M7011E — A dynamic web system

Aron Widforss (arowid-6@student.ltu.se)

Introduction

This report describes a small web system simulating a market of small-scale producers of electricity. By first considering how a real system of its kind would look, it is easier to reason about what different parts should exist and what they should do.

Imagined real world system

- User web client
    - Each small-scale producer is considered a user of the system. They should be able to access a client that lets them interact with the system in a well-specified way.
- Administrative web client
    - The system has to be administered in some way. A manager should therefore be able to access a different client than the ordinary users, where administrative actions can be taken.
- Producing units
    - In this case the producing unit would be a small wind turbine, reporting statistics to a central server. The turbine is owned or leased by a user of the system, but the user does not have control over the turbines computer.
    - The manager is also in charge of a special producing unit, a coal power plant.
- Database
    - The data has to be stored somewhere, so we need a database.
- Web server
    - This server would be tying the whole system together, talking with the clients, the producing units and the database, making sure each party gets the information it needs and nothing more.

Simulated system

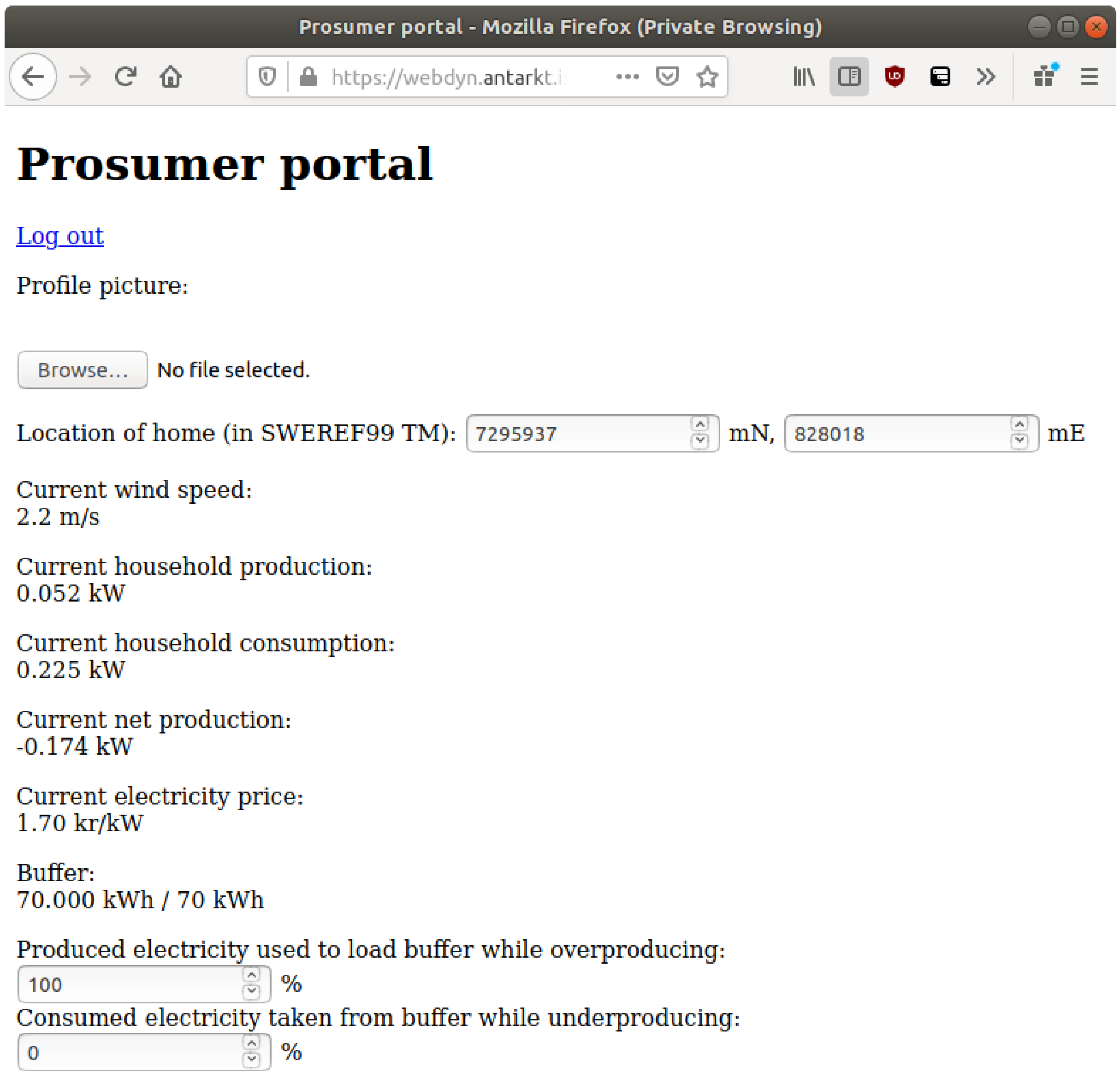
When everything is simulated, we can rationalize and don't need as many parts.

- User web client
- Administrative web client
- Database
- Web server
    - Producing units
        - The web server can simulate the producing units and just write the result to the database. It is tempting to let the web clients simulate the producing units, as they are the one displaying the data, but this would lead to several problems. Most severely, by running the simulation on the web client, the users could fake their produce, by modifying the client code, as it is running on untrusted hardware, the users' computers. But it would also lead to problems where a user's wind turbine would not be simulated if that user were not logged in. And if a

        user would be running multiple instances of their client there would either be a race condition or a case of vastly overestimating that user’s produce, depending on how the results of the simulation were communicated to the web server. Therefore the simulation of the producing units of all users must be run centrally.

- Weather simulator
    - In the real world, wind turbines are subject to real wind, an external variable not part of the web system. However, in a simulation, we need an input to the simulation of the producing units to create this external factor. Therefore we need some kind of wind simulator over the area where we have users.
- Web server
    - The web server still has to perform its regular duties, connecting all parties together.

User web client


Prosumer portal - Mozilla Firefox (Private Browsing)

https://webdyn.antarkt.i

Prosumer portal

Log out

Profile picture:

Browse... No file selected.

Location of home (in SWEREF99 TM): 7295937 mN, 828018 mE

Current wind speed:
2.2 m/s

Current household production:
0.052 kW

Current household consumption:
0.225 kW

Current net production:
-0.174 kW

Current electricity price:
1.70 kr/kW

Buffer:
70.000 kWh / 70 kWh

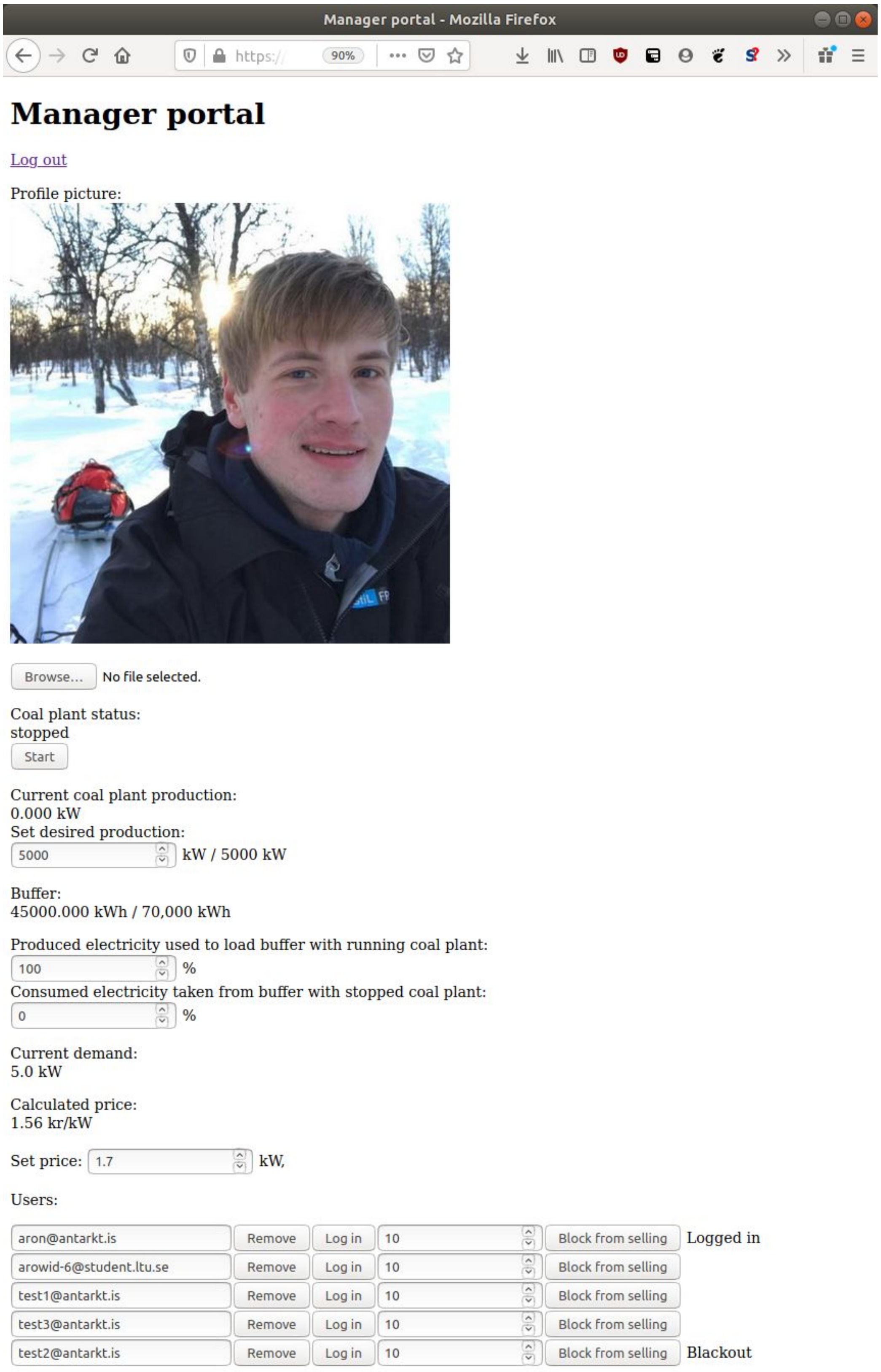
Produced electricity used to load buffer while overproducing:
100 %

Consumed electricity taken from buffer while underproducing:
0 %

Figure 1: User web client

The user need information about current production, pricing and weather parameters. They can also control a small energy buffer coupled with the simulated wind turbine as well as choosing a profile picture and setting the location of the home. The last option is for the simulation of the wind to be correct.

Administrative web client



Manager portal

Log out

Profile picture:

Browse... No file selected.

Coal plant status:
stopped
Start

Current coal plant production:
0.000 kW
Set desired production:
5000 kW / 5000 kW

Buffer:
45000.000 kWh / 70,000 kWh

Produced electricity used to load buffer with running coal plant:
100 %
Consumed electricity taken from buffer with stopped coal plant:
0 %

Current demand:
5.0 kW

Calculated price:
1.56 kr/kW

Set price: 1.7 kW,

Users:

aron@antarkt.is	Remove	Log in	10	Block from selling	Logged in
arowid-6@student.ltu.se	Remove	Log in	10	Block from selling	
test1@antarkt.is	Remove	Log in	10	Block from selling	
test3@antarkt.is	Remove	Log in	10	Block from selling	
test2@antarkt.is	Remove	Log in	10	Block from selling	Blackout

Figure 2: Administrative web client

The manager can control the special coal power plant, and a very large buffer connected to it. The manager can also block other users from selling electricity, remove users and log in as users. As in the case of the user, the manager can set a profile picture.

Design choices

Database

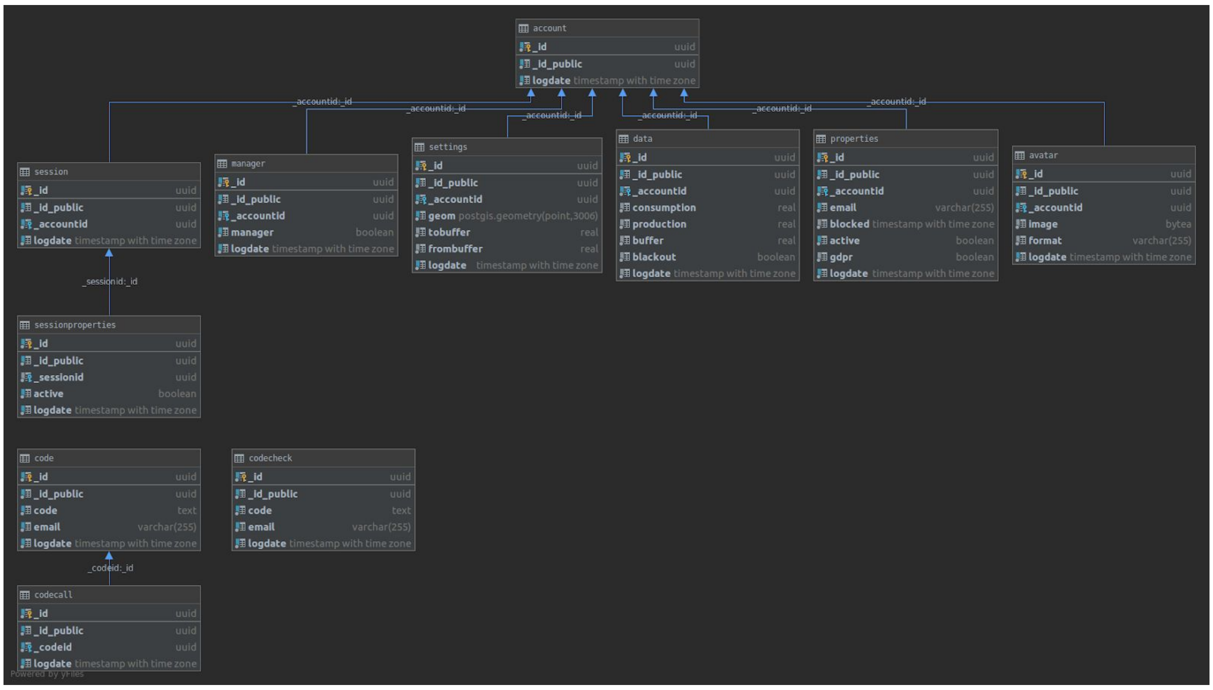


Figure 3: Hierarchy of the most advanced schema in the database, “account”

The database is normalized to a certain degree, see figure 3. In an effort to make it possible to enable rollbacks in the future, updates of most tables are not done using UPDATE. Instead, a new row is simply added to the table using INSERT. The web client then uses a VIEW that filters all old data, as well as joining the normalization. However, tables that are updated frequently (simulation data is updated every second or so, for every user) uses a traditional UPDATE, as they would quickly grow unbearable large if they did not. Insertion is done using procedural SQL functions, often using ordinary JSON as input.

This makes for a very clean SQL interface on the web server side, but is rather messy on the database side, as procedural SQL is very verbose and hard to make succinct.

The web servers’ database account only have access to the views and the PL/pgSQL functions, and can’t access historical data. This also means that it can only perform a certain number of predefined operations on the data, as only the predefined functions have access to the actual tables. The web server can interact with these functions even though they perform privileged operations since they are defined with SECURITY DEFINER.

Web clients

The web clients are very quick and dirty. They are implemented as one file each, with inline JavaScript. The scripts are rather unstructured and bloats the namespace more than necessary.

Web server

The web server offers a small API for web clients to use. Only logged in users can access paths prefixed with “/api/”, and only managers can access paths prefixed with “/api/manager/”. The web server code is relatively well-done, with the exception of the router, which could be refactored into something more easy to overview.

A module of the server handles the wind simulation. This is rather heavy calculations and probably stands for most of the processor time of the server. Simulation of the actual wind turbines and coal power plant is done by a small function in the entry point of the server.

Scalability

Scalability most often means horizontal scalability, i.e. adding more web servers and databases without adding more performance to the individual instances.

Database

A maintenance function purging old, unused data from the tables would be needed for large-scale production.

Newer versions of PostgreSQL does support sharding, but this would need some restructuring of the database structure. However, it would not be very hard to horizontally scale the database.

Web server

If the wind simulator were put on its own machine, a lot of processor time would be freed up to handle incoming requests. The same is true for the wind turbine and coal power plant simulator, but to a lesser degree.

The only state kept on the web server is an object keeping track of what users are logged in (that have pinged the server during the last 10 seconds). If this data was moved to the database, we could add any number of web servers to load balance on without affecting functionality the slightest.

Security

Account setup, removal and login

There is no separate account setup form. Instead, an account is created automatically when a user authenticates using an email from a trusted organisation (email domain). However, we want to be able to ban users misbehaving on the platform. Therefore, if a user is removed by a manager, it will not be able to log in anymore. This scheme depends on the assumption that users do not have access to unlimited email addresses from their organization.

The login is passwordless and send a one-time, 6-digit code to the user's email, which they authenticate with. If it is the first login of said user, an account is created, otherwise they are logged into their existing account. The sending of the email is done using an external email provider. If the email fails to be sent to the ESP, it is retried using exponential backoff a number of times.

The one-time codes are valid for 15 minutes and are rate limited. A new user can only get one code every 15 minutes, and is only ever issued 3 codes before authenticating by entering the code at the login page. An existing user can get a new code every 10 minutes. There is also a global rate limit, where there can only be 150 valid codes at any given times, and only 300 codes issued in a rolling 24 hour period.

The authentication is also rate limited, where a user can only try to authenticate 15 times per 15 minutes. The global restriction is 150 times per 15 minutes and 300 times per day.

These rate-limits helps with two things. The login system can not be used to spam email addresses at the trusted organization, as it will only send 300 emails a day. It also makes it harder to brute-force the system. One way to brute-force would be to target a specific user, which is obvious to rate limit. But as the system automatically creates accounts for authenticated email addresses, we could try to fire hose any number of different email addresses to the system and have a 50 % chance of creating a new account by just trying a static code and 50,000 different email addresses. This is what the global limits try to stop.

The adversary still have a 50 % chance of breaking in if they try for 167 days. However, this means that the global rate limit will be completely consumed by the attacker, and users will notice. A more careful adversary may only try 150 codes a day, which gives a 50 % chance of breaking in in a year. This is probably on par with most other services out there.

Email is an inherently unsafe protocol, but one may argue that this solution is more secure than ordinary services password-solutions. This is because, no matter how good password practices the user employ, the service most often have an email reset, which functions similarly to the authentication described here. This means that, by not offering passwords, the service actually have less attack surface than most other services.

A more secure option would be to use multiple factors. Here, a password makes for a good second factor, but only if you need to provide a password AND a one-time code, not a password OR a one-time code.

Database

The web server does not have full access to the database. It can only access certain views and certain functions. As historic data is mostly kept behind the scenes in the database, as described above, an attack from a compromised web server would not destroy the data, only disrupt the service temporarily.

The PostgreSQL server is not exposed to the outside world. The web server uses an SSH tunnel into a non-privileged account to access it. This means that an adversary could not brute force the database login.

Encryption

The clients talk to the web server using HTTPS with a certificate signed by Let’s Encrypt. The web server talks to the database using SSH with keys installed on provision. The email is sent to the email service provider over implicit TLS. The weakest link is when the authentication email is in the wild, as SMTP is known to be vulnerable to downgrade attacks.

Advanced features

Wind simulator

The wind simulator was the most interesting thing to write in the entire project. It simulates wind over a raster that is supposed to represent Norrbotten county. It is multi-layered, the bottom layer has a resolution of 1 km, then above that there is a 10 km- and 100 km-layer.

All pixels in all layers are instantiated using a Rayleigh distribution. The simulation is then stepped forward. In each step, every pixel gets a new value that is a weighted mean of itself, its neighbours, its parent (the closest pixel in the above layer) and a normal distribution. This means that each pixel has temporal autocorrelation (as it is related to its previous value), as well as spatial autocorrelation (as it is related to its neighbours’ previous value).

After spinning up the spatial autocorrelation works both locally and globally, as the parent pixel may bring in weather from afar rather quickly. It is rather fascinating to see it in action, even if it is just a printout of the 10 km-layer to the terminal.

When the wind speed is requested for a certain point, the speed is interpolated from the 4 closest points in the 1 km-layer, weighting the points proportional to their relative distance to the requested location.

Image upload

While the web clients only allow for upload of file with a suffix indicating JPG and PNG files, the server does not care for file names. Instead it feeds an uploaded image into GraphicsMagick for validation. If GraphicsMagick says that the format is either JPG or PNG, the file is accepted, even if the uploaded filename is something as wrong as “sweden.geojson”. The image is then resized into a VGA resolution bounding box and inserted into the database.

Passwordless login

Please refer to the security section.

Challenges

The wind simulator was the most challenging to write, and also the most interesting.

Future work

Database

The PL/pgSQL functions should be rewritten to be linked to INSERT, UPDATE and DELETE actions on the views visible to the web server. This would probably lead to better structure of the functions’ code.

As mentioned earlier, there should be some kind of maintenance function that purges old data.

Web client

The web client JavaScript code should be lifted out of the HTML files, and be modularized and compiled by something like Webpack. The JavaScript code also need a major refactoring and better structure.

Wind simulator

The wind simulator already interpolates spatially. But right now, it steps forward in time. A better solution would be to simulate a couple of points forward in time, and interpolate using those, probably with a weight proportional to the square of the temporal distance or something like that.

References

Ideas and inspiration for the main repository code are taken from a range of earlier personal projects of the author. The database structure and authentication is inspired by an avalanche observation database, Njunis, written by the author.

The deployment scripts in their separate repository are a fork of nodejs-server-ansible-playbook.

Appendix

Time analysis

The author have obviously procrastinated, as the time spent is clustered around the meetings had with the course lecturers. The time spent working seems rather efficient though, as the author has spent significantly less time than estimated by the syllabus. The hours seems rather evenly distributed between the tasks given.

Grade

The system is developed according to the principle of least effort and some of the code is barely legible. However, there are some work done beyond the basic specification, such as a somewhat decent wind simulator, an actually reasonable image upload, passwordless login, and a somewhat complicated (though not very efficient) database structure. The author thinks the work is worth at least a passing grade.

Release

The GitHub release is available at https://github.com/widforss/m7011e/releases/tag/1.0.0. Provisioning scripts and instruction are available at https://github.com/widforss/m7011e-deploy.