affected.