# Report

## Design of Dynamic Web Systems D7011E

Mikael Alanko - mikala-3@student.ltu.se
Rickard Holmberg - richol-6@student.ltu.se

Date: 2020-1-23

# Introduction

The system works by having an html page that presents the user with a frontend. This frontend connects to the API running locally using a socket, which fetches data from the simulator. To protect information and the service, logging is done, an administrator account exists and passwords are hashed. Data is sent over https and captchas at least hinder bots.

In the background the simulator fetches real temperatures and calculates a base wind speed for the day. Using this it then calculates a new wind speed for everytime it runs. It then calculates the power that is generated from each household from a wind turbine. The power that is used by the household is based on real values and depends on the temperature outside. If the turbine generates an excess amount, some is stored based on the user's preference and some is sold to the market. Otherwise some is bought from the market (powerplant or wind from other users) and/or taken from the buffer.

The API connects the frontend to the simulator and allows data to be sent over. It also makes it possible to change settings in the simulator.

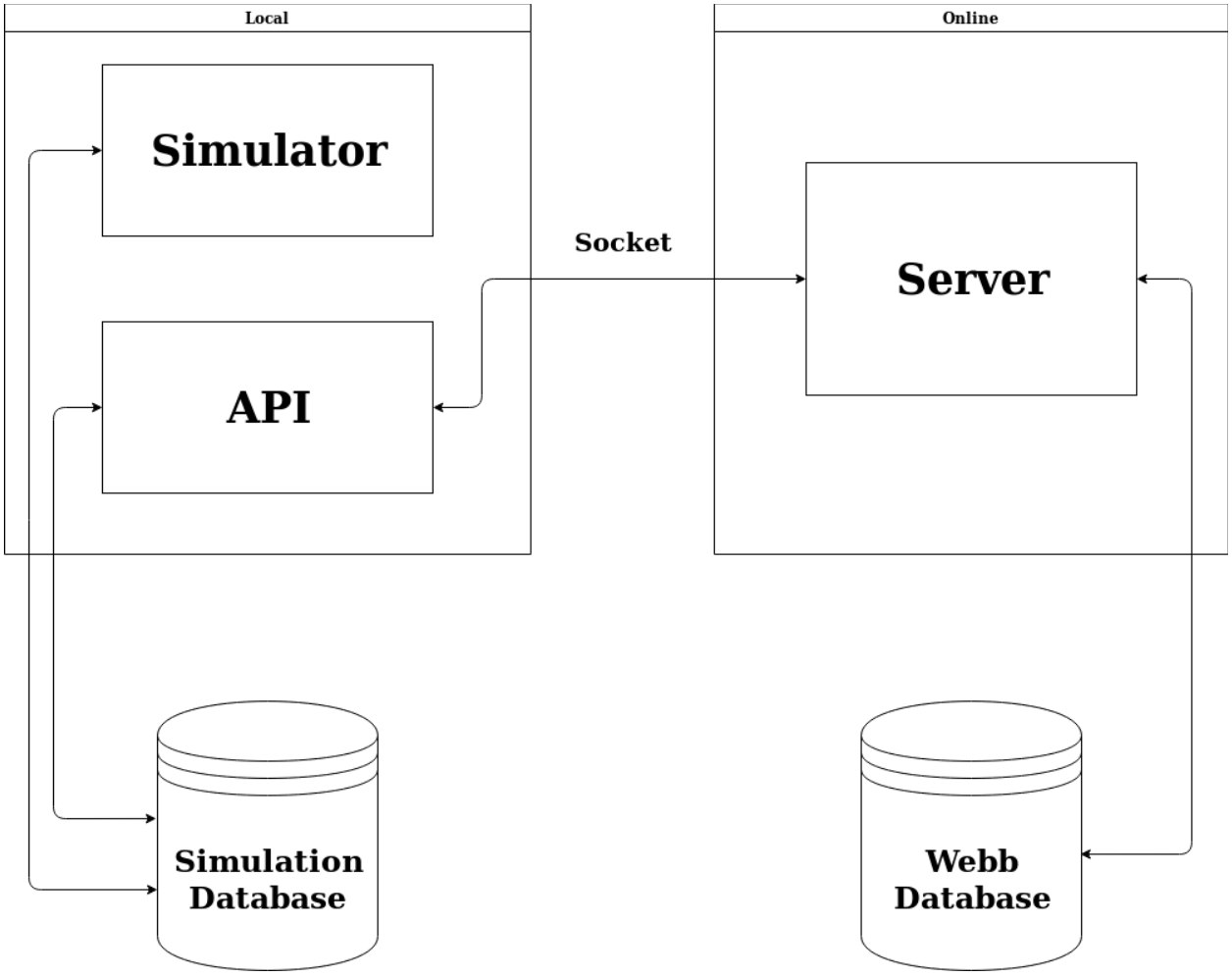


*Image 1 : Final design of system*

Here in image 1 it is clearly shown that the system is split in two, with one way of communication between them. Furthermore the lack of a direct relation between the API and Simulator is also displayed.

# Design choices

The design basics comes from the requirements of having an API, a simulator and some way to see the data. Using these requirements we made a simple diagram (as shown in image 1). Early thoughts included splitting the api and simulator as much as possible. This makes them independent and limits the risk of chain failure. The API:s task is just to fetch data and make small changes to the database, such as settings for the simulator.

Later on the decision to have separate databases arose. One for the simulator that we already had, and the other for the web server. Splitting these up makes for easier database structures and enabled us to make use of more tables. Viewing the tables and understanding the structure easier is also the biggest upside of this. One drawback is that it makes it impossible to make queries using data from one database in the other. This would make things easier instead of having to send this information over the socket from the API to the simulator or vice versa.

To aid in security the simulator and API are kept from the online part of the system (web server). The split of the database also aids in this. Using a local run API has the added benefit of not needing encryption which speeds up the data transfer and queries. Another upside of this is that it is not possible to pry and try to fetch data from the API without permission.

Another thing we added to aid in security is using https to encrypt data over the web to the server. Since we did not want to pay money for a real certificate we used a self-certificate. The drawback of this is that the browser will send a warning to the user of the potential risk of snooping. While this is possible, to use a sort of phishing attack, we didn’t want to pay money for this project.

In the new version of the system (1,1) a config folder exists that changes settings in the API, simulator and server. This includes the ability to change ip address and port to enable it to run on another system with only a few changes in the configs. Making this change should have been included in the original version but it was just forgotten in the design phase.

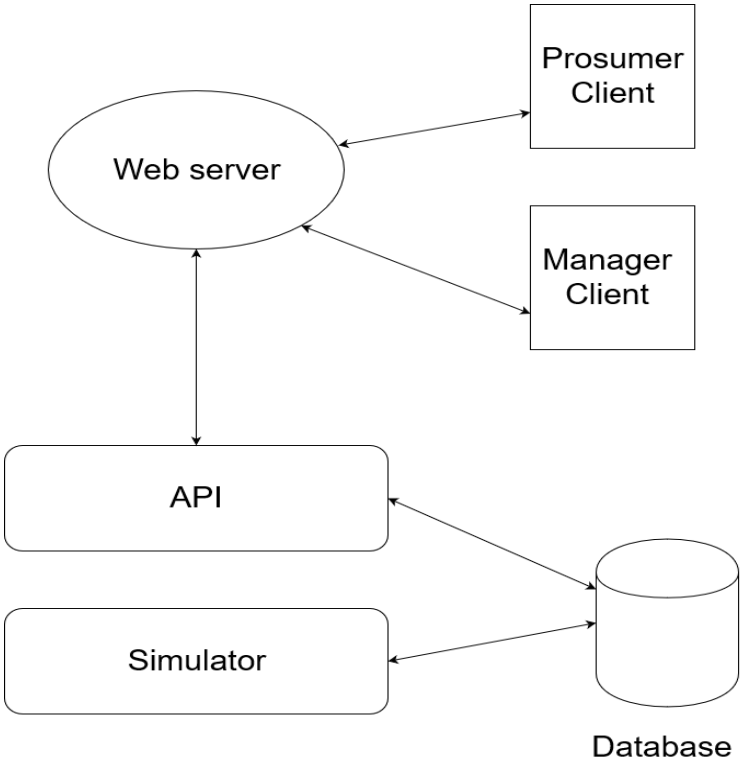


*Image 2 : Early design of system*

Here in image 2, even the early image of the design the split between the API and simulator is still present. Although, the database for the server doesn't exist, and running part of the system locally isn't considered.

# Analysis

## Scalability

Vertical scaling means that the server hardware needs to be upgraded unlike horizontal scaling that adds more machines into the existing server. Since vertical scaling is dependent on the servers hardware it is a lot more risk of downtimes. Caching is a good way to speed up the website, it could be used on the client side and the server side, for example the browser cache could store all files needed to display the page for a while instead of waiting for them every time visiting the page. The server can store the requests for a while, which makes the request from client to server quickly.

Since the server is hosted through Amazon the network problem with scaling can be resolved by upgrading the service to be able to handle a larger amount of users so this part isn't the bottleneck.

Our current solution scales with the help of the restful-api which is stateless, the server won't need to store anything across requests.
To improve the scalability we could minimize the sql queries that includes many join from different tables. To decrease the risk of downtimes we could change the database from MySQL which is using vertical scaling to MongoDB that using horizontal scaling.  Also caching could be used for displaying images to the users, it is already take some time to load pages with images so this problem should disappear with caching.

## Security

While security is light on some fronts we took measures to at least protect information and secure again bots. For security the following is implemented:

### Encryption

All traffic to the server through the internet is encrypted using https, this ensures data is protected from packet sniffers. Although since we use self-encryption this also has some issues.

### Hashing

Passwords are hashed in the database to prevent leaks of usable passwords. It is not possible to somehow send the hashed password to the server and get logged in.

### CAPTCHA

To protect from being able to make several accounts using bots a basic captcha exists when creating a user. It is not possible to check for the answer in the source code for the website. All checks for the captcha is done on the server.

### Token

To be able to access the logged in features a token is needed. This token is only generated when you log in. The token is encrypted using a key in the authentication control. When signing in as an admin a flag is set in the encrypted token. This allows us to easily check and verify if the user is signed in and what user id and whether they are an administrator or not.

## Advanced features

In the simulator we have implemented that the weather outside affects the energy requirements for households. This can be changed using variables in the simulator. Currently 60% is affected by temperature, at 25 degrees celsius no power goes to heat and at -30 the maximum goes to heat (also configurable). The scale between max and min is linear. We also fetch real temperature values from SMHI.

## Challenges

The most difficult part to build this system has been the simulator part, no one of us had prior experience in javascript or any other asynchronous language.
It was a challenge to build the simulator with javascript since we did not understand how to work with callbacks. Further adding to  the issue the way the simulator works is dependant on that the previous queries finish before other functions run. This is what caused the need for callbacks since the await part of asynchronous functions didn’t work as we expected. Also because the simulator has ended up like a nested loop it is very time-consuming to debug that part.

Another challenge was that we wanted to run the API on a local instance and connect to it with some mean. Firstly we tried to use express since this is what we previously used on the server.

As we expected this to just work we continued with the other parts. Later when the finally connecting the API and server this proved incorrect, everything didn't work using express. We had to change to socket.io and use events within the socket. Events proved to work nicely and effectively, so we preferred this over the old express version.

# Future work

For the future it has to be done a lot of front-end development, we did not focus on that part at all, instead we tried to display the values in a proper way. Another important part that should be done is the password security, like it is now it is possible to have a password including only one letter. Rollback/Transactions to the database should be implemented, many sql queries depending on the previous one so if the previous query somehow fails the next query will use wrong values. With rollbacks it will instead fail like a unit, this should be done so the simulator running with the correct values.

With more time available on the project a more decent frontend would also be nice. This would improve the aesthetic and add more features. Such as a chat or better or more robust user interface as we feel it is lacking. If we had considerably more time we would probably just redesign the simulator since when we started a clear idea didn't exist yet. The functions could be ordered better and the callbacks could be more clear to follow. Right now, it's not very fun and enjoyable to find errors, even though we print a lot of information during run-time.

# Appendix

Our time reports does not included 200 hours each which we were expecting to put on this assignment. But we have not included all the time put into discussions about the assignment. Most of the time was spent on debugging and work with the simulator, since we had no prior experience with javascript we really had to spent a lot of time to understand callback functions. We have been working together more or less all the time that we have been working on this assignment. This means the work on all parts of the system is basically split in half.

Since we did not spent 200 hours and didnt made almost any of the advanced features a fair grade for both of us should be 3.

Github link to the project: https://github.com/drakcir97/M7011E