Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR'S THESIS

Study program/specialization:	Spring semester 2023
Bachelor in Computer Science	Open

Authors: Eirik Bakkan, Sune Selchow

Course coordinator: Tom Ryen

Supervisor: Nejm Saadallah

Thesis title:

WPSim: Web Application for Wind Park Simulation

Credits: 20

Keywords:	Pages:
Web application, deployment Frontend, backend Wind park, wind turbines Wake simulation	92 Stavanger 5 June 2023

Contents

Acknowledgements

Special recognition is given to the project supervisor, Nejm, for guiding the team under the project. Nejm demonstrated great knowledge and insight, and the information provided was invaluable. In addition, the support offered during a delay due to a team member's health-related circumstances, has significantly contributed to the project's success.

Summary

This thesis presents the "Web Application for Wind Park Simulation" project, a web based, interactive wind park simulator. The project was completed in motivation to aid in renewable energy planning, specifically for wind farms, in response to global warming.

Advanced technologies was used to create a scalable application, that can be used on different platforms. Despite computational challenges related to multi-variable computing, WPSim successfully achieved initial objectives.

The project was deployed, offering on-demand wake simulations, with future enhancements already being considered. This project serves as a step towards making renewable energy planning tools more accessible and efficient.

Chapter 1

Introduction

1.1 Assignment description

"Web Application for Wind Park Simulation" had the goal of creating a web stack for the FLORIS (“National Renewable Energy Laboratory”, n.d.) wake simulation package.

The assignment had the following main tasks:

- Research and choose appropriate technologies.
- Implement backend services.
- Implement frontend services.

1.2 Project goals

Given the project, the following goals where deemed important:

- Good user UX interaction.
- Low overhead, lightweight application.

  - Of high importance due to the higher processing power required for multi-variable simulation.

- Authentication using external OAuth providers.

  - OAuth provides class leading level of credential security, without increasing development overhead.

- Saving of user simulation configurations.
- On demand FLORIS power production simulation.
- Visualisations of FLORIS result data.
- Downloadable FLORIS result data in appropriate file formats.
- Deployment of stack to the web.

By focusing on these goals, development was streamlined, reducing dead time.

1.3 Motivation

Global warming is a serious concern. The UN stated that "Fossil fuels, such as coal, oil and gas, are by far the largest contributor to global climate change, accounting for over 75 percent of global greenhouse gas emissions and nearly 90 percent of all carbon dioxide emissions." ("Renewable Energy – Powering a Safer Future", n.d.). Further stating that "The science is clear: to avoid the worst impacts of climate change, emissions need to be reduced by almost half by 2030 and reach net-zero by 2050." ("Renewable Energy – Powering a Safer Future", n.d.).

Developing a tool that facilitates quick online estimation simulations for wind farms, may accelerate the transition from fossil fuels, to wind farms.

Efficient wind farms may also reduce the cost of energy, by reducing reliability om fossil fuel trades. Figure 1.1 shows an abrupt increase of electricity prices in Europe, even if compensated for inflation. Statnett lists the increased price of gas and coal, affected by Russo-Ukrainian war, as one of the main reasons for the price increase ("Om strømpriser", 2023).

Reducing reliance on fossil fuels may help withstand unforeseen situations, and reduce greenhouse gases, both of which may be fulfilled by investing in wind energy. The web application developed in this project, may contribute to the first step towards building a wind park, if only showcasing the theoretical power production.

1.3 Motivation

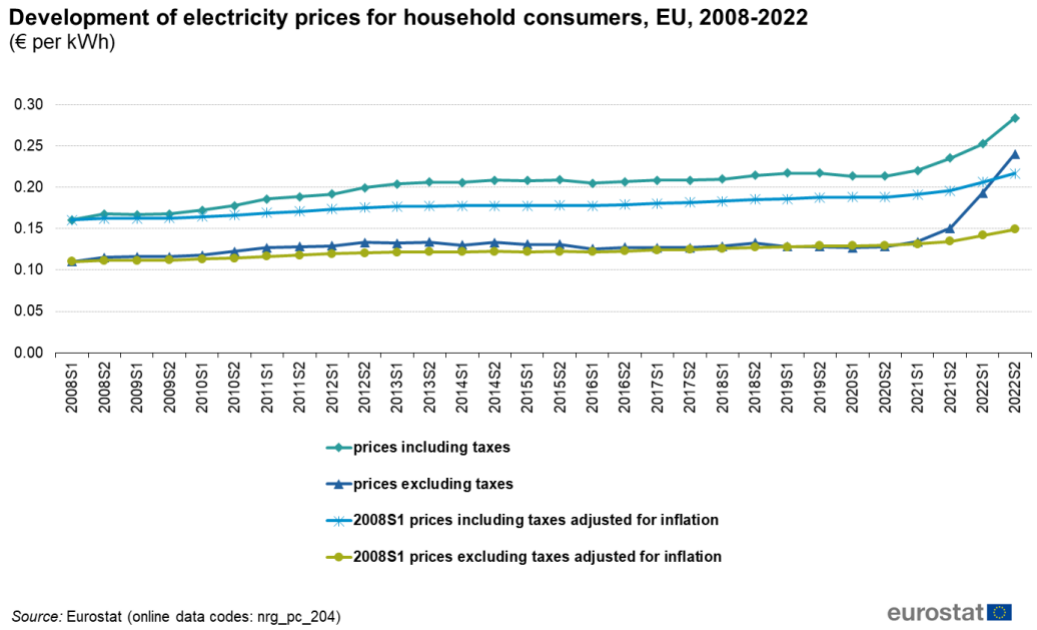


Figure 1.1: Development of electricity prices for household consumers, 2008-2022. The figure is taken from ("Energy statistics - quantities, annual data", 2022).

Chapter 2

Background

This chapter contains theory and background information.

2.1 Wind turbines

The Earth's atmosphere experiences multiple variables, including temperatures, land height differences, gases and thermal mass differences. Variables of which creates differences in air pressure, resulting in a natural phenomena, called wind. Wind turbines can harness the atmospheric air pressure difference, by converting the kinetic energy in wind to potential electric energy. This is done by blades converting wind to angular kinetic energy, which in turn power a generator. The generator uses the rotational input, to energize conductive coils, resulting in electrical energy output. ("Wind Energy Basics", n.d.)

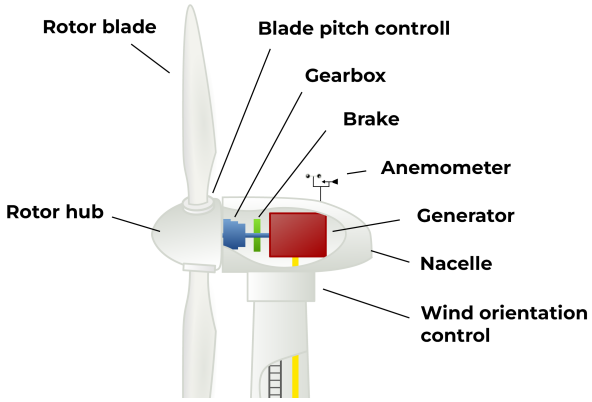


Figure 2.1: Wind turbine schematic, Nordmann, 2014. Image used under CC BY-SA 3.0. Cut and modified.

2.1.1 Wake effect

Energy can only be transformed or transferred from one form to another, following the laws of physics. Wind turbines transforms energy, resulting in a wake field, with lower energy, turbulent wind. Wake fields significantly decrease wind turbine energy output, and are one of the most important factors to consider, when designing wind farms. (“Wake Effect”, 2003)

Figure 2.2: "Horns Rev 1 offshore wind farm 12 February 2008, seen from south", Steiness, 2008. Courtesy: Vattenfall.

2.2 Work flow

The research and development for a web based simulation service, such as WPSim, requires an efficient and streamlined workflow. This is to ensure an throughout research phase, and a efficient development phase. Essential tools utilized were Discord and GitHub.

2.2.1 Discord

Discord is a real-time communication platform ("Discord — Your Place to Talk and Hang Out", n.d.), that enables efficient collaboration within a team. For the project, Discord served as a digital workspace, offering several benefits:

- **Immediate communication:** Discord being a real-time communication platform, facilitated quick and effortless interaction between the team members. This enabled quick clarification of doubts, sharing of ideas, and resolution of issues.
- **Multimedia communication:** Usage of voice, video, and text communication, allowed the team members to work from home, whenever time was available.
- **Multi platform:** The Discord platform is available on both computer and smart-phone devices, giving team members the opportunity to respond and get information quickly.
- **Organization:** Discord uses a channel system, which allowed creation of specific topic channels. The information found and given during the project, was organized in the channel system, preventing information from being lost, and allowing for easy references to previous discussions and findings.
- **Integration:** Compatibility with other platforms, like GitHub, allowed automatic informing on code changes.

2.2.2 GitHub

GitHub, an integral part of the development of WPSim, is a web-based hosting service for code and version control. GitHub was used to store, manage, and track changes to the code. Where key features utilized included:

- **Version control:** Being GitHub's primary functionality, version control facilitated tracking and reviewing changes made to the code. Preventing unintended modifications, while also informing the team on changes done.
- **Branching and merging:** Branching enabled team members to code on different parts of the project simultaneously, without affecting the main codebase. Once a change was ready and tested, it could be merged back into the main branch, maintaining the code's integrity.
- **Collaboration:** Using pull requests, we could review code before merging, enhancing code quality.

By combining the real-time communication capabilities of Discord with the version control and collaborative aspects of GitHub, the team could streamline the workflow and increase productivity in the WPSim project. This ensured efficient progress and ultimately led to a successful project.

2.3 Backend

2.3.1 Python

Python is a programming language, with code readability and ease of use, as the main design philosophy. Code is dynamically typed and runtime garbage-collected ("Python Language Documentation", n.d.), further lowering its barrier of entry.

WPSim uses Python in the simulation backend, due to FLORIS, the simulation framework, being written in Python.

2.3.2 FLORIS

FLORIS (FLOw Redirection and Induction in Steady State) is an open-source tool, developed by NREL ("National Renewable Energy Laboratory", n.d.), a U.S. based center in renewable energy, and energy efficiency research. FLORIS was developed as computational tool, aiding in predicting, analyzing and optimizing wind farm performance under varying conditions. The framework is written in Python.

Alternative simulators

Alternative simulators, like OpenFast (National Renewable Energy Laboratory, n.d.-a) and TurbSim (National Renewable Energy Laboratory, n.d.-b) are also essential tools in wind energy research, although they serve different purposes. FLORIS operates at the wind farm level, focusing on wake interactions and wind farm optimization, while OpenFAST and TurbSim operates at the turbine level, focusing on the detailed physical response of individual turbines to their operating environment. FLORIS is therefore ideal for web deployment, due to the relatively low computational requirement when focusing on wake computations.

WPSim uses the following features of FLORIS:

- **Performance predictions:** FLORIS simulates wake effects between turbines, which are disruptions in the wind flow caused by upstream turbines. Wake simulation is key in WPSim's ability to estimate the power output of the wind farm under different wind conditions.
- **Optimization capabilities:** FLORIS provides a framework for optimizing wind farm layout, with the goal to maximize power output and minimize negative wake interactions. WPSim utilizes this feature to allow the client to see the best case scenario, when no geographical limits are applied.
- **Flexible and extensible:** FLORIS is coded in a modular way, which allowed WPSim to be coded to in the future accept additional models and algorithms from the FLORIS framework.

2.3.3 FastAPI

FastAPI is a modern, low overhead, web framework for building API's with Python. The framework is based on standard Python type hints, which lowers the learning curve when developing API's (Ramírez Sebastián, n.d.). It features interactive API documentation, while also supporting asynchronous request handling. Two of which are key attributes for developing web-based applications like WPSim.

In the context of WPSim, FastAPI is utilized as follows:

- **Fast Development:** Automatic features, including request validation and serialization, are tools that significantly sped up the development process of WPSim.
- **Asynchronous Support:** Asynchronous request handling, which is crucial for maintaining responsiveness in WPSim during complex loads, allows WPSim to serve multiple requests concurrently, improving the overall user experience.
- **Integration with Python-Based Tools:** As FLORIS, the simulation framework, is also Python-based, FastAPI provided seamless integration. Reducing complexity during development.
- **Automatic Documentation:** FastAPI generates automatic interactive API documentation, aiding in the understanding, testing, and debugging of the WPSim API. Utilizing this feature, debugging and developing the API was done at relative ease.

FastAPI's features and capabilities complemented the WPSim stack. Contributing to the efficiency, scalability, and usability of WPSim.

2.3.4 Uvicorn

Uvicorn is an ASGI (Asynchronous Server Gateway Interface) server implementation, written in Python. Designed to serve asynchronous code, for Python web applications ("Uvicorn", n.d.).

Uvicorn is built on uvloop, an asyncio event loop implementation, and httptools, a collection of http protocol utilities. Allowing Uvicorn, when paired with FastAPI, to deliver high performance, low latency serving of data from FLORIS.

Uvicorn has the following WPSim relevant characteristics:

- **Performance:** Uvicorn uses uvloop, a low-overhead drop-in replacement for the asyncio event loop. Using the uvloop implementation is vital for a responsive user experience in WPSim.
- **ASGI compliance:** Uvicorn is an ASGI server, which allows it to serve FastAPI applications. FastAPI compatibility is essential for our Python-based, FLORIS, backend stack.
- **Concurrency:** ASGI servers support asynchronous request handling, which is crucial for managing multiple simulation requests concurrently in the FLORIS backend. Reducing the visibility of server load for the user.
- **Ease of deployment:** Good documentation, describing incorporation into more complex deployment setups, while also providing flexibility for future scaling of WPSim.

Due to relevant characteristics, Uvicorn was chosen as the server to deploy FLORIS on WPSim.

2.3.5 OAuth 2.0

Open Authorization (OAuth) is a widely adopted open standard for authorization. It allows internet users to authorize websites or applications to access their information on other websites without giving their passwords. This protocol is utilized by major corporations like Amazon, Google, Facebook, Microsoft, and Twitter, to enable their users to share their account information with third-party websites or applications. ("OAuth", 2023) Ultimately, this allows users to log in to other websites or applications by using their Google, Facebook, or other supported accounts. Using OAuth in your application eliminate the risks associated with storing user passwords.

When users can log in without having to create a user account, it simplifies the user experience.

2.3.6 Passport and Express

To utilize the OAuth protocol, Passport and Express were used. Passport is an authentication middleware for Node.js that provides a simple and modular way to implement authentication in web applications. Passport is designed to be unobtrusive, lightweight, and extensible, making it easy to integrate with existing Node.js applications. ("Passport - Simple, unobtrusive authentication for Node.js", n.d.) Different so called strategies are used to authenticate via socials. Express was used to utilize Passport. Express is an open source web application framework for Node.js. It is minimal and flexible. ("Express - Fast, unopinionated, minimalist web framework for Node.js", n.d.)

2.3.7 SQLite

SQLite is an excellent choice for a WPSim to be able to store configurations. It is a small, fast, reliable and full-featured Structured Query Language (SQL) relational database. SQLite is an embedded database which means it is a library developers embed in their applications, rather than a standalone application. It is the most widely used database engine in the world. ("SQLite", 2023) ("What Is SQLite?", n.d.) SQL databases uses tables to store, fetch, modify and delete data. A row in a table is uniquely identified by the primary key which can be one or more attributes in the row. Tables can have relations between them by declaring an attribute as a foreign key.

2.4 Frontend

2.4.1 HTML and CSS

Two of the main technologies when it comes to building web applications are HTML (Hypertext Markup Language) and CSS (Cascading Style Sheets). HTML describes the structure of the pages while CSS determines the styling and layout including colors and fonts, and more. HTML consists of different elements such as headings, paragraphs, lists and tables. (“HTML & CSS”, n.d.)

Code 2.1: HTML elements dislaying "Home" as title and "Welcome" as text.

```
1 <h1>Home</h1>
2 <p>Welcome</p>
```

CSS classes can be assigned to the HTML elements to apply a certain style.

Code 2.2: HTML elements dislaying "Home" as title and "Welcome" as text with CSS classes.

```
1 <h1 class="title">Home</h1>
2     <p class="text">Welcome</p>
```

Code 2.3: The CSS file containing the classes.

```
1 color: blue;
2 }
3
4 .text {
5 font-size: 12px;
6 }
```

2.4.2 React

React is a frontend library that enables developers to construct user interfaces using JavaScript (or TypeScript) and HTML. It is maintained by Meta and

supported by a community of individual developers and businesses. React is widely regarded as one of the most commonly used frontend frameworks in the industry. The construction of interfaces is based on components. (“React (software)”, n.d.) (“React - The library for web and native user interfaces”, n.d.) These are reusable and self-contained and determines the appearance and functionality of a specific part of the interface.

Code 2.4: Component that displays the title "Home" and the text "Welcome".

```
1 const TextComponent: React.FunctionComponent = () => {
2   return (
3         <h1>Home</h1>
4         <p>Welcome</p>
5   );
6 };
```

Code 2.5: Using the component in the application.

```
1 const App = () => {
2   return (
3         <TextComponent/>
4   );
5 };
```

It is much more useful to assign values as props to the component to decide what the component should display.

Code 2.6: Component receiving data using props.

```
1 const TextComponent: React.FunctionComponent = (props) => {
2   return (
3         <h1>props.title</h1>
4         <p>props.text</p>
5   );
6 };
```

Code 2.7: Using the component with props in the application.

```
1 const App = () => {
2   return (
3         <TextComponent title={"Hello"} text={"World!"}/>
4   );
5 };
```

React hooks are functions that add state and other functionality to components. The **useState** function returns a stateful value and a function to update it. When the stateful value is updated, the components using the value is also updated.

Code 2.8: Using the component with useState to control the state.

```
1 const App = () => {
2   const [myText, setMyText] = useState("Welcome");
3   return (
4         <TextComponent title={"Hello"} text={myText}/>
5   );
6 };
```

Another important react hook is **useEffect** which adds side effects to a component, for example fetching data from a source. The useEffect function runs after the component has rendered. It is possible to add state variables in the empty array at the end. When one of those variables change the **useEffect** runs again.

Code 2.9: Utilizing useEffect and useState to fetch data and update the state of the component.

```
1 const App = () => {
2   const [myText, setMyText] = useState("Welcome");
3
4   useEffect(() => {
5     data = fetchData();
6     setMyText(data);
7   }, []);
8
9   return (
10        <TextComponent title={"Hello"} text={myText}/>
11  );
12 };
```

2.4.3 TypeScript

TypeScript is a statically typed, object-oriented, programming language. Being a superset of JavaScript, it adds optional types to the language. Designed for development of large multi-platform applications, which is facilitated by transpiling to JavaScript ("TypeScript", n.d.). While JavaScript is an essential language for web development, TypeScript also brings in the advantage of static typing and enhanced IDE support while reducing complexity in larger codebases.

In the context of the WPSim frontend, TypeScript was utilized and chosen due to the following:

- **Static typing:** Static typing catches errors early during the development phase, resulting in more robust code. It also allowed for type checking during compile, which resulted in identifying potential type mismatches and wrongly setup function calls early.
- **Efficient coding:** TypeScript's static type checking capabilities facilitates enhanced auto-completion, navigation, and refactoring services in IDEs (Integrated Development Environments), which improves the overall development experience.
- **Multi-platform:** TypeScript code is transpiled to JavaScript, ensuring compatibility across different web browsers.
- **Improved code readability and maintainability:** Explicit type annotations improved code readability by making code structure, function contracts, and object interfaces, easier to understand. Greatly benefiting the maintainability and scalability of WPSim's codebase.

The use of TypeScript in WPSim's frontend brought several benefits to the development process of WPSim, contributing to the efficiency, robustness, and maintainability of the codebase.

2.4.4 Node.js

Node.js is a JavaScript runtime environment built on V8 JavaScript engine, which is developed by Google. Designed for running scalable, non-static network applications, which may include data-intensive workloads ("Node.js", n.d.).

In the context of the WPSim stack, Node.js was utilized and chosen due to the following:

- **Efficient handling of asynchronous operations:** By operating on a single-threaded event loop, while using non-blocking I/O calls, Node.js can handle multiple concurrent connections in the event loop. This facilitates future expansion of WPSim, when experiencing an theoretically larger userbase.
- **JavaScript runtime environment:** Given that the frontend of WPSim was developed in TypeScript, using Node.js as a runtime environment, maintained consistency in the language used across the stack, speeding up the development process.
- **Ecosystem and package availability:** Node.js has a library of open-source packages available through NPM (Node Package Mananger). This ecosystem seemed and was proven valuable for integrating tools and libraries, further streamlining the development of WPSim.
- **Performance:** The V8 JavaScript engine Node.js provides, facilitates fast execution of JavaScript code. Which is a key requirement for a responsive user experience in WPSim.

Node.js were instrumental in the development and deployment of WPSim, providing a high-performance, scalable and efficient environment, while also offering valuable development tools.

2.5 Deployment

2.5.1 Docker

Docker is a virtualization based, container orchestration system, that simplifies the process of building, running, managing, and distributing applications ("Docker", n.d.). Docker images bundle an application along with required files, libraries, and dependencies into a single package. Deploying docker images to containers ensures that the application runs reliably from one computing environment to another, while also reducing the security risk.

In the context of WPSim, a container orchestration system was chosen due to the following:

- **Environment consistency:** Isolated and consistent environment for running WPSim reduces edge-case problems, while also ensuring that the applications works identically across different machines and environments. Streamlining development and deployment.
- **Isolation and security:** Containers are isolated by usage of virtualization, providing additional layers of security. Isolation minimizes the risk of being affected by code from the host system or other containers.

Docker was chosen due to the following:

- **Scalability and portability:** Docker has native support across multiple hardware and cloud platforms. Lowering the amount of development and work needed to deploy, replicate, stop, or move services. Increasing the flexibility and control when using WPSim.
- **Continuous Integration and Deployment (CI/CD):** Docker tools, including dockerfiles, streamlined the CI/CD process for WPSim by providing a consistent environment for build and test stages, eliminating development overhead when supporting multiple environments.

The use of a container orchestration greatly improved the robustness of WPSim, while Docker further facilitates scaling and management.

Chapter 3

Design and construction of software

This chapter describes how the web application was built in a local environment. The local version is located in the **development** branch of the **wpsim-prod** GitHub repository. (Bakkan & Selchow, 2023)

3.1 Database

This app contains three SQLite tables.

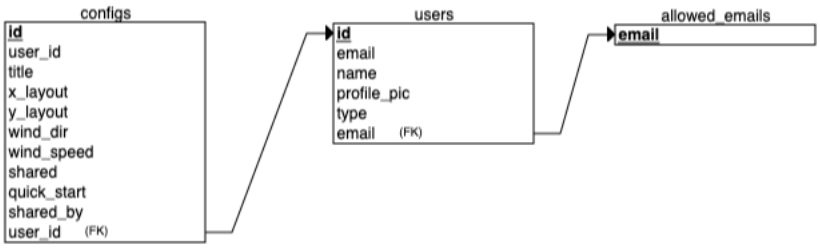


Figure 3.1: The SQLite tables showing their primary and foreign keys.

3.1.1 Allowed emails

This table was implemented to prevent everyone to gain access to the site when deploying on a web server. When attempting to log in it checks if the email fetched from the OAuth protocol is in this table. If the SQL statement gives no result the callback function is called with **false** to indicate that this user do not have access.

Code 3.1: Code that checks if the user is allowed access.

```
1 function (accessToken, refreshToken, profile, cb) {
2     let query = `SELECT * FROM allowed_emails WHERE email ...
          = "${profile.emails[0].value}"`;
3     db.get(query, (err, result) => {
4         if (err) return cb(err);
5         if (!result) return cb(null, false);
6     })
7 ...
```

3.1.2 Users

This table contains the actual users of the app. **user_id** consists of the Google or GitHub ID from the OAuth protocol plus either "google" or "github" added at the end to be sure there is no overlap between Google and GitHub IDs. How the information in this table is gathered is shown in section 3.2.

3.1.3 Configs

This table contains all the information about a configuration (config) so that the users can save their configs for later use:

- Which user it belongs to.
- The title of the config.
- The x and y coordinates for the turbines.

- The wind samples.
- If the config was shared to you and if so by whom.
- Quick start tells if the config is a quick start config.

3.2 Authentication

Implementing social login was learned by watching the video from Lama Dev, 2021.

Database access

The app needs to create a database object to access the database. This object is used to run SQL statements.

Code 3.2: Connecting to the database.

```
1 const sqlite3 = require('sqlite3').verbose();
2 const db = new sqlite3.Database('./database.db');
```

3.2.1 Express

First of all the Express server needs to be set up. The server listens to port 5050 and uses the routes contained in the file "auth.js".

Code 3.3: Express setup.

```
1 const authRoute = require("./routes/auth");
2 const app = express();
3 const frontdomain = "http://localhost:3000";
4
5 app.use(
6   cookieSession({ name: "session", keys: ["WebWindSim"]})
7 );
8
```

```
app.use(passport.initialize());
app.use(passport.session());

app.use(
  cors({
    origin: frontdomain,
    methods: "GET,POST,PUT,DELETE",
    credentials: true,
  })
);

app.use("/auth", authRoute);

app.listen("5050", () => {
  console.log("Server is running!");
});
```

3.2.2 Routes

Then the Google and GitHub routes are specified. The scope tells what information to gather from the OAuth protocol. A callback function is also needed.

Code 3.4: The Google route and callback.

```
router.get("/google", passport.authenticate("google", { ...
    scope: ["profile", "email"] }));

router.get(
  "/google/callback",
  passport.authenticate("google", {
    successRedirect: CLIENT_URL,
    failureRedirect: FAILED_URL,
  })
);
```

If the authentication is successful, the user is redirected to the **CLIENT_URL**.
If it is not successful, the user is redirected to the **FAILED_URL**.

Code 3.5: The callback URLs.

```
1 const CLIENT_URL = "http://localhost:3000/";
2 const FAILED_URL = "http://localhost:3000/failed";
```

3.2.3 App registration

The web application needs to be registered at both Google APIs & Services and GitHub Developer settings to obtain **CLIENT_ID** and **CLIENT_SECRET**. These are necessary for the app to be allowed to gather the data from OAuth.

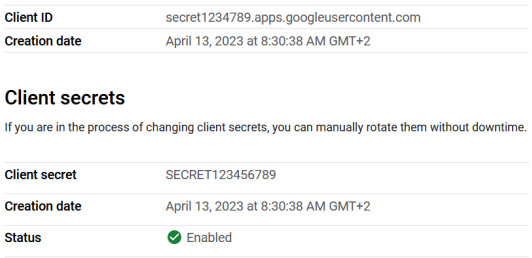


Figure 3.2: The CLIENT_ID and CLIENT_SECRET from Google.

Then specify the app's URL and callback URL.

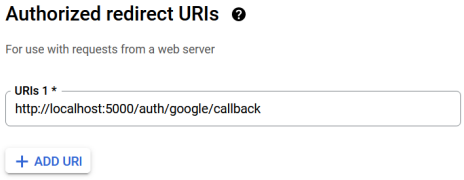


Figure 3.3: The app's URL and callback URL for Google.

3.2.4 Strategies

The strategies define what to do with the gathered data. The first part contains the **CLIENT_ID** and **CLIENT_SECRET**.

Code 3.6: The first part of the Google strategy.

```
passport.use(
  new GoogleStrategy(
    {
      clientID: GOOGLE_CLIENT_ID,
      clientSecret: GOOGLE_CLIENT_SECRET,
      callbackURL: "/auth/google/callback",
    },
    ...
```

Then define the function to be executed when the data is gathered. First the function checks if the user has access as shown in section 3.1.1 by looking up the email in the **allowed_emails** table. If the email is allowed and the user already exists in the database it returns the results. Otherwise it inserts a new user into the database and returns the user object. **cb** is the callback function.

Code 3.7: Code that fetches a user or creates one.

```
...
query = `SELECT * FROM users WHERE id = ...
    "${profile.id+"google"}"`;
    db.get(query, (err, result) => {
    if (err) return cb(err);
    if (result) return cb(null, result);
    const insertQuery = `INSERT INTO users (id, email, ...
        name, profile_pic, type) VALUES ...
        ("${profile.id+"google"}", ...
        "${profile.emails[0].value}", ...
        "${profile.displayName}", ...
        "${profile.photos[0].value}", "Basic")`;
    db.run(insertQuery, (insertErr) => {
        if (insertErr) return cb(insertErr);
        return cb(null, {
            id: profile.id+"google",
            email: profile.emails[0].value,
            name: profile.displayName,
            profile_pic: profile.photos[0].value,
```

```
14              type: "Basic"
15          });
16      });
17  });
18  ...
```

3.2.5 Serialization and deserialization

After the chosen strategy is completed, the serialization of the user starts. This means taking the user object and converting it to a form to be stored in the session to identify the user later. The function returns the user's ID.

Code 3.8: Serialization of the user object.

```
1  passport.serializeUser((user, cb) => {
2    cb(null, user.id);
3  });
```

The deserialization happens whenever the user sends a request to the server. It takes the user ID and returns the user object from the database and attaches it to the **req.user** property.

Code 3.9: Deserialization of the user ID.

```
1  passport.deserializeUser((id, cb) => {
2    let query = `SELECT * FROM users WHERE id = "${id}"`;
3    db.get(query, (err, result) => {
4      if (err) return cb(err);
5      if (result) return cb(null, result);
6      return cb(null, false);
7    });
8  });
```

3.2.6 Verifying logged in state

Now the frontend can send a GET request to the Express server to verify if the user is logged in. The endpoint checks if the request contains a user as

shown in previous subsection. If it does, the user is sent back.

Code 3.10: Express checks if request contains user.

```
router.get("/login/success", (req, res) => {
  if (req.user) {
    res.status(200).json({
      success: true,
      message: "successful",
      user: req.user,
      //   cookies: req.cookies
    });
  } else {
    res.status(200).json({
      success: false,
    })
  }
});
```

If the response object has the field **success** as **true**, the function **getUser** sets the user object with the **setUser** hook and sets the ID and type in the **localStorage** for later use.

Code 3.11: GET request from the frontend to verify if user is logged in.

```
    const getUser = async () => {
        fetch("http://localhost:5000/auth/login/success", {
            method: "GET",
            ...
        },
        })
        .then((response) => {
          if (response.status === 200) return response.json();
          throw new Error("authentication has been failed!");
        })
        .then((resObject) => {
          if (resObject.success) {
            setUser(resObject.user);
            localStorage.setItem("id", resObject.user.id);
            localStorage.setItem("type", resObject.user.type);
          }
          ...
```

3.3 Application programming interface

This section contains explanations of the API.

FastAPI

First of all FastAPI has to be initialized.

Code 3.12: How to initialize FastAPI.

```
app = FastAPI()

frontdomain = "http://localhost:3000"
authdomain = "http://localhost:5000"
simdomain = "http://localhost:8000"
database = "../auth/database.db"

origins = [
    frontdomain,
    authdomain,
    simdomain
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Database access

The endpoints that use the database needs to first connect to it and then commit and close the connection when done.

Code 3.13: How to connect to the database and close it.

```
con = sqlite3.connect(database)
cur = con.cursor()
cur.execute(SQL STATEMENT)
con.commit()
con.close()
```

FLORIS and JSON

The endpoints that use FLORIS starts with the code shown below. FLORIS is initalized and the data from the request is loaded from JSON (JavaScript Object Notation). The data consists of the x and y coordinates for the turbines, wind directions and wind speeds. This will be referred to as a configuration or config. Other endpoints may load data containing for example user ID. The data sent back is first converted to JSON.

Code 3.14: How to initialize FLORIS and load JSON data. Data sent back is converted to JSON.

```
fi = FlorisInterface("./floris/inputs/gch.yaml")

body = await req.json()
data = body["body"]
x = json.loads(data["layout_x"])
y = json.loads(data["layout_y"])
wind_directions = json.loads(data["wind_directions"])
wind_speeds = json.loads(data["wind_speeds"])

fi.reinitialize(
    layout_x=x,
    layout_y=y,
    )
... #Calculations
turbine_powers = json.dumps(turbine_powers)
return {"turbine_powers": turbine_powers}
```

3.3.1 Startup

First up is the startup event which creates the SQLite configs table if it does not exist and inserts the quick start configurations from a file. These are predefined configs.

Code 3.15: Code snippet that shows the SQL statements.

```
1 @app.on_event("startup")
2     ...
3     cur.execute("CREATE TABLE IF NOT EXISTS configs(id ...
          integer primary key autoincrement, user_id ...
          VARCHAR, title VARCHAR, x_layout VARCHAR, y_layout ...
          VARCHAR, wind_dir VARCHAR, wind_speed VARCHAR, ...
          shared VARCHAR, quick_start VARCHAR, shared_by ...
          VARCHAR, foreign key(user_id) references users(id))")
4     ...
5     for config in configs:
6     ...
7         cur.execute("INSERT INTO configs (title, x_layout, ...
              y_layout, wind_dir, wind_speed, quick_start) ...
              VALUES(?, ?, ?, ?, ?, ?)",
8                           (config[0], config[1], config[2], ...
                                config[3], config[4], "true"))
9     ...
```

3.3.2 Turbine powers

This endpoint calculates the power production from the turbine positions and the wind samples. After initalizing FLORIS and loading the JSON data the calculations begin.

Code 3.16: How to calculate power production.

```
@app.post("/floris/calculate/powers")
async def powers(req: Request):
    ...
    turbine_powers = []
    aep = []

    for i in range(len(wind_directions)):
        dir = [wind_directions[i]]
        speed = [wind_speeds[i]]
        fi.reinitialize(
            wind_directions=dir,
            wind_speeds=speed
        )

        fi.calculate_wake()
        powers = fi.get_turbine_powers() / 1E3
        production = fi.get_farm_power().sum()  / 1E9 * ...
            365 * 24
        aep.append(round(production, 1))

        turbines = []

        for k in range(powers.shape[2]):
            turbines.append(round(powers[0, 0, k], 1))
        turbine_powers.append(turbines)
```

The FLORIS object needs to be reinitialized for each wind sample because FLORIS calculates for each possible combination. A wind sample consists of one wind direction and one wind speed. FLORIS calculates 100 conditions when given 10 directions and 10 speeds, when the desired number of conditions is 10. So for each wind sample the turbine powers and the annual energy production of the turbines is calculated and added to separate arrays. The endpoint returns:

- The turbine powers in kW for each wind sample.
- The annual energy production for each wind sample.
- The average annual energy production.
- The maximum annual energy production.
- The wind samples which gives the maximum annual energy production.

Code 3.17: Code snippet that shows the convertion to JSON and returning values.

```
...
averageAep = sum(aep) / len(aep)
averageAep = json.dumps(round(averageAep, 1))
turbine_powers = json.dumps(turbine_powers)
maxAep = max(aep)
maxConditions = [i+1 for i, x in enumerate(aep) if x == ...
    maxAep]
maxAep = json.dumps(maxAep)
maxConditions = json.dumps(maxConditions)
aep = json.dumps(aep)

return {"turbine_powers": turbine_powers, "aep": aep, ...
    "average": averageAep, "maxAep": maxAep, ...
    "maxConditions": maxConditions}
```

3.3.3 Turbine wind speeds

This endpoint is quite similar to previous subsection, but instead of calculating the powers, it calculates the wind speeds at every turbine for each wind sample.

Code 3.18: How to calculate wind speeds.

```
@app.post("/floris/calculate/speeds")
async def speeds(req: Request):
    ...
    avg_vel = fi.get_turbine_average_velocities()
    ...
    turbine_speeds = json.dumps(turbine_speeds)

    return {"turbine_speeds": turbine_speeds}
```

3.3.4 Wake plots

This is the endpoint which produces plots that show the wake effect for each wind sample. After the initialization of FLORIS and loading of data the

plots are produced. The FLORIS object calculates the horizontal plane and **visualize_cut_plane** from **floris.tools** visualizes it.

Code 3.19: How to produce the wake plots.

```
@app.post("/floris/calculate/plot/wake")
async def wake(req: Request):
    ...
    cMAP = ...
        col.LinearSegmentedColormap.from_list("cCMAP",[(0.0, ...
        "#96007C"), (0.50, "#00488F"), (0.86, "#BEC0C8"), ...
        (1, "#ffffff")], N=1024)
    plots = []

    for i in range(len(wind_directions)):
        dir = [wind_directions[i]]
        speed = [wind_speeds[i]]
        fi.reinitialize(
            wind_directions=dir,
            wind_speeds=speed
        )
        fi.calculate_wake()

        fig, axarr = plt.subplots(1, 1, figsize=(10,8))
        horizontal_plane = ...
            fi.calculate_horizontal_plane(x_bounds=(0,2200), ...
            y_bounds=(-500,1000),wd=[wind_directions[i]], ...
            height=90.0)
        visualize_cut_plane(horizontal_plane, ...
            ax=axarr,cmap=cMAP, title="Turbines aligned ...
            with wind, coming from the left"+ "\n"+"Wind ...
            sample "+ str(i+1) + ": " ...
            +str(wind_directions[i]) + "\N{DEGREE SIGN} ...
            ("+str(wind_speeds[i])+ " m/s)")
```

After each plot is produced it needs to be encoded using base64 to be able to be converted to JSON.

Code 3.20: How to encode the plots using base64.

```
    ...
    buffer = BytesIO()
    fig.savefig(buffer, format='png', bbox_inches='tight')
    buffer.seek(0)
    image_base64 = ...
        base64.b64encode(buffer.getvalue()).decode('utf-8')
```

```
    plots.append(image_base64)

plots_json = json.dumps(plots)
plt.close()

return {"plots": plots_json}
```

3.3.5 Power plot

This endpoint produces a plot containing three subplots. Each subplot has wind sample number as x values and the y values are wind direction, wind speed and turbine power. First it determines how many wind samples and turbines there are and then calculates the turbine powers.

Code 3.21: Determining number of wind samples and turbines, and calculation of turbine powers.

```
@app.post("/floris/calculate/plot/power")
async def power(req: Request):
    ...
    condition = range(1, len(wind_directions)+1)
    num_turbines = len(fi.layout_x)
    turbine_powers = fi.get_turbine_powers() / 1000.
    ...
```

Then the three subplots are produced.

Code 3.22: How to produce the three subplots.

```
...
fig, axarr = plt.subplots(3, 1, sharex=True, figsize=(11,9))

ax = axarr[0]
ax.plot(condition, wind_directions, 'o-')
ax.set_ylabel('Wind Direction (Deg)')
ax.grid(True)

ax = axarr[1]
ax.plot(condition, wind_speeds, 'o-')
ax.set_ylabel('Wind Speed (m/s)')
ax.grid(True)

```

```
14 ax = axarr[2]
15 for t in range(num_turbines):
16     ax.plot(condition, turbine_powers[:, 0, t], 'o-', ...
           label='Turbine %d' % (t+1))
17
18 ax.legend(bbox_to_anchor=(1.0, 1.0))
19 ax.set_ylabel('Turbine Power (kW)')
20 ax.set_xlabel('Wind Sample')
21 ax.grid(True)
22 ...
```

Lastly the plots are encoded using base64 and sent as JSON as shown in previous subsection.

3.3.6 Improve layout

This endpoint takes in the turbine positions and calculates new positions to improve the energy production. FLORIS contains a class which calculates this using SciPy (Scientific Python) optimization. SciPy's optimize module offers various functions that can be used to minimize or maximize objective functions, while also taking into account possible constraints. ("Optimization and root finding", n.d.) Using a lot of wind samples causes long calculation times. Random plausible wind samples are therefore created to give a good approximation.

Code 3.23: How to create random plausible wind samples.

```
1 ...
2 wind_directions = np.arange(0, 360.0, 5.0)
3 np.random.seed(1)
4 wind_speeds = 8.0 + np.random.randn(1) * 0.5
5 ...
```

The boundaries are the corners in the square where the turbines can be placed. These are added to tell the optimizer which area to work in. After reinitializing FLORIS with the turbine positions and random wind samples the optimization can begin.

Code 3.24: Define the boundaries and create the optimization object.

```
1 ...
2 boundaries = [(25.0, 25.0), (25.0, 575.0), (975.0, 575.0), ...
      (975.0, 25.0)]
3 layout_opt = LayoutOptimizationScipy(fi, boundaries, ...
      freq=freq)
4 sol = layout_opt.optimize()
5 ...
```

Then calculate wake and annual energy production for both the old positions and the optimized positions to calculate the gain in energy produced.

Code 3.25: Calculate the gain in energy produced.

```
1 ...
2 fi.calculate_wake()
3 base_aep = fi.get_farm_AEP(freq=freq) / 1e6
4 fi.reinitialize(layout=sol)
5 fi.calculate_wake()
6 opt_aep = fi.get_farm_AEP(freq=freq) / 1e6
7 percent_gain = 100 * (opt_aep - base_aep) / base_aep
8 ...
```

Lastly get the improved positions from the FLORIS object.

Code 3.26: Get the new turbine positions.

```
1 layout = fi.get_turbine_layout()
2 layout_x = list(layout[0])
3 layout_y = list(layout[1])
4 ...
5 return {"layout_x": layout_x, "layout_y": layout_y, ...
      "percent_gain": percent_gain}
```

3.3.7 Get config

This endpoints fetches a the whole configuration given it's ID. After connecting to the database and loading JSON data, the code below is executed.

Code 3.27: How to fetch a configuration from the database.

```
1 @app.post("/configs/get")
2 async def get(req: Request):
3     ...
4     config = cur.execute("SELECT * FROM configs WHERE id = ...
          (?)", (id,)).fetchone()
5     ...
6     return {"config" : config}
```

3.3.8 Load configs

This endpoint can fetch all the configuration ID's and title/sharer a given user has access to. It always fetches the user's saved configs. If **quick_start** and **shared** are set to **true** it also fetches the quick start configs and the configs shared with the user. The previous subsection describes how to fetch the whole config.

Code 3.28: How to fetch the configurations a user has access to from the database.

```
1  @app.post("/configs/load")
2  async def load(req: Request):
3      ...
4      saved_configs = cur.execute("SELECT id, title FROM ...
           configs WHERE user_id = (?) AND shared = (?)", ...
           (user_id, "false")).fetchall()
5      if quick_start:
6          quick_start = cur.execute("SELECT id, title FROM ...
               configs WHERE quick_start = (?)", ...
               ("true",)).fetchall()
7      if shared:
8         shared = cur.execute("SELECT id, shared_by FROM ...
              configs WHERE user_id = (?) AND shared = (?)", ...
              (user_id, "true")).fetchall()
9      ...
10     return {"saved_configs" : saved_configs, "quick_start" ...
           : quick_start, "shared" : shared}
```

3.3.9 Save config

This endpoint inserts a configuration to the database given a configuration and user ID. First it checks if the title already exists to prevent confusion for the user. If it does not exist, it inserts the config and sends back the config's ID.

Code 3.29: How to insert a configuration into the database.

```
1  @app.post("/configs/save")
2  async def save(req: Request):
3      ...
4      title_exist = cur.execute("SELECT * FROM configs where ...
           title = ? and user_id = ? and shared != 'true'", ...
           (data["title"], data["user_id"])).fetchone()
5      if title_exist:
6          return {"title_exists": "true"}
7
8      cur.execute("INSERT INTO configs (user_id, title, ...
           x_layout, y_layout, wind_dir, wind_speed, shared) ...
           VALUES(?, ?, ?, ?, ?, ?, ?)", (data["user_id"], ...
           data["title"], data["layout_x"], data["layout_y"], ...
           data["wind_directions"], data["wind_speeds"], ...
           "false"))
9      id = cur.execute("SELECT last_insert_rowid() FROM ...
           configs;").fetchone()
10     ...
11     return {"id": id[0]}
```

3.3.10 Update config

This endpoint does the same as last subsection except it updates the current configuration instead of inserting it.

Code 3.30: How to update a configuration in the database.

```
1  @app.put("/configs/update")
2  async def update(req: Request):
3      ...
4      title_exist = cur.execute("SELECT * FROM configs where ...
           title = ? and user_id = ? and id != ? and shared ...
```

```
            != 'true'", (data["title"], data["user_id"], ...
            data["id"])).fetchone()
    if title_exist:
        return {"title_exists": "true"}

    cur.execute("UPDATE configs SET title = ?, x_layout = ...
        ?, y_layout = ?, wind_dir = ?, wind_speed = ? ...
        WHERE id = ?", (data["title"], data["layout_x"], ...
        data["layout_y"], data["wind_directions"], ...
        data["wind_speeds"], data["id"]))
    ...
     return {"updated": "true"}
```

3.3.11 Delete config

This endpoints deletes a configuration given it's ID.

Code 3.31: How to delete a configuration in the database.

```
@app.post("/configs/delete")
async def delete(req: Request):
    ...
    cur.execute("DELETE from configs where id = ?", (id,))
    ...
    return {"deleted": "true"}
```

3.3.12 Share config

This endpoint inserts a configuration into the database with the attribute **shared** set to **true** given a configuration, ID of sharer and email of the user shared to. First the name of the sharer and the ID of the user shared to are fetched from the database. If there are no user ID's with the given email it returns with a message telling the sharing failed.

Code 3.32: How to fetch name of the sharer and ID of the user shared to.

```
1 @app.post("/configs/share")
2 async def share(req: Request):
3     ...
4     name = cur.execute("SELECT name FROM users WHERE id = ...
          ?", (data["user_id"],)).fetchone()
5     user_id = cur.execute("SELECT id FROM users WHERE ...
          email = ?", (data["email"],)).fetchall()
6     if len(user_id) == 0:
7         return {"shared": "false"}
```

There might be more than one ID connected to an email because a user might have the same email for both Google and GitHub. So the config is inserted into the database for each ID found.

Code 3.33: How to insert the shared configuration into the database.

```
1     ...
2     for id in user_id:
3         cur.execute("INSERT INTO configs (user_id, title, ...
              x_layout, y_layout, wind_dir, wind_speed, ...
              shared, shared_by) VALUES(?, ?, ?, ?, ?, ?, ?, ...
              ?)", (id[0], data["title"], data["layout_x"], ...
              data["layout_y"], data["wind_directions"], ...
              data["wind_speeds"], "true", name[0]))
4     ...
5     return {"shared": "true"}
```

3.3.13 Update user

This endpoint updates a user's account type given a user ID and a new account type. The types are **Basic** and **Pro**. A Pro user has access to more functionality than a Basic user.

Code 3.34: How to update the account type of a user.

```
@app.put("/users/update")
async def update_type(req: Request):
    ...
    cur.execute("UPDATE users SET type = ? WHERE id = ?", ...
        (new_type, user_id))
    ...
    if new_type == "Pro":
        return {"updated_to": "Pro"}
    if new_type == "Basic":
        return {"updated_to": "Basic"}
```

3.4 Pages

Routes and navigation

The package React Router DOM was used for routing. It takes a React component and connects it with a specified path.

Code 3.35: React Router DOM routes for home and login pages.

```
const App = () => {
    ...
    return (
    <BrowserRouter>
      <div>
      ...
        <Routes>
          <Route
            path="/"
            element= { <Home /> }/>
          <Route
            path="/login"
            element={ <Login /> }
          />
        ...
        </Routes>
      </div>
    </BrowserRouter>
  );
};
```

If the user tries to access pages that requires to be logged in, they will be redirected to the login page by **useNavigate** from React Router DOM.

Code 3.36: Redirected to login page when not logged in.

```
1    const navigate = useNavigate();
2    useEffect(() => {
3      if (!localStorage.getItem("id")) {
4         navigate("/login")
5      }
6      }, []);
```

Navigation bar

The navigation bar is displayed on every page. It contains links to the home, start and login pages for easy navigation. If the user is logged in, the login link is replaced by logout, and name and profile picture is displayed and linked to the profile page.

Figure 3.4: The navigation bar before a user is logged in. Graphics: Authors (Apache 2.0).

The navigation bar's props is defined by an interface where both props are optional. If the props are **undefined** the component returns the bar shown in figure 3.5. If the user is logged in the props would be defined and the user is shown in the bar.

Code 3.37: The interface for the navigation bar component.

```
1  interface Props {
2    email?: string;
3    name?: string;
4    profile_pic?: string;
5  }
```

Fetching data

Fetching data from the backend is done by Axios. An example is shown below. Define the parameters in the body and use the response data as desired.

Code 3.38: Loading configurations from the backend with Axios.

```
1  const REACT_APP_simdomain = "http://localhost:8000"
2  const fetch_configs = async () => {
3      try {
4          const response = await ...
               axios.post((REACT_APP_simdomain+'/configs/load'), ...
               { body : {
5              "user_id" : localStorage.getItem("id"),
6              "quick_start" : JSON.stringify(true),
7              "shared" : JSON.stringify(true),
8          }});
9          setQuickStart(response.data["quick_start"]);
10         setSavedConfigs(response.data["saved_configs"]);
11         setSharedConfigs(response.data["shared"]);
12     } catch (error) {
13         console.error(error);
14     }
15 };
```

3.4.1 Home (/)

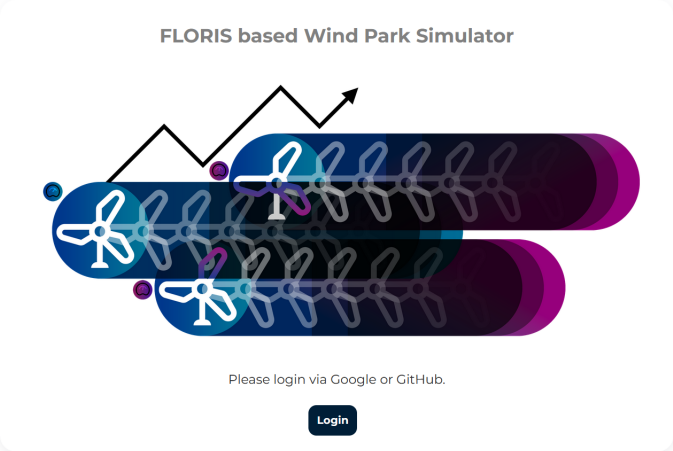


Figure 3.5: The HomeCard component on the home page. Graphics: Authors (Apache 2.0).

This page consists of a React component welcoming the user to the application. The component checks the **localStorage** to see if the user is logged in. If so, the login button which links to the login page changes to a button which links to the start page. The text also changes.

Code 3.39: The Home component using the HomeCard component.

```
1  const Home: React.FC = () => {
2      return (
3          <div>
4          <div className="home">
5              {localStorage.getItem("id") ? (
6                  <HomeCard title={"Welcome to WPSim - Wind ...
                       Park Simulator"} img={pic} desc={"You ...
                       are logged in."} linkTo={"/start"} ...
                       buttonText={"Go to Simulations"}/>
7              ) :
8              <HomeCard title={"Welcome to WPSim - Wind Park ...
                   Simulator"} img={pic} desc={"Please login ...
                   via Google or GitHub to gain access:"} ...
                   linkTo={"/login"} buttonText={"Login"}/>
9              }
10         </div>
11         </div>)}
```

3.4.2 Login (/login)

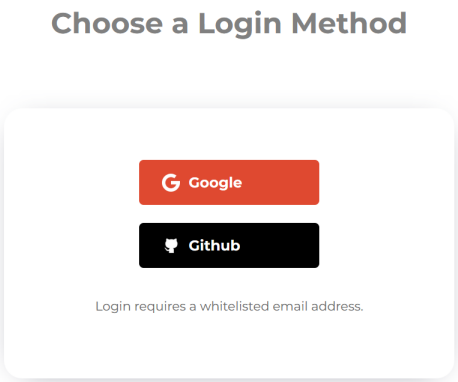


Figure 3.6: The Login component on the login page.

This page consists of a component which the user can log in via Google or GitHub. The two HTML elements have **onClick** functions attached to them. The functions starts the authentication process described earlier.

Code 3.40: The two functions starting the authentication process.

```
1 const REACT_APP_authdomain = "http://localhost:5000"
2
3 const google = () => {
4     window.open((REACT_APP_authdomain+"/auth/google"), ...
          "_self");
5   };
6 const github = () => {
7     window.open((REACT_APP_authdomain+"/auth/github"), ...
          "_self");
8   };
```

3.4.3 Profile (/profile)

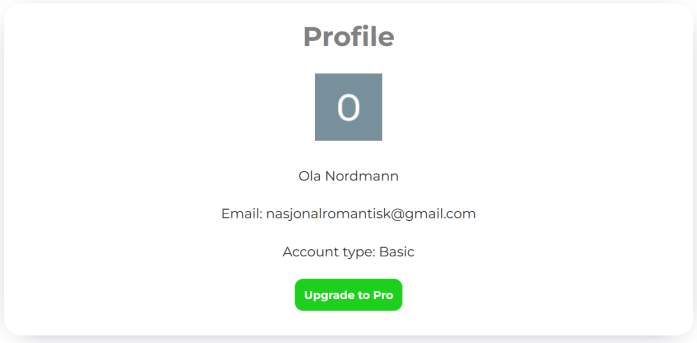


Figure 3.7: The Profile component on the profile page.

This page shows the information about your profile which consists of name, email, profile picture and account type. The button updates the type the user's account by sending a request to /**users**/**update** at the backend.

3.4.4 Start (/start)

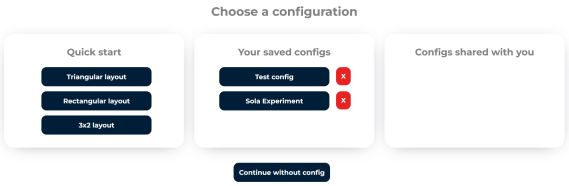


Figure 3.8: Three StartCard components on the start page.

This page consists of three **StartCard** components which receives different props. The configs listed in each component are fetched with Axios from /**configs**/**load** using the **useEffect** hook. The "Your saved configs" and "Configs shared with you" components also has a delete button and receives this function as a prop. The function sends a request to /**configs**/**delete** with the corresponding config ID.

Code 3.41: The three StartCard components.

```
1 <StartCard title={"Quick start"} configs={quickStart}/>
2 <StartCard title={"Your saved configs"} ...
      configs={savedConfigs} deleteButton={true} ...
      onClick={deleteAndFetch}/>
3 <StartCard title={"Configs shared with you"} ...
      configs={sharedConfigs} deleteButton={true} ...
      onClick={deleteAndFetch}/>
```

Clicking on one of the configs navigates to the simulation page with the clicked config. The red buttons deletes the config and then fetches the configs again to show the updated list. The bottom button navigates to the simulation page with an empty config.

3.4.5 Simulation (/simulation)

This is the page where the user performs simulations.

Alerts

The user is alerted if they miss some input or the input is invalid.

Code 3.42: Example where the user is alerted if one of the values are empty when attempting to fetch from the backend.

```
1 if (layout_x.length === 0 || layout_y.length === 0 || ...
      wind_directions.length === 0 || wind_speeds.length === ...
      0) {
2       alert("Please add turbines and wind samples.");
3       return
4     }
```

Turbine layout

The canvas HTML element is utilized to draw graphics, and was used to determine the turbine positions.

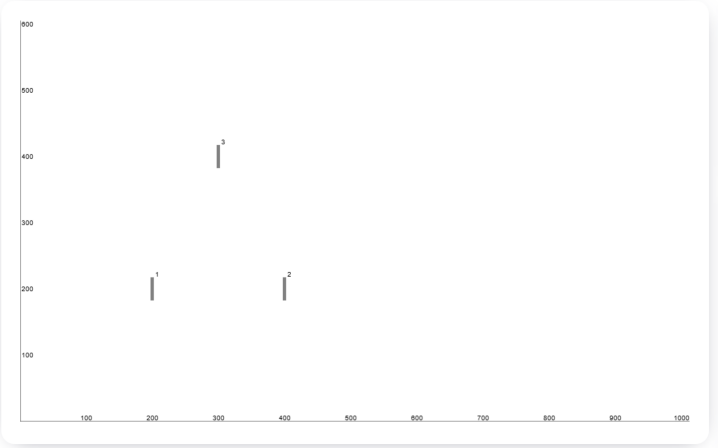


Figure 3.9: The canvas element containing the turbine positions.

Code 3.43: The canvas HTML element.

```
<canvas className='canvas'
    ref={canvasRef}
    width={1012}
    height={606}
    onClick={handleClickCanvas}
/>
```

Clicking on the canvas triggers a function which adds the x and y coordinates to separate arrays, and both as an object to the **clicks** array. The function also checks if the click is at least 150 meters apart from other turbines before adding.

Code 3.44: The **handleClickCanvas** function.

```
const handleClickCanvas = (event: ...
    React.MouseEvent<HTMLCanvasElement>) => {
    const canvas = canvasRef.current;
    if (!canvas) return;

    const rect = canvas.getBoundingClientRect();
    const x = Math.round(event.clientX - rect.left);
    const y = Math.round(canvas.height - (event.clientY - ...
        rect.top));

    for (var k = 0; k < clicks.length; k++) {
```

```
10          if ((Math.abs(x-clicks[k]["x"])) < 150 && ...
                (Math.abs(y-clicks[k]["y"]) < 150)) {
11              alert("Turbines need to be at least 150 meters ...
                    apart.");
12              return
13        }
14      }
15
16      setClicks([...clicks, { x, y }]);
17      setLayoutX([...layout_x, x]);
18      setLayoutY([...layout_y, y]);
19  };
```

The **useEffect** hook draws the canvas axes and their values before drawing the clicks as turbines. This gets triggered again when the value of **clicks** changes, which is when clicks are added or removed.

Code 3.45: Drawing of axes and turbines.

```
1  useEffect(() => {
2      ...
3
4      // Draw X axis
5      ctx.beginPath();
6      ctx.moveTo(0, canvas.height);
7      ctx.lineTo(canvas.width, canvas.height);
8      ctx.stroke();
9
10     // Draw X axis values
11     ctx.textBaseline = "bottom";
12     ctx.textAlign = "center";
13     for (let x = 100; x < canvas.width; x += 100) {
14       ctx.fillText(x.toString(), x, canvas.height);
15     }
16
17     ...
18
19     // Draw clicks
20     clicks.forEach(({ x, y }, index) => {
21       ctx.fillStyle = "gray";
22       ctx.fillRect(x - 3, canvas.height - y - 18, 5, 35);
23       ctx.fillStyle = "#000000";
24       ctx.fillText(`${index + 1}`, x + 10, canvas.height - ...
             y - 22);
25     });
26
```

```
27      setShowImproved(false);
28  }, [clicks, reRender]);
```

The numbering and positions of the turbines are listed in the **LayoutCard** component. Clicking the red button removes the corresponding turbine. The function receives the index of the turbine and removes the values from the arrays.

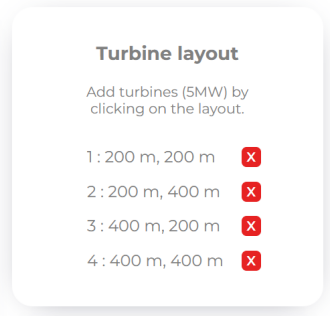


Figure 3.10: The LayoutCard component listing the turbine positions.

Code 3.46: Removing a turbine.

```
1  const removeTurbine = (index:number) => {
2      removeClick(index);
3      removeX(index);
4      removeY(index);
5  }
```

Wind samples

The **WindCard** component is similar to the **LayoutCard** component. It lists the wind samples consisting of a direction and a speed, and the users are able to remove them.

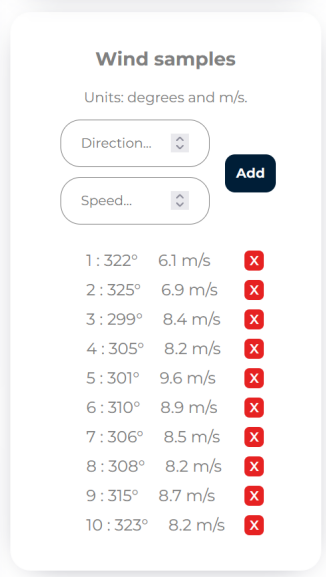


Figure 3.11: The WindCard component listing the wind samples.

The two input HTML elements only accepts numbers by defining the type.

Code 3.47: The WindCard input HTML elements.

```
1 ...
2 <input className='inputBox' type="number" ...
      placeholder="Insert direction..." ...
      onChange={props.onChangeDir} />
3 ...
4 <input className='inputBox' type="number" step={0.1} ...
      placeholder="Insert speed..." ...
      onChange={props.onChangeSpeed} />
```

Clicking add button checks if both fields are filled and if both fields are within legal range before adding them.

Code 3.48: The addWind function.

```
1 const addWind = () => {
2     if (wind_directions_input.length===0 || ...
          wind_speeds_input.length===0) {
3         alert("Please insert direction and speed.");
```

```
4           return
5       }
6       if ( 25 < wind_speeds_input || wind_speeds_input < 0 ...
            || wind_directions_input < 0 || ...
            wind_directions_input > 360) {
7           alert("Please insert speed from 0 to 25 and ...
                direction from 0 to 360.");
8           return
9       }
10      addDirection();
11      addSpeed();
12  };
```

Save, update and share

This component contains three buttons which saves a new config, updates the current one or shares the current one with another user by sending requests with Axios to the backend. The user can also download the config as CSV.

Figure 3.12: Component which saves, updates shares or downloads a config.

Code 3.49: The three buttons with corresponding onClick functions.

```
1  ...
2  <button className="cardButton" ...
       onClick={save}>{saveButton}</button>
3  ...
4  <button className="updateButton" ...
       onClick={update}>{updateButton}</button>
5  ...
6  <button className="shareButton" ...
       onClick={share}>{shareButton}</button>
```

Calculations and results

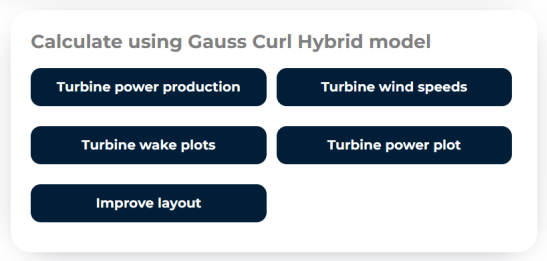


Figure 3.13: Component with buttons to request calculations.

This component consists of five buttons which triggers different functions where each function sends requests to the backend with Axios. Importing wind samples from Seklima is possible using Papa Parse. **Turbine power production** and **Turbine wind speeds** creates graphs using Recharts. The data used in the charts needs to be in a specific object form. After setting the response data from the backend, the objects are generated. The code beneath generates the data for the **Turbine powers** graph. The code loops through the array of turbine powers and generates objects. Each object consists of the turbine powers for a specific wind sample.

Code 3.50: The loop which generates graph data.

```
1  for (var k=0; k < powers.length; k++) {
2      myObject = {};
3      myObject["name"] = `Wind sample ${k+1}: ...
           ${wind_directions[k]}\xB0 ${wind_speeds[k]}m/s`;
4      samples.push(`${k+1}: ${wind_directions[k]}\xB0 ...
           ${wind_speeds[k]}m/s`);
5      for (var i=0; i < powers[k].length; i++) {
6          myObject[`Turb${i+1}`] = powers[k][i];
7          if (k===0) {
8          myTurbines.push(`Turb${i+1}`);
9          }
10     }
11     myData.push(myObject);
12     }
```

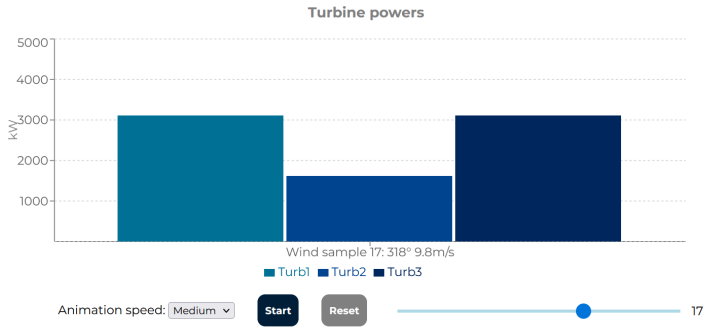


Figure 3.14: Turbine powers graph.

The graph shows the turbine powers for each wind sample. The user can drag a slider or press **Start** to begin the simulation to switch between the samples. The graph component receives the **activeIndex** variable which is changed by the slider. This variable determines which sample to show in the graph.

Code 3.51: The Slider component from React Slider.

```
<Slider
    value={activeIndex}
    min={0}
    max={sliderLength}
    onChange={handleSliderChange}
    className="slider"
    thumbClassName="thumb"
    trackClassName="track"
    marks
/>
```

Code 3.52: One of the components from Recharts showing which data to display using **activeIndex**.

```
<BarChart width={900} height={325} ...
    data={[props.data[props.activeIndex]]}>
```

The **Power production** graph also shows the previous data when changing the **activeIndex** to obtain a simulation effect.

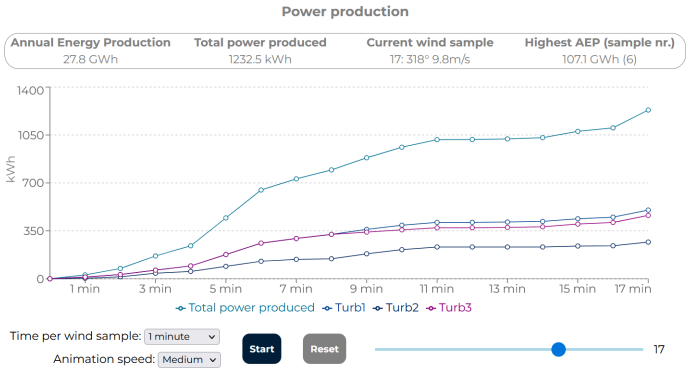


Figure 3.15: Power production graph.

Time per wind sample determines how long each wind sample produces energy before moving to the next. Changing it triggers a function which calculates the correct power production with the new time per wind sample. **Simulation speed** determines how long before the active index changes when pressing **Start**. The **Reset** button sets the **activeIndex** to 0.

The calculation buttons **Turbine wake plots** and **Turbine power plot** requests these plots from the backend where they are generated. The turbines in each wake plot are aligned with the wind coming from the left to achieve some consistency in the different plots. Otherwise each plot would have drastic differences when it comes to the length of the axes which would be perceived as chaotic.

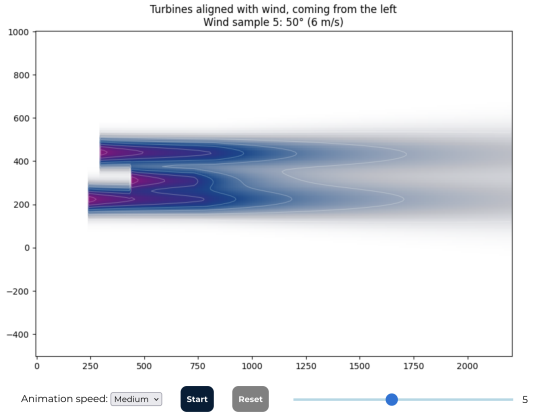


Figure 3.16: Wake plot.

The **Improve layout** button sends a request to the backend and displays improved positions and the increased energy production.

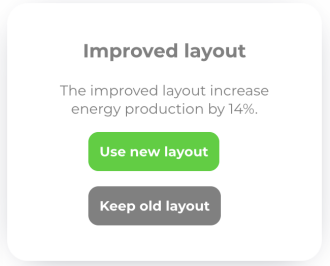


Figure 3.17: The ImprovementCard component.

The user can choose to use the improved layout or keep the old. Both layouts are displayed in the canvas.

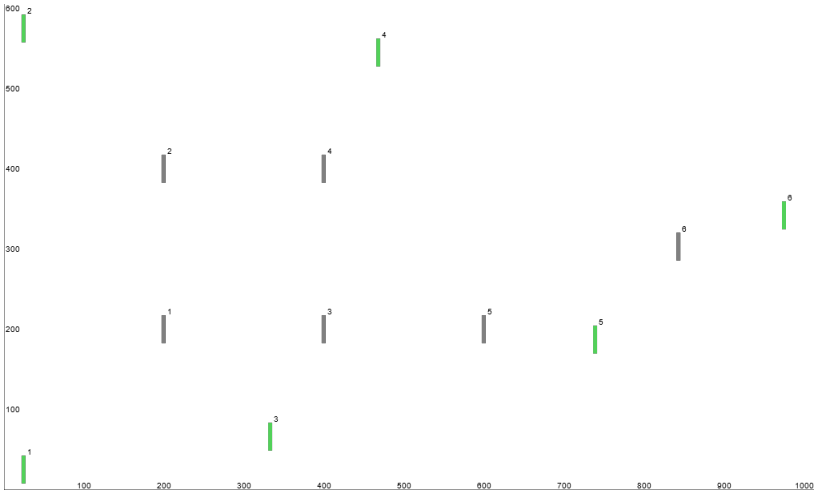


Figure 3.18: The canvas with improved layout in green.

Downloading results

The results from the calculations can be downloaded.

Power production, turbine powers and turbine wind speeds can be downloaded as separate CSV files using React CSV.

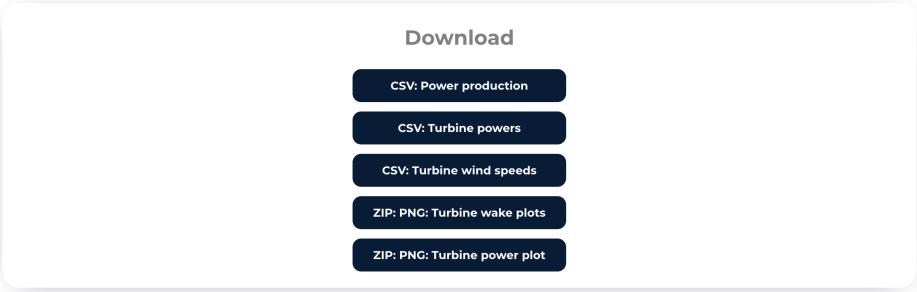


Figure 3.19: Download request buttons.

Code 3.53: One of the components from React CSV.

```
<CSVLink filename='wpsim_production.csv' ...
    onClick={generateCsvProduction} data={csvProduction} ...
    separator={";"}><button ...
    className='downloadButton'>Power production ...
    CSV</button></CSVLink>
```

Clicking the buttons triggers the functions which generates the CSV data, and is downloaded as a CSV file afterwards.

Code 3.54: The function which generates CSV data for the power production.

```
const generateCsvProduction = () => {
    var csv = [
        ["Time", "Direction", "Speed"],
    ];
    csv = [csv[0].concat(totalPowerChartTurbines)];
    for (var i = 1; i < totalPowerChartData.length; i++) {
        var data = [(timeRes * i).toString(), ...
            wind_directions[i - 1].toString(), ...
            wind_speeds[i - 1].toString()];
        for (var k = 0; k < ...
            totalPowerChartTurbines.length; k++) {
            data.push(totalPowerChartData[i][csv[0][3 + ...
                k]].toString());
        }
        csv.push(data);
    }
    setCsvProduction(csv);
}
```

The wake plots and the power plot can be downloaded as PNG using JSZip and File Saver.

Code 3.55: The function which downloads the wake plots as PNG in a ZIP file.

```
1 const downloadWakePlots = () => {
2     const zip = new JSZip();
3     for (var i=0; i < wakePlots.length; i++) {
4         zip.file("cond"+(i+1).toString()+".png", ...
              wakePlots[i], {base64: true});
5     }
6     zip.generateAsync({type:"blob"}).then(blob => {
7         saveAs(blob, 'wake_plots.zip');
8     });
9 }
```

3.4.6 NoAccess (/failed)

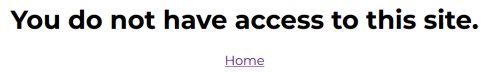


Figure 3.20: The NoAccess component on the failed page.

A failed authentication is redirected to this page.

3.4.7 NotFound (/*)

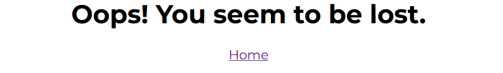


Figure 3.21: The NotFound component on non defined paths.

This component is shown if a user attempts to access a path which does not exist.

Chapter 4

Deployment

The transition from development to deployment, is a critical phase during software engineering. Changing from a development environment to production-ready software, involves rigorous testing, optimization, and coordination between engineers. Ultimately leading to the deployment of a stable and reliable software product. The deployed version is located in the **main** branch of the **wpsim-prod** GitHub repository. (Bakkan & Selchow, 2023)

4.1 From development to production

The process of transitioning a web application from development to production, involves optimizing the application for performance, security and scalability, while also removing quality of life features used during development.

4.1.1 Shifting to modular service architecture

During development, a frontend-backend architecture was used to limit complexity. This decision was crucial in ensuring the project's viability in

its initial stages. By having a monolith backend, development overhead was reduced and streamlined, during the proof-of-concept phase. The reduced complexity resulted in less time spent at facilitating premature inter-service compatibility. The effectiveness of this approach proved to be extremely valuable during the early project phase, setting a strong foundation for subsequent developments.

When shifting to production, transitioning from the traditional frontend-backend architecture, to a modular system was deemed necessary, and challenging. By having developed a monolith backend, with a singular local database, remote database technology had to be implemented. This was implemented, by having the host system share the database, from a singular authentication service, over NFSv3 (Stateless Network File System). Facilitating the ability to run the simulation service on faster, less redundant hardware, as illustrated in figure **??**.

A modular system composed of frontend, authentication, and simulation services, significantly enhanced scalability, without compromising reliability or introducing any significant risks. By deploying individual modules independently, potential downtime could be mitigated, and updates streamlined, creating a robust and flexible solution, ready to meet future challenges.

4.1.2 Switching runtimes

Switching from development to production-ready runtimes reduces overhead and security risks, by removing quality of life features used during development. Production-ready runtimes have stricter security, and will often use a whitelist policy for connections.

NodeJS

The authentication service required transitioning from nodemon, which is a development, on-the-fly, runtime ("Nodemon", n.d.), to a direct NodeJS runtime. Using NodeJS directly, reduced risks coming from development features.

This was done using a single line implementation, in the Dockerfile, as shown in figure 4.1).

Code 4.1: Changes in command for authentication service initialization.

```
1 Original: CMD ["nodemon", "index.js"]
2 Revised:  CMD ["node", "index.js"]
```

The frontend service transitioned from node runtimes to NGINX. Since the frontend only serves static files, NGINX, a purpose built webserver ("NGINX", n.d.), could be used instead. Using NGINX was done by having a multi-step compile, as shown in figure 4.2). The static files was first compiled by NodeJS, before being served to NGINX. By using NGINX, performance, security and scalability was facilitated.

Code 4.2: Changes in Dockerfile, used by the frontend.

```
1  # Node 19 slim image
2  FROM node:19-slim AS builder
3  ...
4  # Install dependencies
5  RUN npm ci --omit=dev
6  ...
7  # Build the React app
8  RUN npm run build
9  # NGINX image
10 FROM nginx:stable-alpine3.17-slim
11 # Copy build files to Nginx
12 COPY --from=builder /app/build /usr/share/nginx/html
13 ...
14 # Start NGINX
15 CMD ["nginx", "-g", "daemon off;"]
```

Uvicorn

The simulation service was transitioned from the uvicorn runtime, to gunicorn. Gunicorn uses a central master process to control a set of workers, which each processes their own request ("Design of Gunicorn", n.d.). By using multiple workers, concurrent request can be processed simultaneously, improving performance scaling on a multi-threaded host server.

This was done by installing gunicorn and modifying the Dockerfile, as shown in figure 4.3).

Code 4.3: Changes in Dockerfile, used by the frontend.

```
1 Original CMD: CMD ["uvicorn", "main:app", "--host", ...
      "0.0.0.0", "--port", "8000"]
2 Revised ENV: ENV WORKERS=8
3 Revised CMD: CMD gunicorn -k uvicorn.workers.UvicornWorker ...
      main:app --workers $WORKERS --bind 0.0.0.0:80 ...
      --log-level debug
```

4.2 From production to deployment

The process of deploying production-ready software involves complex setup to handle traffic and maintain availability. Including using suitable hardware.

4.2.1 Hardware

Hardware used in deployment.

High uptime unit

- CPU: Intel Celeron J4125 (4 cores, 4 threads)
- RAM: 20GB DDR4 RAM
- Storage SSD: 2x 920GB SATA SSD (in SHR1 for redundancy)
- Networking: 2x Gigabit Ethernet ports

This unit is used to host the frontend and authentication service, as well as serving the reverse proxy and NFS file-server. The Intel Celeron J4125 is a low-power CPU, suitable for lighter workloads. The 20GB of RAM provides

headroom for improved IO performance, while the dual gigabit Ethernet ports ensure reliable network connectivity.

Data: The storage used by the services is stored in a SHR1 (1 disk redundancy) SSD volume. The database is stored in the aforementioned volume, in a Btrfs file system, with hourly snapshots and data checksum checking. Snapshots are encrypted before being duplicated to a secondary SHR1 volume. The secondary volume is synchronised with Jottacloud, a Norwegian cloud storage provider.

Data security: If power is lost; full disk encryption must be unlocked at boot.

Performant unit

- CPU: AMD Ryzen R9-5950X (16 cores, 32 threads)
- RAM: 64GB DDR4 RAM
- Storage: 2TB NVMe SSD
- Networking: 1x 2.5 Gigabit Ethernet port

This unit is used to host the simulation service. The AMD Ryzen R9-5950X is a high-performance CPU, that can handle concurrent simulations without affecting the user experience. The 64GB of RAM is sufficient for most workloads, allowing several instances in parallel. The single 2.5 gigabit Ethernet port provide adequate network connectivity, considering the max bandwidth throughput if and when the CPU would be at full load.

Data: The 2TB NVMe SSD provides quick IO, with easily reproducible setup if the drive should crash.

Data security: Critical information is never stored in non-volatile memory.

Always-on unit

- SoC: Broadcom BCM6750KFEBG (3 cores, 3 threads)
- RAM: 512MB DDR3L RAM
- Storage: 256MB eMMC
- Networking: 4x Gigabit Ethernet ports

This unit is used to host critical dormant services, that only goes active once in a while. It hosts the DDNS client, which reports the IP served by the internet service provider, to the domain name registrar. The Broadcom BCM6750KFEBG is a low-power SoC (System on Chip) that can handle light services. The 512MB of RAM is sufficient for the DDNS client, and more services if needed. The four gigabit Ethernet ports provide adequate network connectivity.

Data: The 256MB eMMC is suitable for the required services.

Data security: The eMMC storage is read once at boot, and requires authentication to be modified.

Router

- SoC: Broadcom BCM4912 (4 cores, 4 threads (ARM Cortex-B53 (v8)))
- RAM: 1GB DDR3L RAM
- Storage: 256MB eMMC
- LAN Networking: 4x 1, 1x 2.5 Gigabit Ethernet ports
- WAN Networking 1x 2.5 Gigabit Ethernet ports

This unit acts as the gateway to the WAN, while also port forwarding to the reverse proxy in the "High uptime unit". The Broadcom BCM4912 is a low-power, low-performance SoC (System on Chip) with 1GB of RAM and 256MB eMMC. Ethernet operations are hardware accelerated by Broadcom

Ethernet transceivers, reducing load from the SoC. The four gigabit, and single 2.5 gigabit Ethernet ports provide adequate LAN network connectivity between the units. With the "Performant unit" having the 2.5 gigabit connection.

Data: The 256MB eMMC is suitable for the required services.

Data security: The eMMC storage is read once at boot, and requires authentication to be modified during runtime.

WAN: Considering the computation limitation of downstream internal hardware, a WAN uplink/downlink of 750/750 Mbit/s, is adequate with bandwidth to spare.

WAN security: External control of internal services is accessed trough an encrypted, Wireguard VPN proxy, with multiple steps of authentication. Internal LAN communication is on a whitelist basis.

The hardware used is optimized for the specific needs of each subsystem. The "High uptime unit" ensures data redundancy and handles frontend, authentication, reverse-proxy and file-server tasks. The "Performant unit" does the computationally heavy, non-critical tasks. The "Always-on unit" runs critical dormant services such as the DDNS client. The router serves as the gateway for WAN/internet connections and manages port forwarding. By using several units, configured to deliver optimal performance and reliability for their respective roles, a stable and efficient system is achieved.

4.2.2 Network technologies

To achieve optimal network infrastructure performance and security, the correct usage of networking technologies, both software and hardware, is needed.

A communications map is located in Appendix B. This map showcases the communication lines between a client and the WPSim application, including the communication lines needed for deploying WPSim.

DNS

The Domain Name System (DNS) is a crucial component of internet infrastructure, responsible for translating human-readable domain names into IP addresses. A DNS record is an essential element of web applications, as it enables users to access the application using a human memorable, domain name. DNS facilitates scalability and flexibility for web apps, trough allowing developers to switch hosting providers, or migrate to a different server and IP, with no visible change to the user.

When deploying `wpsim.no`, a DDNS service was deemed necessary to limit downtime. Dynamic DNS (DDNS) is a service that automatically updates DNS records when an IP address change is detected. It is particularly useful for individuals and businesses with dynamic IP addresses, which are common among residential and small business internet connections.

The DNS records used for `wpsim.no` is listed in the 4.1 table. Where the ANAME record, also known as "alias", is a record that allows a domain to be associated with another domain. In the case of wpsim.no, this is the DDNS address. Subdomains are listed with CNAME records, which prompts the DNS resolver to look up the target domain specified in the CNAME data field.

Host name	TTL	RR Type	Data
wpsim.no	1 hour	ANAME	[DDNS address]
auth.wpsim.no	1 hour	CNAME	wpsim.no
sim.wpsim.no	1 hour	CNAME	wpsim.no

Table 4.1: DNS Records for wpsim.no (TTL - Time To Lease, RR - Resource Record)

SSL certificate

Secure Socket Layer (SSL) certificates are a vital element in ensuring security and integrity of data transmitted over the internet. SSL certificates ensures that the data sent over the SSL encryption is not intercepted by unauthorized parties.

For `wpsim.no`, implementing an SSL certificate was a crucial step in safeguarding the privacy of our users and building trust in our web application. SSL certificates not only protect sensitive data but also help to establish the authenticity of a website, ensuring that users are interacting with the intended site.

SSL certificates are issued by trusted Certificate Authorities (CA), which verify the identity of the website owner and issue the certificate accordingly. When a user accesses the `wpsim.no` site, their browser checks the SSL certificate to ensure that it was issued by a trusted CA, and that it is still valid. This validation process helps to prevent man-in-the-middle attacks, where an attacker intercepts the communication between the user and the web server.

In addition to the security benefits, having an SSL certificate also has a positive impact on the search engine ranking and user perception of a website. Major search engines, such as Google, consider SSL certification as a ranking factor, meaning that websites with SSL certificates are more likely to appear higher in search results. Furthermore, modern browsers display a padlock icon next to the URL of a website with a valid SSL certificate, signaling to users that the site is secure and trustworthy.

The `wpsim.no` SSL certificate setup and configuration involved the following steps:

1. Obtaining an SSL certificate from a trusted Certificate Authority (CA).
2. Installing the SSL certificate on the reverse proxy, infront of the `wpsim.no` services.
3. Configuring a Apache instance to redirect HTTP to HTTPS, ensuring that all connections to the site are encrypted.

With the SSL certificate was in place, communication to and from `wpsim.no` was secured. Giving the users confidence that the site is genuine. The SSL certificate, in conjunction with the DNS and DDNS services, contributes to the overall reliability, security, and accessibility of `wpsim.no`.

Port forwarding

Port forwarding enables efficient routing of incoming traffic to the appropriate devices and services within a local network. By configuring port forwarding rules on a network router, administrators can forward incoming traffic to specific devices or services based on the port number. By using port forwarding, services on local networks can be accessed from the wide area network (WAN).

In the case of `wpsim.no`, port forwarding plays a vital role, by managing the flow of traffic, to the services within the network. WPSim, requires two ports to function, 80 (HTTP) and 443 (HTTPS), in a domain pointed setup.

Reverse proxy

The reverse proxy server, a service in the "High uptime unit", acts as an intermediary between the user and the web application. It accepts incoming connections and forwards them to the appropriate services, based on predefined rules and configurations. Using a reverse proxy can not only enhance security, but also facilitate load balancing and traffic management, by distributing requests across multiple servers.

A significant advantage of using a reverse proxy server is the ability to serve HTTPS connections while communicating with services using HTTP. The reduces security risks by reducing the points of failure, while also allowing quick management of the SSL certificates.

To implement this functionality, the reverse proxy is configured with a CA authorized SSL certificate for the `wpsim.no`, `auth.wpsim.no` and `sim.wpsim.no`. When users access the site, their browser establishes a secure HTTPS connection with the reverse proxy. Communication is then done with the reverse proxy in the middle, with the services on LAN using the HTTP protocol.

This approach provides several benefits for the `wpsim.no` web application:

1. **Enhanced Security:** The reverse proxy server ensures that all WAN communication between the user and the web application is encrypted using SSL, protecting sensitive data from unauthorized access and interception.
2. **Quicker development:** By using HTTP for communication between the reverse proxy server and internal services, the development and maintenance overhead associated with encryption is minimized.
3. **Simplified Certificate Management:** With the reverse proxy server handling SSL certificates, administrators can manage certificates centrally, making it easier to maintain and update them as needed.

By forwarding incoming traffic on ports 80 and 443 to the reverse proxy server within the "High uptime unit", traffic flow can be efficiently managed, while maintaining high levels of security, and scalability. This configuration ensures that the web application can adapt to changes in network infrastructure and to increased traffic loads without compromising the user experience.

4.2.3 Physical connections and connection speeds

The physical connections and connection speeds, used in network infrastructure, plays a crucial role in the performance and reliability of web applications.

This subsection details the limitations originating from connections, and network components that form the foundation of the `wpsim.no` architecture. Detailing the "High uptime unit" and the "Performant unit", which host the developed services.

High uptime unit

The unit has dual load balancing gigabit Ethernet connections, which serve to ensure stable communication and data transfer. Incoming WAN traffic, from ports 80 and 443, is directed trough this unit.

Traffic originating from port 80, is forwarded by the reverse proxy, to an Apache server. The Apache server redirects users to port 443, which is used for secure SSL communication. Incoming traffic from port 443, is forwarded to the appropriate services, based on the source domain and subdomain.

This unit could theoretically experience a connection bottleneck, originating from the internet service provider (ISP) connection, if scaled.

Performant unit

The "Performant unit" hosts the the simulation service, responsible for running on-demand FLORIS computations. With a 2.5 Gigabit end-to-end, internal, Ethernet connection, this unit is could theoretically experience a WAN up-link bottleneck, when serving several users. However, considering the processing power, data sent during computation of user requests is not likely to saturate the connection. A processing power bottleneck is more likely to occur, as demonstrated in figure 4.1.

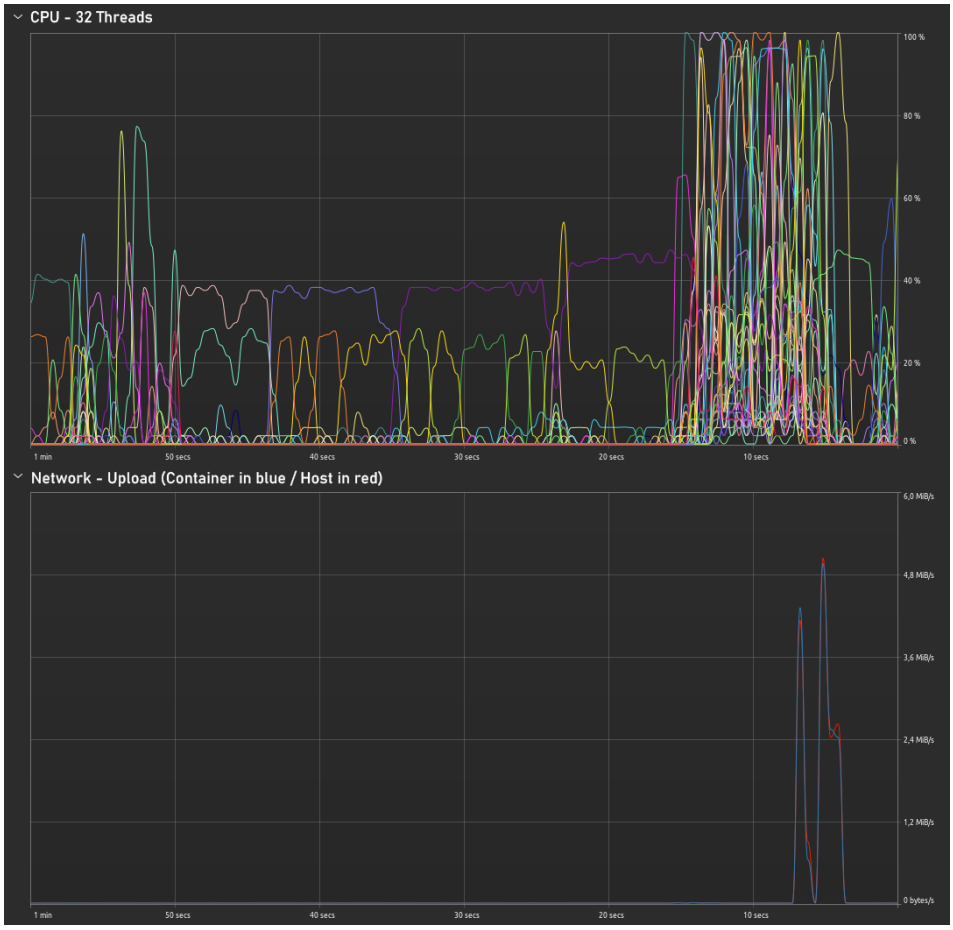


Figure 4.1: Load test for the Performant unit, 3 users requesting all features at once, using the quickstart 3x2 layout.

Chapter 5

Evaluation of software

Evaluating WPSim is an important step towards ensuring software quality and user experience. This chapter details assessments of WPSim, including functional tests, challenges and limitations encountered, and reflects on potential enhancements and future development prospects.

5.1 Functional test

Functional testing is a type of black box testing and was performed to ensure that the application behaves as expected. The tests listed in Appendix A were performed and passed.

5.2 User experience

The user experience (UX) of the software application was carefully considered throughout the development process. The design principles prioritized were responsiveness, and intuitiveness. The aim was to create an interface, that could be easily navigated and understood, by users of various knowledge levels.

To ensure this, the visual layout was designed to be quickly understood, reducing visual clutter that could lead to confusion, or a sense of being overwhelmed. Key functionalities were positioned to be accessible, without having to search the page, reducing the number of steps a user must take to perform an operation. Interactive elements were designed to provide clear feedback, informing users about the actions they've taken.

The UX design aimed to deliver an smooth and responsive experience, contributing to product usability.

5.3 Challenges and limitations

The development of the software faced a number of challenges and limitations. One of the primary challenges was ensuring reliability when handling compute intensive tasks. Designing a system that can effectively handle potential errors or impossible computations, while recovering smoothly is a complex task, requiring planning and testing.

Limitations in the simulation service posed challenges, where a balance between feature-richness and responsiveness had to done. `wpsim.no` resolves this issue by giving the user a set amount of computational time, before the request will inform the user that it has timed out.

Further, the scope of the project and time constraints inevitably led to some features and enhancements not being prioritized, see 5.5. This is a common challenge in software development.

Limitations in the development environment, challenged predicting the WPSim's behavior in deployment. Rigorous testing was carried out when implementing Docker, allowing coverage of most scenarios. Using Docker limited edge-cases which could surface during usage of the development environment.

Improve layout

This feature will attempt modifying the user created layout, creating an optimized version. WPSim "Improve layout" accepts at most 6 turbines. More turbines increases the computational requirements exponentially, on an average case. The feature uses random wind samples, instead of the user samples. This creates a good approximation, without reducing responsiveness. If an accurate approximation is necessary, accurate wind data over several seasons, would be necessary, quickly requiring out of scope computational resources.

5.4 Features considered

Placing turbines on a real map

Being able to place turbines on a real map instead of a canvas, could enhance the application. This was not implemented due to the following:

- FLORIS uses a 2D plane to calculate wakes.
- A real map could give the user too much confidence on approximated data.
- The map would ideally have to be imported from an API, increasing reliance on external services.

Receiving wind samples from

Using an external API for wind data was considered, as it could significantly speed up the setup of a user configuration. No free to use, trustworthy, stable API was found, resulting in searching for other solutions. Seklima was found as a replacement, where users are able to download a CSV, containing wind data, before uploading it to `wpsim.no`.

5.5 Further development

Development items considered, if continuation of the project.

- UX: Drag and drop turbines to new positions, requiring a new canvas system.
- UX: Edit singular turbine positions and wind samples, without delete.
- UX/Sim: Configure the turbine parameters. Requires development of all features comprehensively, as providing users with detailed controls for a singular aspect, might inadvertently lead excessive trust into the precision of the data. Care must be taken to appropriately manage user expectations about FLORIS capabilities.
- UX/Sim: Wake model select. WPSim uses the Gauss Curl Hybrid model, used by default in FLORIS code demonstrations.
- Auth: Refined, non-SQL, access control. Auth0 seems suitable.

Chapter 6

Conclusion

The project, "Web Application for Wind Park Simulation," was undertaken with the objective of developing a web stack for the FLORIS (“National Renewable Energy Laboratory”, n.d.) wake simulation package. Motivation was rooted in the urgency to reduce global warming, by developing a tool that enabled quick online estimation simulations for wind farms, ultimately serving to reduce reliance on fossil fuels.

One of the project's foundational elements was the wake effect concept, a phenomenon causing wind turbines leave behind low energy wind, which significantly affects the energy output of a wind farm. The web application effectively incorporates this concept, successfully meeting one of the primary goals of the project.

Several core objectives were outlined for this project. One of these was ensuring an outstanding user experience, which was achieved thanks to the application's high responsiveness and single-page design. A further goal was keeping application as lightweight as possible, critical due to the intense processing power required for multi-variable simulations. The implementation of OAuth was another successful milestone, providing robust user authentication without risking credentials.

The application also met the objective of enabling users to save simulation configurations. Additionally, it provides on-demand FLORIS power and

production estimation simulations, with visualizations of result data. Users can conveniently download these results, which further substantiates the achievement of the project's core objectives.

Discord and GitHub were central to project communication and collaboration, helping the team to stay synchronized and maintain efficient version control throughout the development process. In terms of architecture, the web application featured a frontend serving static files and an per-case serving backend.

The transition from development to production and subsequent deployment were crucial phases in the project. The team effectively managed this transition through a shift in runtimes, the usage of Docker, and the strategic use of a reverse proxy for hosting Docker containers.

Although functional tests were successful and user experience was responsive, the project did face a significant challenge with the automatic layout improvement function. The function, due to its compute-intensive nature, currently restricts the application to manage only 5-6 turbines before exponentially increasing compute time. This limitation presents a considerable area for future development, including hardware accelerated functions.

Looking ahead to future iterations of the project, a range of enhancements are being considered. These include allowing users to drag turbines to new placements, enabling turbine placement editing without needing to remove and add, offering the ability to configure turbine parameters, and providing an option to change the wake model from the default Gauss Curl Hybrid model.

In summary, the "Web Application for Wind Park Simulation" project successfully met its critical objectives, providing a useful tool for wind farm simulations, and contributing to global efforts to reduce fossil fuel dependence. While challenges remain, the foundation laid by this project promises exciting potential for future development.

Bibliography

Bakkan, E., & Selchow, S. (2023). *Bac-2023-wind-park-sim / wpsim-prod*. Retrieved June 5, 2023, from https://github.com/BAC-2023-Wind-Park-Sim/wpsim-prod

Design of gunicorn. (n.d.). Gunicorn. Retrieved May 24, 2023, from https://docs.gunicorn.org/en/stable/design.html

Discord — your place to talk and hang out. (n.d.). Discord Inc. Retrieved May 23, 2023, from https://discord.com/

Docker. (n.d.). Docker, Inc. Retrieved May 23, 2023, from https://www.docker.com/

Energy statistics - quantities, annual data. (2022). Eurostat. Retrieved May 30, 2023, from https://ec.europa.eu/eurostat/databrowser/view/nrg_pc_204/default/table?lang=en

Express - fast, unopinionated, minimalist web framework for node.js. (n.d.). Express. Retrieved May 9, 2023, from http://expressjs.com/

Html & css. (n.d.). W3C. Retrieved May 9, 2023, from https://www.w3.org/standards/webdesign/htmlcss

Lama Dev. (2021). *React social login with passport.js | react oauth w/ google, facebook, github*. Retrieved May 15, 2023, from https://www.youtube.com/watch?v=7K9kDrtc4S8

National renewable energy laboratory. (n.d.). National Renewable Energy Laboratory. Retrieved May 29, 2023, from https://www.nrel.gov/

National Renewable Energy Laboratory. (n.d.-a). *OpenFAST*. Retrieved May 23, 2023, from https://www.nrel.gov/wind/nwtc/openfast.html

National Renewable Energy Laboratory. (n.d.-b). *TurbSim*. Retrieved May 23, 2023, from https://www.nrel.gov/wind/nwtc/turbsim.html

Nginx. (n.d.). NGINX, Inc. Retrieved May 23, 2023, from https://www.nginx.com/

Node.js. (n.d.). OpenJS Foundation. Retrieved May 23, 2023, from https://nodejs.org/

Nodemon. (n.d.). Nodemon. Retrieved May 23, 2023, from https://nodemon.io/

Nordmann, A. (2014). *Wind turbine int*. Retrieved May 30, 2023, from https://commons.wikimedia.org/wiki/File:Wind_turbine_int.svg

Oauth. (2023). Wikipedia. Retrieved April 27, 2023, from https://en.wikipedia.org/w/index.php?title=OAuth&oldid=1151302967

Om strømpriser. (2023). Statnett. Retrieved April 24, 2023, from https://www.statnett.no/om-statnett/bli-bedre-kjent-med-statnett/om-strompriser/

Optimization and root finding. (n.d.). SciPy. Retrieved May 11, 2023, from https://docs.scipy.org/doc/scipy/reference/optimize.html

Passport - simple, unobtrusive authentication for node.js. (n.d.). Passport. Retrieved May 9, 2023, from https://www.passportjs.org/

Python language documentation. (n.d.). Python Software Foundation. Retrieved May 23, 2023, from https://docs.python.org/3/

Ramírez Sebastián. (n.d.). *Fastapi*. Retrieved May 23, 2023, from https://fastapi.tiangolo.com/

React - the library for web and native user interfaces. (n.d.). React. Retrieved April 27, 2023, from https://react.dev/

React (software). (n.d.). Wikipedia. Retrieved April 27, 2023, from https://en.wikipedia.org/w/index.php?title=React%5C_(software)&oldid=1157304333

Renewable energy – powering a safer future. (n.d.). United Nations. Retrieved May 30, 2023, from https://www.un.org/en/climatechange/raising-ambition/renewable-energy

Sqlite. (2023). Wikipedia. Retrieved April 27, 2023, from https://en.wikipedia.org/w/index.php?title=SQLite&oldid=1149593284

Steiness, C. (2008). *Horns rev wind farm 06*. Retrieved June 3, 2023, from https://group.vattenfall.com/press-and-media/media-bank/wind-solar-and-energy-storage

Typescript. (n.d.). Microsoft Corporation. Retrieved May 23, 2023, from https://www.typescriptlang.org/

Uvicorn. (n.d.). Encode. Retrieved May 23, 2023, from https://www.uvicorn.org/

Wake effect. (2003). Danish Wind Industry Association. Retrieved May 11, 2023, from https://web.archive.org/web/20090630074838/www.windpower.org/en/tour/wres/wake.htm

BIBLIOGRAPHY

What is sqlite? (n.d.). SQLite. Retrieved April 27, 2023, from https://sqlite.org/index.html

Wind energy basics. (n.d.). National Renewable Energy Laboratory. Retrieved May 11, 2023, from https://www.nrel.gov/research/re-wind.html

Appendix A

Functional testing

A.0.1 Login/logout

- Logging in with either Google or GitHub should give the user access to the site if their email is in the **allowed_emails** table.
- Clicking Logout should logout the user.

A.0.2 Redirect/navigate

- Users that are not logged in should be redirected to Login when trying to access any of the pages that requires to be logged in.
- When logged in, Login page should redirect to Home.
- A failed login redirects to /**failed** page.
- Accessing a path which does not exist redirects to shows the **NotFound** component.
- Navbar navigates the user to the appropriate page when clicked.
- Clicking one of the configurations on the Start page should navigate to the Simulation page with this configuration.
- Reloading should show the same page the user is currently on.

A.0.3 Configuration

- Turbines must be 150 meters apart and added to the list when clicking on the layout. Deleting turbines should remove them from the list.
- Wind directions must be a number between 0 and 360. Wind speeds must be a number between 0 and 25. Both must be inserted to add a wind sample. Deleting a sample should remove it from the list.

A.0.4 Save/update/share

- Saving a configuration requires a title. The saved configuration should now appear on the Start page. The config can now be deleted at the Start page.
- Updating an already saved configuration should update this configuration and not save a new one. Clicking on the updated configuration on the Start page should show the updated version.
- Sharing the current config requires a title and an email to share with the given user and gives them a copy of the current config which they can find on the Start page. This copy can be deleted without deleting the original.
- Account type can be changed on the Profile page by clicking the button.

A.0.5 Simulation

- Requesting calculations without turbines or wind samples alerts the user.
- Importing wind samples from Seklima as explained in the README should add the samples to the list.
- Clicking **Turbine power production** should create two graphs: one which shows total power production, and one which shows the turbine powers for each wind sample.
- Clicking **Turbine wind speeds** should show a graph which shows the turbine wind speeds for each wind sample.

- Clicking **Turbine wake plots** should show plots which shows how the wind behaves for each sample.
- Clicking **Turbine power plot** should show a plot which shows the turbine powers for each wind sample.
- Each plot/graph except Turbine power plot comes with a slider which is synchronized with the other sliders. Dragging it should update each graph/plot to the corresponding wind sample. Clicking **Start** should start a simulation where graphs/plots go through the different wind samples given by the simulation speed. The **Reset** button resets the slider.
- Changing **Time per wind sample** under Power production updates the graph and shows the correct numbers.
- Clicking **Improve layout** requires max 7 turbines, wind samples and a Pro account. New and old layout are compared, and the user can choose which to keep. Adding more turbines before choosing keeps the old layout.

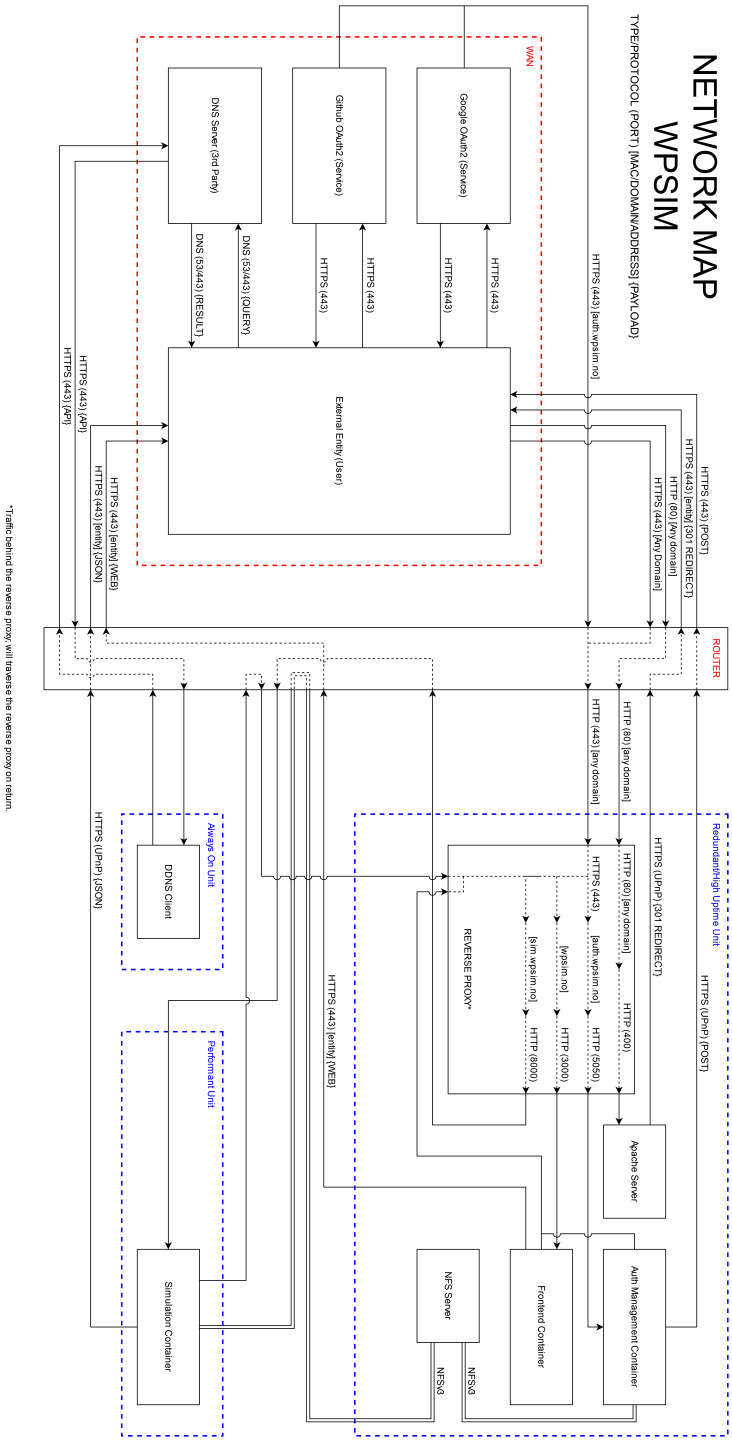
A.0.6 Download

- Downloading the Configuration CSV requires turbines and wind samples to download, and contains the correct values.
- After each calculation it is possible to download the results in appropriate formats (CSV/PNG) and these contain the correct values.

Appendix B

Network map

Network map


NETWORK MAP
WPSIM
TYPE/PROTOCOL (PORT) [MAC/DOMAIN/ADDRESS] {PAYLOAD}
WAN
Google OAuth2 (Service)
Github OAuth2 (Service)
DNS Server (3rd Party)
External Entity (User)
HTTPS (443)
DNS (53/443) {QUERY}
DNS (53/443) {RESULT}
HTTPS (443) [auth.wpsim.no]
HTTPS (443) {POST}
HTTP (80) [any domain] {301 REDIRECT}
HTTPS (443) [any Domain]
HTTPS (443) {API}
HTTPS (443) {entity} {WEB}
HTTPS (443) {entity} {JSON}
ROUTER
HTTP (80) [any domain]
HTTPS (UPnP) {JSON}
HTTPS (UPnP) {301 REDIRECT}
HTTPS (UPnP) {POST}
Always On Unit
DDNS Client
Performant Unit
Simulation Container
REVERSE PROXY*
[sim.wpsim.no]
[wpsim.no]
[auth.wpsim.no]
HTTP (6000)
HTTP (3000)
HTTP (5050)
HTTP (400)
Apache Server
NFS Server
Frontend Container
Auth Management Container
NFSv3
*Traffic behind the reverse proxy will traverse the reverse proxy on return.