Universitetet i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Bachelor i ingeniørfag / Data	Vårsemesteret 2023 Åpen

Forfatter(e): Simeon Vyizigiro, Hammad Munir Hussain, Atle Bjørnstadjordet Ericson

Fagansvarlig: Erlend Tøssebro

Veileder(e): Nejm Saadallah & Yngve Heggeland

Tittel på bacheloroppgaven: Web application for energy system integration

Engelsk tittel: Web application for energy system integration

Studiepoeng: 20

Emneord: testing, web app, simulation	Sidetall: 51 + vedlegg/annet: 56 Stavanger 15. mai 2023

Contents

Table of contents **i**

Declaration **v**

Abstract **vii**

acknowledgements **1**

1 Introduction **1**

1.1 Description of task 1

1.2 Motivation 1

2 Theory & Development Process **3**

2.1 Project Development Process 3

2.1.1 CI/CD 4

2.1.2 Continuous Integration 4

2.1.3 Continuous Delivery/Deployment 5
2.2 GraphQL 5
2.3 OAuth2/Open ID Connect 5
2.3.1 Refresh token rotation 8
2.4 Auth0 9
2.5 Single-Page Application 11
2.6 SvelteKit 11
2.6.1 Server-Side Rendering 12
2.7 Pico.css 13
2.8 Domain Driven Design 13
2.8.1 Ubiquitous Language 14
2.8.2 Domain Modeling 15
2.9 Hyperscalers 18
2.9.1 Azure 19
2.9.2 Azure Container Apps 19
2.10 Docker 19
2.10.1 Infrastructure as Code 20
2.11 Project Management & Agile Methodology 21
2.12 Databases 23
2.12.1 Relational Databases 23

2.12.2 NoSQL Databases 23

2.12.3 Cassandra 23

3 Design and Architecture 25

3.1 Data Architecture 27

3.1.1 Data Model 28

3.1.2 Persistence Method 32

3.2 Technical Architecture 33

3.2.1 Technology Stack 34

3.2.2 Deployment Environment 35

3.2.3 Authentication 37

3.3 Application Architecture 37

3.3.1 Bounded Contexts 38

3.3.2 Event-Driven Messaging 41

4 Results & Discussion 43

4.1 User Story 43

4.2 Results 46

4.3 Development Process 48

4.4 Improvements 49

5 Conclusion 51

Bibliografi 55

Appendix 55

A Source Code 56

Declaration

I, Simeon Vyizigiro, Hammad Hussain and Atle Bjørnstadjordet Ericson, declare that this thesis titled, "Web application for energy system integration" and the work presented in it is our own. I confirm that:

- This work was done wholly or mainly while in candidature for a bachelors's degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work. I have acknowledged all main sources of help.

Abstract

This bachelor thesis describes the development of a web application that allows users to configure and save simulations for an energy system.

The application utilizes a pre-existing simulator pack to generate real-time results, which are displayed to the user in chart form. The development stack includes Sveltekit, FastAPI, and GraphQL, with the latter used to retrieve and display data. The application features a login system and user-specific configuration options, enabling users to tailor simulations to their needs.

The system is built to be scalable and easy-to-implement new features such as a better system for storage and automated test. The team focused on long term for this project and made sure it would be easy for other people to work on it later.

The resulting application provides an efficient and user-friendly means of simulating energy systems, with potential applications in both research and industry.

Acknowledgements

We would like to thank Nejm Saadallah and Yngve Heggeland for being our supervisors.

We would also like to thank our families for being supportive and encouraging for the last 5 months of work.

Finally, we would like to acknowledge all the individuals who have played a part in shaping our academic and personal development, including our professors, colleagues, and friends. Thank you for your guidance, insights, and friendship.

Chapter 1

Introduction

1.1 Description of task

The Norwegian Research Center (NORCE) has developed a power system simulator (python package) consisting of a combination of a wind turbine, wind-speed simulation, battery pack, gas turbines, and power demand simulation. Each component can be configured separately before running a simulation and plotting results. The task is creating a web application where a user can plot real-life data simulating various energy demands and production scenarios.

The simulation we used for this project was already created by NORCE, and the project was all about visualizing the data from the simulation with a free range of technology to do this.

1.2 Motivation

The main objective of this project was to improve the visualization and user experience of the simulation data. Previously, the data output was obtained through console logs and required additional software to create graphs and visuals. Our task was to simplify this process by developing a user-friendly web application that enables users to plot different scenarios and view real-time results through charts. In addition, there needed to be implemented a feature that allows users to easily download the data from the website.

Chapter 2

Theory & Development Process

2.1 Project Development Process

This project required us to use the knowledge gathered from the previous semesters on how to set up a full-scale working application. This application required a Database, front-end, and back-end component. We needed to sort out which database would be best for the deployment of the application and discuss futuristic scalability possibilities. Every choice we made, was dependent on how it would work long-term. Was this the best option for us? What is the upside/downside? Is it easy to make changes if they want to make some? These are some of the questions we asked ourselves throughout this project. We also decided to use technologies that were familiar to us and commonly used. This was done to ensure that there is enough documentation on how to change the application if someone would wish to do so.

Throughout this project, we used tools like Miro and Jira to plan ahead. These tools are widely used to make mind maps and a plan for working with the project. We set up different tasks that each had a time limit to be completed. The reason for this is a continuous effort put into the project to assure that we would be done in time. Jira is one of the most widely used development tools to use for teams to work agile. Here you can create tasks and assign them to each other. It was very easy to use, with a good modernistic user experience making it clear to us what to do. We always made sure to update the Jira confluence before and after meetings to update the questions and answers we had. This was then used to further plan our application. To develop this application, we got handed a branch on a GitHub repository where the simulation package was added. Using GitHub for development is something we were familiar with beforehand, after having used it for previous projects.

2.1.1 CI/CD

CI/CD stands for Continuous Integration/Continuous Delivery and is a software development method that emphasizes the importance of integrating, testing, and delivering code frequently [38]. During our development of this application, we continuously pushed code and ran tests to make sure it all worked at all times. Bugs and lack of function were checked often. We also had meetings with our supervisors every week, where we got to display our application and how it developed over time. By continuously having a close connection with our supervisors, we were able to make necessary changes, and further improve our application.

2.1.2 Continuous Integration

Practicing continuous integration means that developers constantly merge changes to the main branch when making a change. This is done so that one can pick up any errors that may occur when running the main branch after a change is pushed. Writing automated tests ensure that the application builds as expected and doesn't run into any errors. It is also done so that your partners ensure that they are working with the latest version of the application when implementing changes [38]. .

2.1.3 Continuous Delivery/Deployment

Continuous Delivery/Deployment builds upon continuous integration since it automatically deploys all code changes to the testing/production environment after the build stage. By releasing as often as possible, you ensure that the application is working at all times, especially when changes are made [38]. If you make many changes at once, you risk having build errors due to many different things merging at once without being tested. We continuously showed our supervisors what was done, which changes were made, and what it looked like.

2.2 GraphQL

GraphQL is a query language that was developed by Facebook back in 2012 before being publically released in 2015. GraphQL allows clients to retrieve only the data they need, reducing the amount of data transferred over the network. This leads to significantly better performance. GraphQL is also very flexible when it comes to data structure. Due to its hierarchical nature, it allows users to retrieve complex data all over one single request. GraphQL supports real-time updates through subscriptions, which work as WebSockets. [4].

To use GraphQL, the Python library Strawberry was used. Strawberry is a GraphQL library for Python designed to be user-friendly, type-safe, and high performing. It was created due to some complications using regular GraphQL in Python. Strawberry also integrates with famous Python frameworks like Django, SQLAlchemy and FastAPI [11]. GraphQL returns a result in a JSON format. JSON objects work very well with Python due to Python having built-in functions for JSON [8].

2.3 OAuth2/Open ID Connect

OAuth2 is an authentication and authorization framework that enables third-party applications to access resources on behalf of a user, without the user having to give their password. Instead, the third-party application receives an access token from the server once the user authenticates with a server that is trusted by both the user and the program. This access token can then be used to access the user's resources. OAuth2 defines four roles:

1. Resource owner: An entity capable of granting access to a protected resource.
2. Resource server: The server which hosts the protected resources and is capable of accepting and responding to requests using access tokens.
3. Client: An application making requests on behalf of the resource owner and with its authorization.
4. Authorization server: The server issuing access tokens to the client after successfully authenticating the resource owner.

OpenID Connect is a simple identity layer in top of the OAuth2 protocol. It allows clients to verify the ID of end-users based on the authentication performed by the authorization server, as well as to obtain basic profile information about a user. OAuth2.0 was not built for authentication, so OpenID connect adds several additional features to provide these authentication abilities. Features like ID tokens and UserInfo are used for authentication of users.

Both OAuth2 and OpenID connect have very similar flow. To fully understand the flows, we can look at some visual examples. First we have the flow of OAuth2:

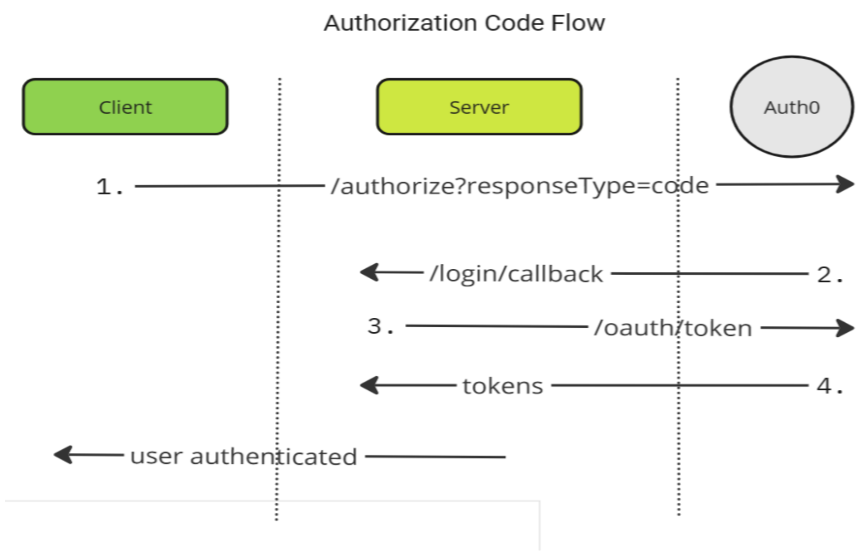


Figure 2.1: Visual image of the authorization code flow.

Using this flow our web application (1) redirects users to Auth0's login site when they initiate a login request. If the authentication request succeeds it (2) redirects to the server callback route. At the callback route, the server obtains an authoriza- tion code from the URL query parameters. Using this randomly generated code issued by Auth0, the server can use this code to (3) request the ac- cess, id, and refresh tokens. The server then stores the refresh token in a cookie. After the server receives the tokens it can then request a user's information using the newly received access token. If the application is able to get the user information successfully the web app considers the user as authenticated. To explain OpenID connects visually, there are some components to define. These are:

1. Relying party (RP): The client application that the user is trying to log in to.
2. OpenID provider (OP): The server that authenticates the user and provides identity information to the RP.

Now to visualize the process, we have taken a diagram from [16]. This is a bit more complicated to understand, so we will go through each step. Authentication = AuthN, Authorization = AuthZ.

```
+--------+                                   +--------+
|        |                                   |        |
|        |--------(1) AuthN Request--------->|        |
|        |                                   |        |
|        |  +--------+                       |        |
|        |  |        |                       |        |
|        |  |  End-  |<--(2) AuthN & AuthZ-->|        |
|        |  |  User  |                       |        |
|   RP   |  |        |                       |   OP   |
|        |  +--------+                       |        |
|        |                                   |        |
|        |<--------(3) AuthN Response--------|        |
|        |                                   |        |
|        |---------(4) UserInfo Request----->|        |
|        |                                   |        |
|        |<--------(5) UserInfo Response-----|        |
|        |                                   |        |
+--------+                                   +--------+
```

1. The RP sends a request to the OpenID provider.
2. The OP authenticates the end-user and obtains authorization.
3. The OP responds with an ID token and an Access token.
4. The RP can send a request to the UserInfo endpoint with the Access token.
5. The UserInfo endpoint returns information about the end user.

2.3.1 Refresh token rotation

Refresh token rotation is a technique used to get new access tokens using refresh tokens. Refresh tokens are usually longer-lived and can be used to get new access tokens after short-lived access tokens expire. With refresh token rotation enabled in Auth0, every time an application exchanges a refresh token to get a new access token, a new refresh token is also returned. This way, you don't have a long-lived refresh token that can be stolen and used as a security threat [10]. A refresh token is a string granted to the client from the resource owner. Unlike access tokens, refresh tokens are intended to be sent directly from Auth0 to the client and never sent to the server [15].

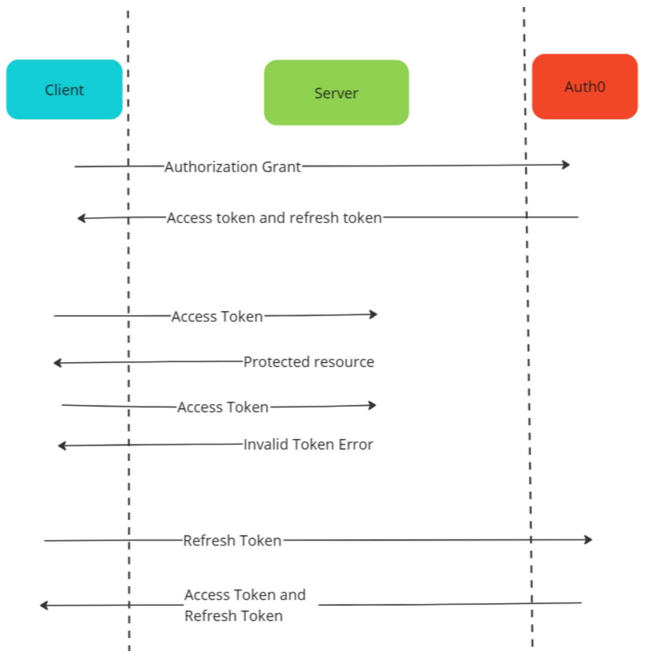
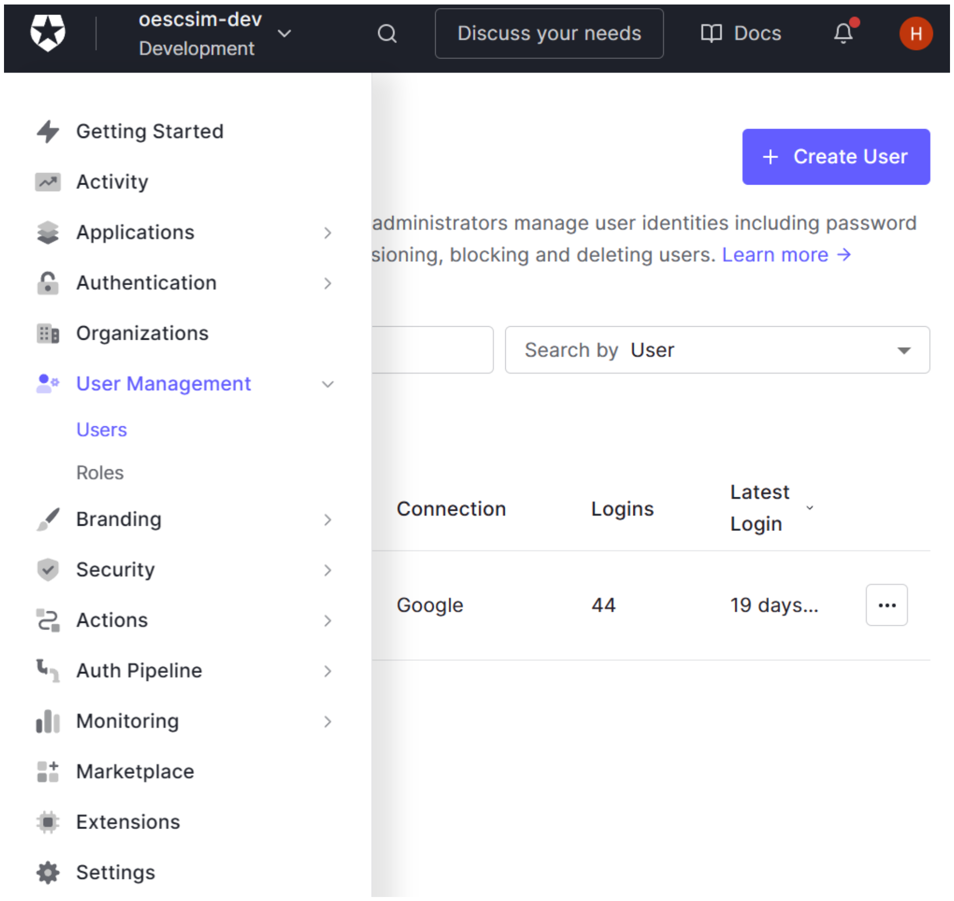


Figure 2.2: This figure explains how a user obtains refresh tokens.

2.4 Auth0

Auth0 is a cloud-based identity and access management platform that provides authentication and authorization as a service. It is widely used as a tool to build websites with secure user authentication, password-less login, and multi-factor authentication (MFA). Auth0 and OAuth2 are connected as Auth0 can act as an authorization server that allows applications to authenticate users and obtain access tokens that can be used to access protected resources. In our application, we use Auth0 to have an overview of IAM tools. Auth0 supports a free plan available for up to 7500 active users, which is very fitting to the scale of our application. At no point will the web simulation application be used by that many users simultaneously. Auth0 is very popular and is used by companies like AMD, Pfizer, and Mozilla [7]



administrators manage user identities including password sioning, blocking and deleting users. Learn more →

Connection	Logins	Latest Login
Google	44	19 days...

Figure 2.3: We have an overview of users, that can be assigned roles, deleted, or altered as we wish. This is a good overview of keeping the application safe and controlled.

2.5 Single-Page Application

Traditional websites are built as multi-page applications (MPA). A MPA is a web application where page navigation and user interactions triggers a load from the server. In recent years software developers have discovered that this approach has some disadvantages. Some of these disadvantages include:

- Bad user experience
- Performance
- High coupling between backend and frontend
- Difficult to maintain large web applications

Single-page applications (SPA) were created to fix some of the problems with MPAs. In contrast to a MPA, a SPA is run entirely in your browser using JavaScript (JS) to navigate and handle user interactions. This way a user only has to request the HTML from the server once and subsequent requests are made using JS.

When using an SPA the browser requests an empty HTML page and a single JS file. After the JS is loaded it programmatically changes the HTML DOM object to show the website. Compared to MPAs this means that the initial page load may be slower when using an SPA.

2.6 SvelteKit

Sveltekit's very own Svelte frontend framework is designed to be very developer friendly, and has many benefits with the main one being performance. Sveltekit has performance benefits over traditional web development frameworks, and a more simplified code structure. Sveltekit compiles the application code into highly optimized Javascript, CSS, and HTML files that have a more efficient cooperation with the browser than traditional client-side frameworks. This approach leads to faster load times, improved page rendering, and reduced network latency. The framework also provides a clean and concise API that is not hard to learn. The built-in serverless deployment capabilities of Sveltekit includes a set of adapters that allow applications to be deployed to popular serverless platforms such as AWS Lambda, Google Cloud Functions, and Azure Functions [13].

2.6.1 Server-Side Rendering

Sveltekit has built-in support for Server-side rendering which allows developers to easily generate HTML content on the server and send it to the client's browser. This means that the response content is generated when needed, dynamically. On a dynamic website, HTML pages are normally created by inserting data from a database into placeholders in HTML templates. A dynamic site can return different data for a URL based on the information given by the user [6]. Most of the code run to support a dynamic website must run on the server, hence 'server-side' rendering.

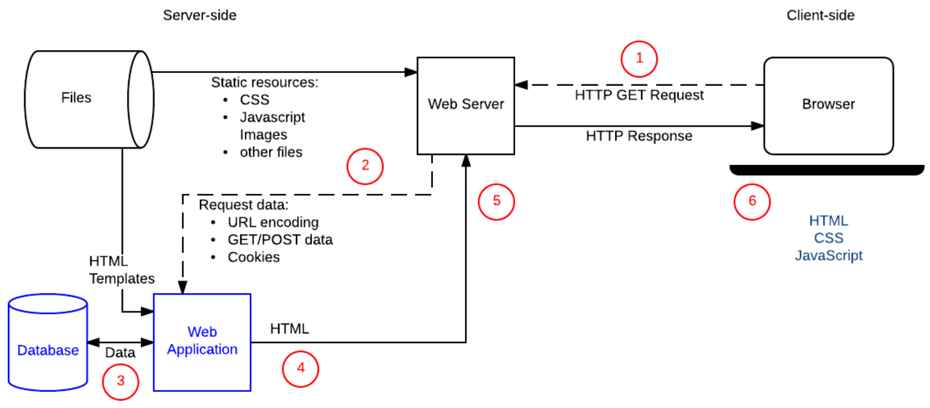


Figure 2.4: Visual example of server-side rendering

Sveltekit has an SSR implementation that works by generating HTML content for each route defined in the application. When a user requests a route, the server generates content for that route and sends it back to the client's browser. This is shown in figure 2.4. Sveltekit also provides several optimisation techniques such as lazy-loading and data fetching. Lady-loading loads components only when they are needed, which reduces the amount of HTML content that needs to be generated on the server. Data fetching allows the server to pre-fetch data required for a particular route, which reduces the amount of time required to generate the HTML content [6].

2.7 Pico.css

Pico.css is a minimal CSS framework that offers a clean and lightweight design system. It was chosen to be used in the project to facilitate quicker front-end development. The decision to use Pico.css was made early on in the project to maintain a good look and feel of the website while focusing on implementing functional details. The combination of Pico.css with TailwindCSS, a utility-first CSS framework, provided flexibility during the creation of the user interface. [9].

2.8 Domain Driven Design

Domain-driven design (DDD) advocates for a model based on the reality of a business as relevant to the use case. When we look at it in the context of building applications, DDD talks about problems as domains. it describes independent problem areas as bounded contexts and emphasizes a common language to talk between these problems. DDD's most important part is organizing the course so it is aligned to the business problems, and using the same business terms. It should be structured so that the code is written specifically for the business problem is it created to solve [2].

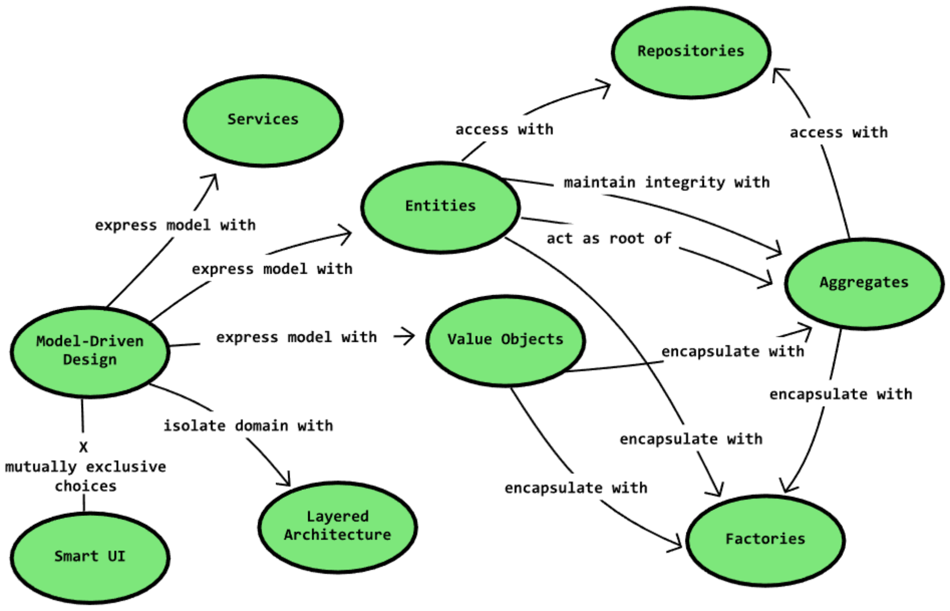


Figure 2.5: Figure explaining the different components in DDD.

2.8.1 Ubiquitous Language

Ubiquitous language is a term that is used to describe a language that is shared and commonly understood by the whole application team, not just the domain experts. The developers use the ubiquitous language and translate it into code. The use of ubiquitous language in building an application ensures that everyone is able to understand the different domains and that different domains share no ambiguity. Making sure to represent your application goals in your code is what ubiquitous language is all about [17].

To explain what we mean by ubiquitous language, let's have a look at an example. For instance, imagine you are creating a site where a client should be able to change email. How should this goal be represented in the code? You can for instance write the code like this:

```python
class People(Entity):
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def update(self, name=None, email=None):
        if name:
            self.name = name

        if email:
            self.email = email
```

This does not look like ubiquitous language because it is very general in for example the function language. The `update()` function is a general function that can be used not only for email change but also for other parameters. If this function was to be used by for example aggregates, domain services, or events that also need to express a ubiquitous language, then this would ruin the connection. To make it appear in a ubiquitous manner, something like this would work:

```python
class Client(Entity):
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def change_email(self, email):
        self.email = email
```

The difference in writing code like this is that it resembles the goal it is created to fulfill. This is important in DDD to create a structured, well-organized code readable to everyone working on a project.

2.8.2 Domain Modeling

Domain modeling is an approach used in software engineering to create a conceptual model of a system. The idea behind this is to find entities and the relationships between them. You can then use this to create your codebase based on the different objects. In order to guarantee that specific relationships and objects are established and user demands are addressed, this is typically done early in the project development process. There are four well-known objects in domain modeling:

- Boundary object: These are objects that are at the boundary of your application. Any object that takes input from or produces output to another system can be labeled as a boundary object.
- Controller object: These objects coordinate activities of a collection of entity objects and interface with the boundary objects to produce a default behavior of the system. These are the object that "control" and make use of other objects.
- Entity object: These are the objects that correspond to real-world entity in a problem space. For example Simulations, Books, User, etc.
- Value objects: Value objects represent a value in your domain. They are immutable and have no identity [33].

After a domain model is created, it can then be used in the development process to make sure the structure remains intact. This also helps developers when creating the application to stay effective and efficient.

Context Mapping & Aggregates

Context mapping in DDD represents a global vision of business ideas. For example, if we look at a theatre, we have two contexts: Operational (staff, movies, schedules) and Sales (customer, purchase tickets). In the operational context, the employee is the main actor and controls which movies to play and the schedule. In the sales context, the customer is the main actors and does the ticket purchase. Every time a ticket is bought, it affects the amount of space in the theatre room for that movie [26].

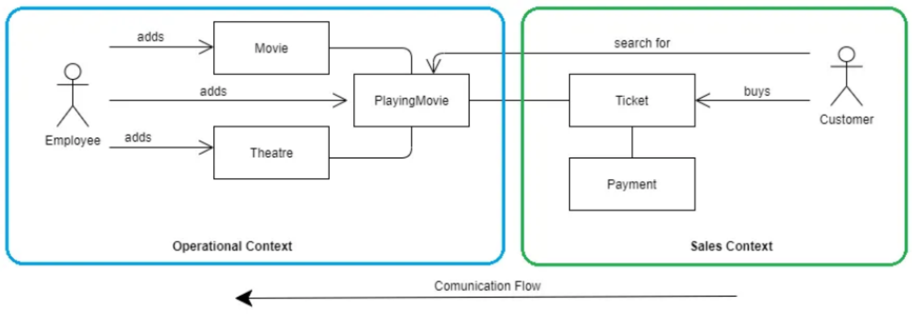


Figure 2.6: Operational and sales context in a context map. Figure is taken from [26]

Having this displayed as a visual figure helps developers understand what there is to build, and how to structure the code. It also displays the relationship between the contexts which is very important to create ubiquitous language.

Aggregates, within the context of Domain-Driven Design, are clusters of entities and value objects that represent complex real-world concepts that involve multiple entities. The primary goal of an aggregate is to accurately model the intricate relationships between these entities in the domain.

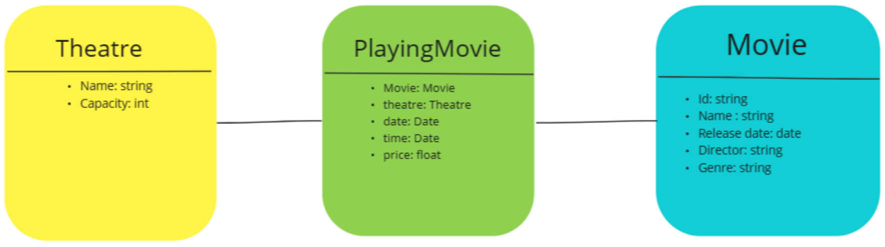


Figure 2.7: Aggregates of the entities from figure 2.6.

Domain Events

In Domain-Driven Design, a Domain event refers to an occurrence within a particular domain that triggers a notification to other components within the same domain. The notified components typically have specific protocols to follow when such events occur, and the communication between components happens asynchronously. The use of Domain events enables the expression of domain rules based on a common, ubiquitous language, while also promoting better separation of concerns among classes within the domain [35].

Event storming is a collaborative workshop where a group works together to identify the domain events and the corresponding commands required to execute them, to create a complete domain model for a project. Typically, a mind map is used to visualize the relationships between entities, aggregates, and events. The main objective is to establish a shared understanding of the application's domains among all participants [22].

2.9 Hyperscalers

As mentioned earlier in chapter 2.6, Sveltekit has built-in support for serverless deployment capabilities which makes it easier to connect it with a hyperscaler. A hyperscaler is a large cloud service provider, that provides services such as computing and storage at larger scales. It is a way to scale and grow software architecture. Hyperscalers are able to support services like computing, storage, and other workloads on a grand scale while using multiple datacenters around the world with thousands of physical servers running millions of virtual machines. This type of outsourcing of infrastructure management allows developers to fully focus on building applications without worries of maintenance [18]. There are several famous cloud vendors such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP).

2.9.1 Azure

Through Microsoft's extensive network of data centers, businesses may create, test, deploy, and manage apps and services using the Microsoft Azure cloud computing platform. Azure offers a variety of services such as virtual machines, storage, analytics, and networking. It also supports a huge range of programming languages, including Sveltekit. The flexibility of Azure is one of its strengths. Businesses can use Azure to support a public cloud service, a private cloud service, and also a hybrid cloud service. Due to this flexibility, Azure is very famous and widely used. Cloud computing has its benefits in paying-as-you-use deals and the easy options to scale and enlarge an application. Azure also has integration with a range of Microsoft applications such as Power BI, Office 365, and Dynamics 365. For businesses that use Office applications, using Azure will come naturally [23].

2.9.2 Azure Container Apps

Using Azure container apps, deploying the application, and managing it became simplified. It gives the ability to contain the application in an environment where you have an overview of different components such as API endpoints, background processing applications, and microservices. This technology allows developers to focus on creating the application, rather than thinking about infrastructure management such as patching, scaling, and availability [19]. Kubernetes can be used with Azure container apps to provide a fully managed platform for deploying and managing applications on Azure. Kubernetes is a technology that automates the deployment, scaling, and management of containerized applications. It comes with a set of useful tools such as load balancing, automatic scaling, and self-healing [20].

2.10 Docker

Azure container apps use Docker containers as the underlying technology to deploy and run containerized applications[19]. Docker is a platform that allows developers to build, ship, and run containerized applications. This is a method of virtualization that allows multiple isolated environments, and containers, to run on a single host operating system. The whole idea behind Docker is to allow an application to run similarly on all computers regardless of the operating system. The applications are packaged within a container to make this possible [12].

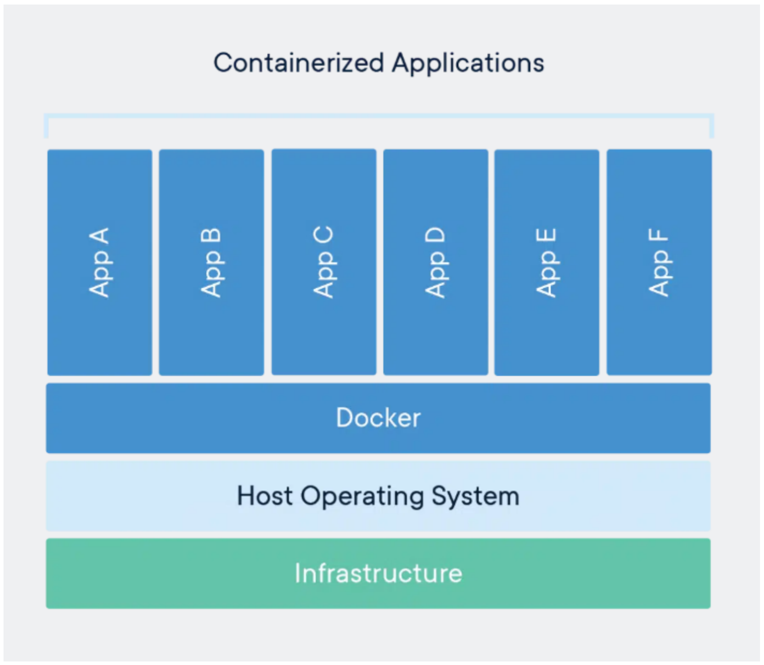


Figure 2.8: Visual example of what Docker looks like.

2.10.1 Infrastructure as Code

Infrastructure as code (IaC) is a way to approach IT infrastructure where configurations are defined and managed using code, rather than manual processes. By doing this, developers and operations teams can automate the creation, deployment, and management of infrastructure. There are many benefits with IaC such as speed, consistency, and agility [28]. In order to have IaC in our application, we used Terraform. Terraform is a tool that allows you to define your infrastructure in a simple configuration file which can be version controlled and then shared with team members. This configuration file specifies the different resources you want to create, their properties, and dependencies if there are any. Terraform then creates, modifies, or deletes the resources to match the desired state as given by the configuration file [14].

2.11 Project Management & Agile Methodology

To create an application in a team, you need to break it up into smaller problems and solve them one by one. To be the most efficient while doing this, requires a team to be on the same wavelength when it comes to building an application. Having a plan on how to structure the project is very widely used, so there are different techniques on how to do this. The most famous one is by working "Agile".

Agile methodology is designed for project management and software development. The goal of Agile is to find a method on how to reach from point A to B in a project and do so in the most efficient way possible. There are different ways of doing this, such as shorter work cycles, sprints, or extreme programming (XP). The two most famous Agile methodologies are Scrum and Kanban[30].

Scrum is a method in which work that needs to be done is broken into smaller pieces known as sprints. These sprints encourage quick production. You also have daily scrum meetings where a team will have a short meeting at the beginning of the day to discuss what has been done and what the daily goals are. There is often a sprint backlog with different tasks that need to be implemented, where you plan on which ones to work on each day [30].

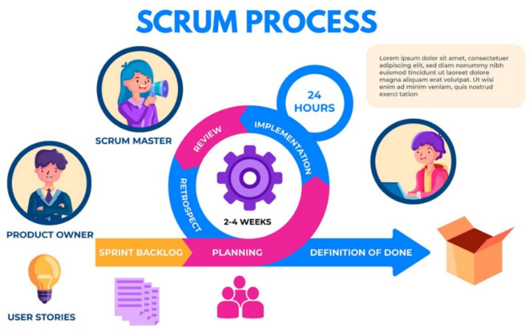

SCRUM PROCESS
SCRUM MASTER
PRODUCT OWNER
USER STORIES
24 HOURS
REVIEW
IMPLEMENTATION
RETROSPECT
2-4 WEEKS
SPRINT BACKLOG
PLANNING
DEFINITION OF DONE
Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation

Figure 2.9: How Scrum works in a business context.

Kanban is a method in which you use a visual system of cards to track work. The theory behind this method is to split work into smaller tasks, and then assign these tasks to the ones who work on solving them. The tasks are then put in different stages such as 'tasks, in progress, done' where team members have an overview of what is being done and what is left [30].

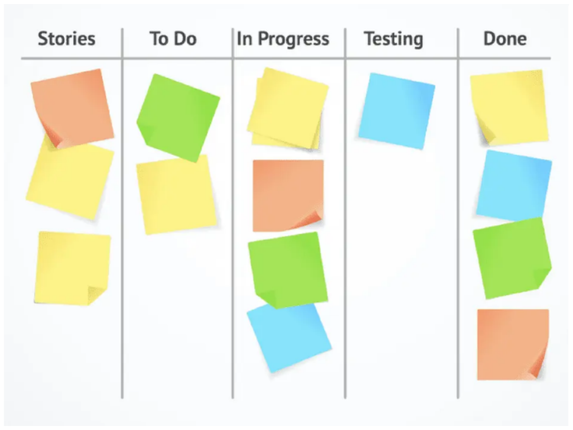


Figure 2.10: A Kanban board can look like this.

2.12 Databases

Most web applications use databases to store different types of data. There are a lot of different options to choose from when it comes to database and structuring of data.

Setup of a database can take time if you have a lot of data and different types of relations, and data types.

2.12.1 Relational Databases

Traditionally you would use a relational database such as MySQL, Oracle DB, or PostgreSQL. The conventional approach to representing data in a database involves using two-dimensional tables with strict consistency in transactions. Real-time reading and writing capabilities allow for complex SQL queries, including those involving multiple tables. However, when dealing with large amounts of data, multi-table queries can become inefficient[44].

2.12.2 NoSQL Databases

A new wave in database storage is NoSQL databases. This is necessary because of the rapid increase in the amount of data needed to store.

The majority of NoSQL systems prioritize high performance, availability, data replication, and scalability in distributed databases or data storage. This differs from traditional relational databases, which prioritize immediate data consistency, powerful query languages, and structured data storage.[39]

2.12.3 Cassandra

Cassandra is a NoSQL database that is made to manage large amounts of data, it can even manage petabytes of data (1M GB). It is decentralized and has high scalability.

Cassandra is designed across multiple servers, this provides high availability without single points of failure. Cassandra also consistently outperforms other NoSQL databases because of the fundamental architectural setup.[32] Cassandra is used by huge companies like Netflix, Apple, and Reddit. [1]

Chapter 3

Design and Architecture

The design phase of the software development process is of paramount importance, as it lays the foundation for the entire project and heavily influences the system's quality, maintainability, and overall success [25]. During this critical stage, developers translate requirements into a detailed design, ensuring that the system's structure and components are well-defined, efficient, and robust [37]. The design phase also enables developers to identify potential challenges and make informed decisions, thereby reducing risks and minimizing rework during later stages of development [41]. Furthermore, research has shown that investing time and effort in the design phase can lead to significant cost savings, as a majority of software defects can be traced back to inadequate design [24]. In summary, the design phase is integral to the software development process, as it sets the stage for a successful, high-quality software system.

Creating a good and sufficient architecture became a top priority for us. To achieve this there was a need for understanding of the domain and the application requirements better by collaborating closely with the domain experts. Involving stakeholders in the development process, particularly during the design phase, is of paramount importance as it fosters a collaborative environment and ensures that the resulting system meets the needs and expectations of its users and other parties [21]. Engaging stakeholders facilitates a better understanding of the requirements, allows for early identification of potential risks, and helps in establishing a shared vision of the project's goals and priorities [31]. This collaborative approach leads to more accurate and complete requirements, reducing the likelihood of costly rework and enhancing overall project outcomes [45]. Moreover, stakeholder involvement promotes a sense of ownership and commitment, fostering a smoother adoption process and enhancing the long-term success of the software system [40]. In summary, engaging stakeholders throughout the software development process, and especially during the design phase, is crucial for ensuring that the resulting software system is both relevant and effective in addressing the needs of its users and other interested parties.

Upon acquiring a more comprehensive understanding of the various requirements, refining the requirements and scope of the project scope accordingly became imperative. Continuous refinement of requirements and scoping during the design phase is crucial as this iterative approach facilitates the identification and incorporation of new requirements or changes in existing requirements, as a response to evolving user needs [34]. Continuous refinement also enables teams to address issues and potential risks early in the design phase, leading to improved project outcomes [27]. Moreover, this practice allows for the ongoing adjustment of project scope, ensuring alignment with stakeholder expectations and avoiding scope creep, which can result in cost overruns and delays [29]. In essence, the continuous refinement of requirements and scoping during the design phase is vital for ensuring the software system remains relevant and adaptable in the face of changing circumstances, ultimately contributing to project success.

The ultimate requirements for the application remained consistent with the initial specifications. Nevertheless, through collaborative sessions with domain experts, it became evident that the application's objective was to mitigate the intricacies associated with managing data input files and executing simulations. Consequently, functionalities were incorporated to enable users to upload input files for demand and wind speed data.

- The user has to authenticate before usage (login)
- Predefined simulations cases should be made available for the end-use for a "quick-start"
- The user should be able to configure wind speed simulation parameters.

- The user should be able to configure gas turbine parameters.
- The user should be able to configure wind turbine parameters.
- The user should be able to configure battery pack parameters.
- The user should be able to pause, run, stop, start, fast forward, and fast backward a simulation case.
- The user should be able to export the results to appropriate file formats (CSV, images,. . .).

It was determined that certain additional properties, such as high maintainability and scalability, were important to achieve. High maintainability was essential due to the uncertainty surrounding the project's future maintenance personnel. Consequently, it is important to establish a solid foundation that enables subsequent developers to comprehend and maintain the code effectively. Furthermore, high scalability emerged as a vital property, as the system is anticipated to process substantial amounts of data stemming from simulations and input files. Collaborative sessions with domain experts revealed that a single simulation could generate gigabytes of data, and the input files provided by these experts contained megabytes of data.

In the context of this paper, Information Technology (IT) architecture is comprehensively defined as a composition of technical architecture, application architecture, and data architecture [43]. Additionally the term "domain experts" refers specifically to our supervisors, who possess extensive knowledge and understanding of the particular domain in question.

3.1 Data Architecture

> The data architecture is an abstract representation of the data files, databases, and relationships to the application architecture. [43]

3.1.1 Data Model

Given that the task was supplied with the core simulation code, our primary task involved constructing our application to complement the existing implementation. This approach allowed us to focus on ancillary aspects of the system without the need to devote resources to the development of the central functionalities.

As the project proceeded to design the data model, it relied heavily on the code snippet furnished by our supervisors to establish a foundation for our application's structure.

Kode 3.1: Snippet from demo simulation code provided by our supervisors.

```
control=Control(nwtr=2,nb=2,bat_MaxE=10e6,bat_Initial=30, ...
    low=30,high=90)
steps=3600*4
dt=1
wind_speed=np.zeros(steps)
wind_speed.fill(20)

Demand=np.zeros(steps)
Demand.fill(-15e6)

policy = Policy(LowBatteryPolicy.DYNAMIC_LIMIT, ...
    ChargingPolicy.LIMITED, 0.1)
result_df = control.simulate(WS=wind_speed, D=Demand, ...
    steps=steps, dt=dt, policy=policy)
```

In the design phase of our project, we substantially leveraged the principles and concepts of DDD 2.8. Our primary aim was to create a flexible data model that would support the evolving needs of our application. Two critical DDD concepts that were incorporated into our design strategy were Bounded Contexts 3.3.1 and Aggregates. Furthermore, we pursued to attain a loosely coupled system - a characteristic that enhances the modularity and flexibility of the application architecture. Adhering to these DDD practices is crucial as they significantly enhance the maintainability of the project by promoting clear boundaries, reducing dependencies, and facilitating a more robust, evolvable system. As a result of this design strategy, we identified and developed the following aggregates:

- User
- Simulation
- Configuration
- WindSpeed
- Demand

The `Simulation` entity encapsulates a single instance of a simulation case within our system. This entity holds the necessary configuration properties to instantiate a `Control` object and invoke its `simulate` method. These properties were identified and defined through a thorough examination of the simulation code examples, generously provided by our supervisors (3.1). In addition to the configuration properties, the `Simulation` entity maintains reference IDs for both wind speed and demand input files. These references facilitate the retrieval and utilization of the required input data for each simulation case.

Kode 3.2: JSON representation of the `Simulation` aggregate.

```
{
    "id": "350bd848-e290-4257-9217-a1811db5fa85",
    "name": "Sim 3",
    "user_id": "user id",
    "configuration": {
        "steps": 50,
        "dt": 1,
        "nr_batteries": 2,
        "nr_wind_turbines": 3,
        "battery_max_energy": 10000000,
        "battery_initial_charge": 30,
        "policy": {
            "low_battery_policy": "FIXED_LIMIT",
            "charging_policy": "FULL",
            "charging_limitation": 0.1
        }
    },
    "ws_file_id": "a3649424-19f2-4d20-b510-5d2f70b61cba",
    "demand_file_id": "86cde126-53d7-47e0-bca0-9317e6654817"
}
```

The `WindSpeed` and `Demand` entities are designed to encapsulate metadata and information related to the respective stored files within our system. At the current stage of implementation, these entities incorporate three key pieces of data: the name of the entity, a user's identifier, and the entity's unique identifier. This encapsulation ensures efficient access and manipulation of data associated with wind speed and demand files.

Kode 3.3: JSON representation of the `WindSpeed` and `Demand` aggregate.

```
{
    "id": "00000000-0000-0000-0000-000000000000"
    "user_id": "user id",
    "name": "Some Name"
}
```

The `Configuration` entity essentially mirrors the configuration property of the `Simulation` entity. The purpose of this entity is to enable users to generate predefined templates that can be leveraged when creating new simulations. Interestingly, the `Configuration` entity is not directly referenced by the `Simulation` entity. This design choice permits users to independently modify configurations and simulations, fostering greater flexibility and control over the simulation parameters. In addition to the configuration parameters, the `Configuration` entity also includes properties that encapsulate the user's identifier and the name of the configuration. These attributes further enhance the functionality and user-centric nature of our application.

Kode 3.4: JSON representation of the `Configuration` aggregate.

```
    "id": "00000000-0000-0000-0000-000000000000",
    "name": "Some Name",
    "user_id": "user id",
    "steps": 50,
    "dt": 1,
    "nr_batteries": 2,
    "nr_wind_turbines": 3,
    "battery_max_energy": 10000000,
    "battery_initial_charge": 30,
    "policy": {
        "low_battery_policy": "FIXED_LIMIT",
        "charging_policy": "FULL",
        "charging_limitation": 0.1
```

The `User` entity serves to track the resources created by each user within the system. Originally, we aspired for the `User` entity to reside entirely within Auth0, thereby maintaining clear boundaries and reducing redundancy. However, due to time constraints, we opted to include a `User` entity within our application as well, in addition to the user information housed in Auth0. This decision allowed us to decrease the complexity of our application, thereby enabling us to make further progress in its development. The data that we store within our application for each `User` entity includes the user's identifier, which we retrieve from the access token 3.5 provided by Auth0. Additionally, we maintain arrays of reference identifiers linking the user to the other entities within our system. This arrangement facilitates efficient tracking and retrieval of user-associated resources.

Kode 3.5: JSON representation of the `User` aggregate

```
{
    "id": "auth0 obtained id",
    "simulations": [
        "00000000-0000-0000-0000-000000000000",
        "00000000-0000-0000-0000-000000000000"
    ],
    "configurations": [
        "00000000-0000-0000-0000-000000000000",
        "00000000-0000-0000-0000-000000000000"
    ],
    "ws_files": [
        "00000000-0000-0000-0000-000000000000",
        "00000000-0000-0000-0000-000000000000"
    ],
    "demand_files": [
        "00000000-0000-0000-0000-000000000000",
        "00000000-0000-0000-0000-000000000000"
    ]
}
```

This data model contains the minimal amount of data needed for the application to function correctly. It also facilitates for each entity to develop separately which is of importance as they live in different contexts. If the aggregates are highly coupled they wouldn't be able to develop entirely on their own, thus also reducing the maintainability of the application.

3.1.2 Persistence Method

Given our observations from the collaboration sessions, it became evident that both the input files and the data generated from simulations were quite large. Considering this, we identified a hybrid persistence mechanism as the most optimal approach for data management. Using a relational database (RDBMS) for storing aggregates would yield performance benefits during data querying and offer superior data integrity compared to file storage. Given that the application doesn't modify the content of any input files, these files are essentially immutable. As such, storing these files in file storage introduces the least complexity while adequately meeting the application's requirements.

Conversely, the data emanating from simulation runs constitutes the majority of the data managed by the application, with a single simulation run often reaching gigabytes in file size. Through our discussions with domain experts, we also understood that the data output for each simulation case could vary based on the number of specified gas turbines. These requirements make Cassandra the most optimal storage method for the simulation generated data. Cassandra offers a scalable and reliable persistence method and is capable of handling petabytes of data. And being a NoSQL database, Cassandra offers a flexible database schema that allows for storing unpredictable data.

Nevertheless, time constraints necessitated a simplification of our implementation approach. As a result, we decided to resort to file storage for all data, thereby reducing the complexity and expediting the development process. Consequently, our approach to data storage is as follows: the aggregates are retained as JSON files, providing a structured and human-readable format. The input files are directly stored as text files, maintaining their original state for unaltered access. Lastly, the data generated from simulations is preserved in CSV files, offering a tabular and easily manipulable format for large datasets.

To successfully preserve the bounded contexts it is important to establish the boundaries at the storage level as well. Therefore each bounded context is assigned its own directory. Within this directory, only data related to that context will be stored.

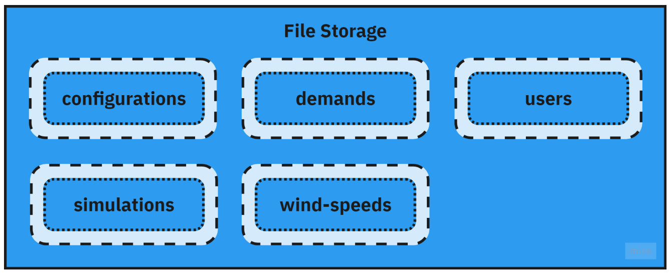
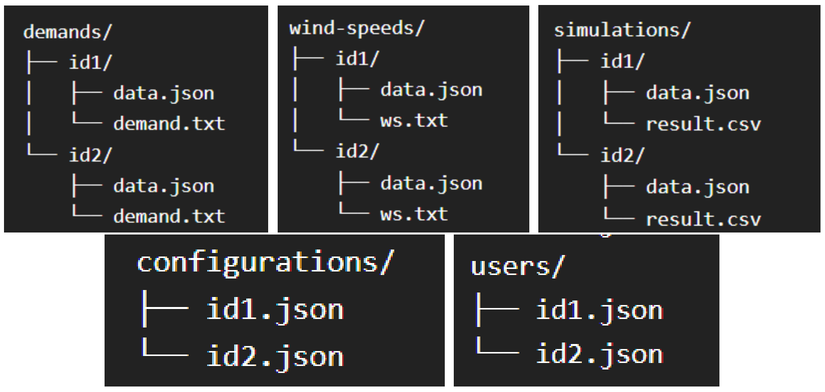


Figure 3.1: Each bounded context is assigned a directory where all its respective data lives.



```
demands/
├─ id1/
│   ├─ data.json
│   └─ demand.txt
└─ id2/
    ├─ data.json
    └─ demand.txt
```

```
wind-speeds/
├─ id1/
│   ├─ data.json
│   └─ ws.txt
└─ id2/
    ├─ data.json
    └─ ws.txt
```

```
simulations/
├─ id1/
│   ├─ data.json
│   └─ result.csv
└─ id2/
    ├─ data.json
    └─ result.csv
```

```
configurations/
├─ id1.json
└─ id2.json
```

```
users/
├─ id1.json
└─ id2.json
```

Figure 3.2: The exact folder structures of each directory. The `data.json` and the `id.json` files store the aggregate entities. The `result.csv` file in the simulations folder store the data generated from a simulation run. The `.txt` in the `wind-speeds/` and `demand/` folders are the input files that users can upload.

3.2 Technical Architecture

The technical architecture is an abstract representation of the platforms, languages, and supporting technological structures that provide the "plumbing" that keeps the environment functioning. [43]

3.2.1 Technology Stack

The application consists of a frontend created using SvelteKit, Auth0 as an Identity Access Management (IAM) service, and a Python backend written with the FastAPI and Strawberry GraphQL frameworks. The communication between the frontend and backend happens mostly over GraphQL and all other communication happens using REST.

The application architecture comprises a frontend developed utilizing the SvelteKit framework, supplemented by Auth0, serving as an Identity Access Management (IAM) service. The backend is built with Python, employing the FastAPI framework for building APIs and the Strawberry GraphQL framework for constructing a GraphQL API. The primary mode of communication between the front and backend is through GraphQL, while any remaining interactions are conducted using REST. The term 'frontend' refers to both the browser-based application, often referred to as the client, and the server-side component that manages frontend operations.

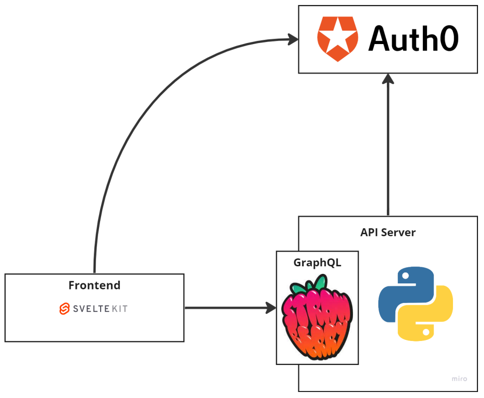


Figure 3.3: The technical architecture. Each arrow represents network communication. All network communication happens over HTTP.

The technologies chosen were chosen to enhance our development speed and performance. As the application would be running real-time simulations performance is essential to provide a smooth user experience. By choosing Auth0 we didn't have to implement any authorization and authentication logic ourselves. We chose SvelteKit both for its performance advantages, but also the reputation for its excellent developer experience. For our backend, we chose to use Python frameworks because the code we were provided was already written in Python. Since performance and development speed was our main priorities we decided to utilize the FastAPI framework. FastAPI applications running under Uvicorn are benchmarked as one of the fastest Python frameworks available[3]. FastAPI also lists development speed as a key feature of the framework in its documentation [3].

Finally, we chose to use GraphQL to communicate between the frontend and backend application. GraphQL's strongly-typed schema and introspection feature allows the API to be self-documenting, making it easier for developers to understand the available data, queries, types, and interfaces, ultimately leading to increased developer productivity. Moreover, GraphQL's strongly-typed schema provides a contract for the data, enabling tools to leverage these schemas for better developer tooling, automation, and optimization. This type of safety also enhances error handling and debugging, offering more precise and informative error messages. To query the API we used HoudiniClient [5] as a GraphQL client in our frontend application.

Additionally, we utilized TailwindCSS and Pico.css [9] to help improve the UI and UX.

3.2.2 Deployment Environment

Our software application has been constructed with containerization in mind, utilizing Docker technology as the foundation for this approach. This containerization strategy ensures the application's portability and compatibility across diverse execution environments.

In our deployment experiments, we successfully deployed the containerized application to Microsoft Azure via Azure Container Apps (ACA). After comprehensive evaluation and research, we determined that ACA was a cost-effective and straightforward solution for hosting our application.

One of the critical features of ACA is its serverless architecture. This architecture allows the service to dynamically scale the application resources in response to the incoming traffic. In periods of high demand, ACA can increase resources to maintain performance, while in low-demand periods, ACA can automatically scale down to zero.

Importantly, the ability of ACA to scale down to zero during periods of no traffic presents a significant cost advantage. When the application is not in use and scales down to zero, it does not incur any operational costs.

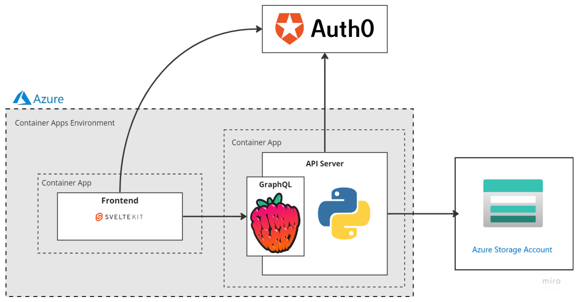


Figure 3.4: The application in the deployment environment.

As a result of our decision to use file storage for our application, it becomes necessary to utilize volume mounts. This measure ensures that files persist across different instances of the container. Within the Azure infrastructure, this persistence is achieved by mounting directories and files to Azure File Shares. This is a reliable mechanism for our application to store and access data across container instances.

Additionally, ACA provides a distinct advantage in terms of its integrated network infrastructure. All containers in ACA are positioned behind a managed reverse proxy, which offers automatic support for secure HTTP connections (HTTPS). This arrangement simplifies our development process significantly as it obviates the need for us to implement any specific HTTPS handling logic within our application, thereby enhancing both security and efficiency.

3.2.3 Authentication

One requirement of the The application employs the Authorization Code Flow for user authentication, a standardized authorization procedure outlined in the OAuth2 specification [15]. Upon initiating a login request, the user is directed to the Auth0 login page. Following a successful login attempt, the client receives an access token, which encapsulates relevant information about the authenticated user. The obtained access token can be utilized to access the backend API. When the user initiates a request to our backend, the access token is validated. Should the access token be invalid, the request is consequently unsuccessful.

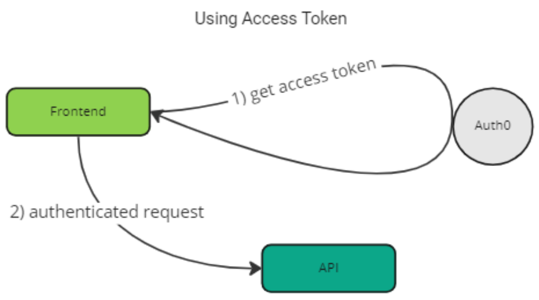


Figure 3.5: Abstract representation of how to get and use an access token

3.3 Application Architecture

> The application architecture is an abstract representation of the applications, subsystems, services, and orchestration structures and their relationship to each other and to the data. (W. Ulrich, 2010)[43]

In this section, our focus will be solely on the architectural design of the backend. We have not made any specific architectural design decisions regarding the frontend application, and hence it will not be addressed in the subsequent discourse. The structure of our codebase can be comprehended utilizing a general mental model illustrated in figure 3.6.

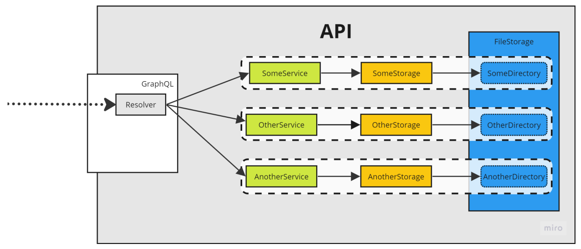


Figure 3.6: Upon initiating a request to the backend API, the first point of contact is a resolver. Each module in our architecture is divided into a service layer and a data access layer. The resolver should solely interact with the service layer of any given module. The service layer's responsibility is the conversion of internal objects into Data Transfer Objects (DTOs) and vice versa. Conversely, the data access layer functions as an abstraction layer to the storage mechanism, primarily entrusted with the transformation of data into the requisite internal objects.

Because all our data is stored on the file system, each `Storage` object currently depends on a `FileStorage` object internally. The `FileStorage` object operates as an abstraction layer to the file system, contributing primarily for convenience. A significant detail to note is that the boundaries of the modules are meticulously maintained down to the storage level, further emphasizing the discrete nature of each module.

3.3.1 Bounded Contexts

To attain high maintainability, we employed concepts and principles derived from various architectural styles, primarily Domain-Driven Design (DDD) and Hexagonal Architecture. Our principal objective was to develop a modular monolith, known for offering exceptional maintainability. A modular monolithic architecture bears a resemblance to a microservice architecture, with the primary distinction being that a modular monolith encompasses all the modules or microservices within a single monolithic structure.

We conducted a context map of the business domain to identify suitable modules to divide our application into. The context map delineates the distinct subdomains, or bounded contexts, present within our business domain. Each subdomain assumes complete responsibility for a portion of the business logic.

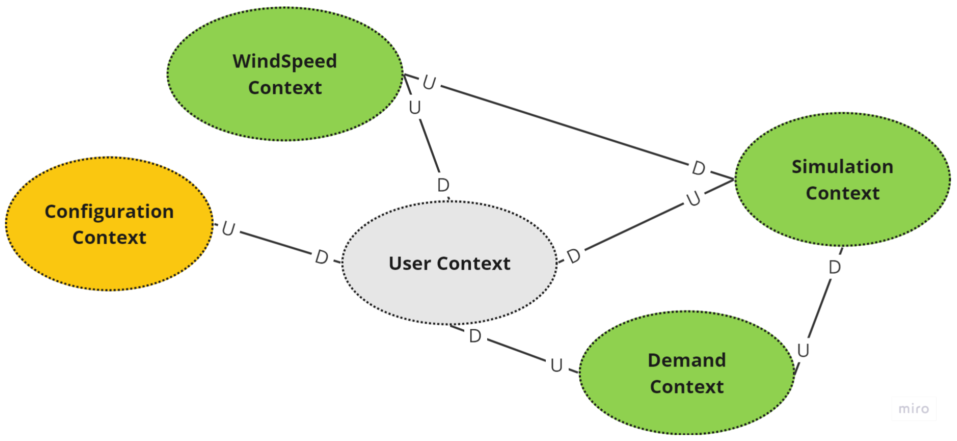


Figure 3.7: The subsequent context map depicts our final domain classification. The color yellow denotes a core domain, green denotes a supporting domain, and gray denotes a generic domain. The interconnecting lines between each context provide a visual representation of the communication dynamics among the contexts. In this setup, an upstream context (U) initiates communication to a downstream context (D).

In DDD, subdomains can be categorized as core, supporting, or generic domains. Core domains embody the essence of the business, encapsulating its unique value proposition and competitive advantage. In contrast, supporting domains, although not as critical as core domains, play a vital role in facilitating the effective functioning of core domains. They typically represent essential yet non-differentiating aspects of the business. The design of supporting domains should focus on enhancing and facilitating the operation of core domains. Generic domains offer standard and reusable functionality across diverse applications and industries, addressing common concerns such as authentication, logging, or data storage. They can be modeled utilizing established patterns and pre-existing solutions, allowing developers to concentrate on designing and refining the core and supporting domains.

In our software architecture, the application identifies the simulation context as a core domain since it addresses the primary purpose of the application: running simulations. The demand, wind speed, and configuration contexts are classified as supporting domains, as they assist users in managing resources necessary for running a simulation. Finally, we recognize the user domain as a generic domain, where we employ Auth0 to manage our users.

At the data level each bounded context has a single aggregate attached to it. Each bounded context defines a boundary between different segments of the business logic, resulting in each context assuming complete responsibility for its respective entities. These entities encapsulate the data consumed and stored by the context, thereby promoting a clear separation of concerns and facilitating the maintenance of the associated business logic.

The entities are internal objects that should never be used outside of the context. Outside of their contexts entities should be represented as data transfer objects (DTO). A DTO is used as a certain representation of an entity outside of it's respective context. This means that each context's service layer, figure 3.6, should always digest and emit DTOs. The purpose of using DTOs is to allow the internal implementations to develop freely without breaking other applications that depend on the already established interface. Using DTOs therefore further enhances the maintainability of the application.

Domain Events

Together with the context map we also conducted an event storming exercise in order to identify the domain events, chapter 2.8. The exercise resulted in the domain events visualized in figure 3.8. The most noticeable characteristic is that most of the domain events are consumed by the user context. This is because the user context is responsible of tracking the resources that a user owns, thus whenever a user stores or deletes a resource it should be reflected in the user context as well.

Additionally the simulation context consumes domain events whenever a wind speed or demand file is deleted. This is to remove any reference to that input file and ensure that no simulation tries to access the deleted file.

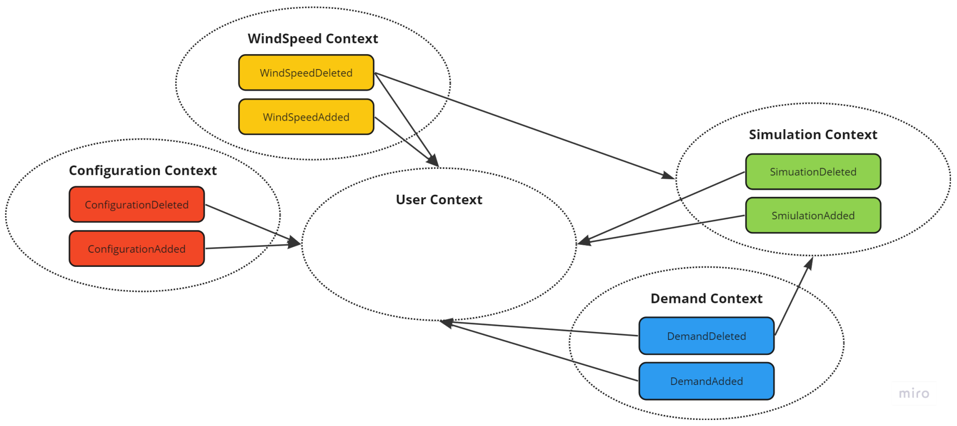


Figure 3.8: Visualization of the domain events and an overview of which contexts consume each event. A consuming context is referred to as the downstream context and the emitting context is referred to as the upstream context, figure 3.7.

3.3.2 Event-Driven Messaging

> A messaging bus is a way to achieve loose coupling between applications in a distributed system by exchanging messages via a logical bus. The bus transmits messages between the connected applications; the listening applications decide what messages they will listen to. Software participating in the system only needs to be aware of how to connect to the bus and how to publish or consume appropriate messages. Applications that publish information to the bus do not need to know about which applications are consuming that information, and similarly the applications consuming data do not need to know about those publishing it. - (P. Onyisi 2015) [36]

If a context were to handle the change in a different context directly it would require for the upstream context to know about the downstream context, and thus breaking the boundary between the two contexts. Because the bounded contexts establishes a boundary within our code, and if this is implemented correctly, the implementation within any bounded context can be changed without affecting any other context. As a result the application would get a truly modular and loosely coupled software system.

However, sometimes it is necessary for one context to communicate with another context. In order to react to domain events in a way that conserves the boundary between contexts, the team wanted to use a message bus. When designing our domain model we wanted the only communication between contexts to be domain events. Therefore the application can utilize a messaging bus to communicate between bounded contexts whilst preserving the boundaries. A consequence of this is that our software system would become eventually consistent. Even though messaging systems are most commonly used in distributed software systems, utilizing them in our application will enhance modularity and flexibility. These properties are highly desirable as it facilitates for a smoother and incremental transition to a microservice architecture. Moving an application to a microservice architecture becomes relevant when the application becomes large and/or complex.

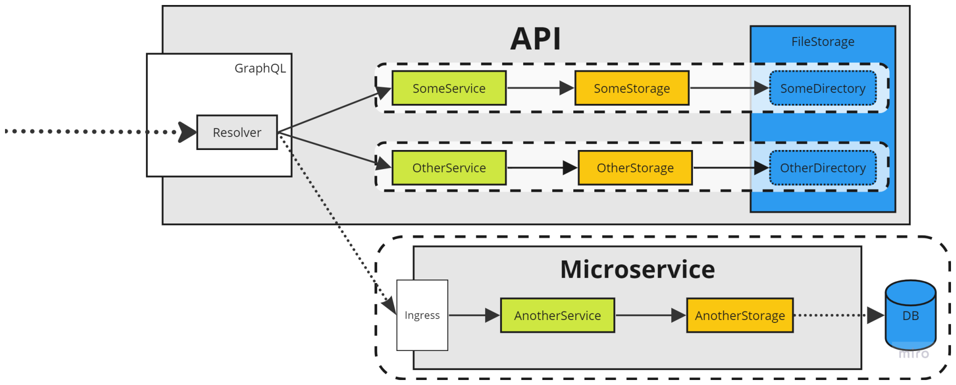


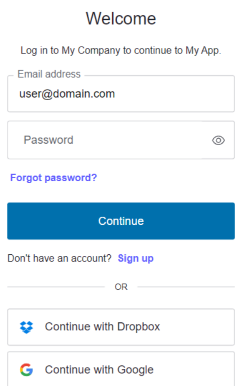
Figure 3.9: The architecture of our application facilitates the extraction of any bounded context into a microservice without influencing the other contexts. The solitary necessary refactor would entail the resolver forwarding the incoming request to the microservice and subsequently returning the response. The dotted lines in the illustration symbolize a network request.

Chapter 4

Results & Discussion

4.1 User Story

In this section, you will go through the process by which a user would interact with our application. Initially, the user would be required to create a user/login by providing the requisite information. Following this, the user would proceed to navigate to the profile page.


Welcome

Log in to My Company to continue to My App.

Email address: user@domain.com

Figure 4.1: Here is how our login looks, and the ability to change it very easily using Auth0.

4.1 User Story

All the functions available in this application can be conveniently accessed through the user's profile page. This is where the user can add simulations, wind speeds, and different configurations. These configurations can serve as "quick start" templates for new simulations.

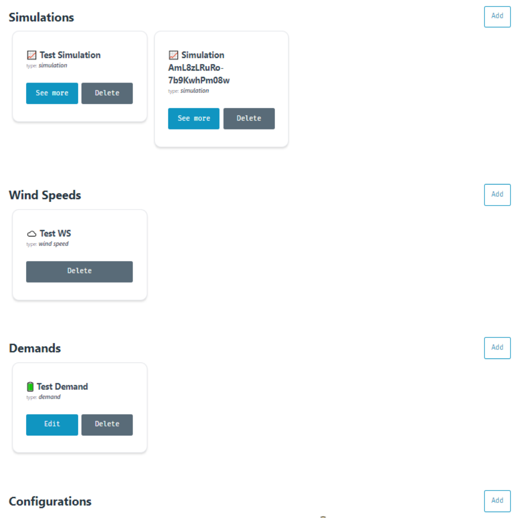


Figure 4.2: Here is how our Profile page looks, easy to navigate and use.

Before you can create your simulation both a wind-speed text file and a demand text file need to be uploaded.

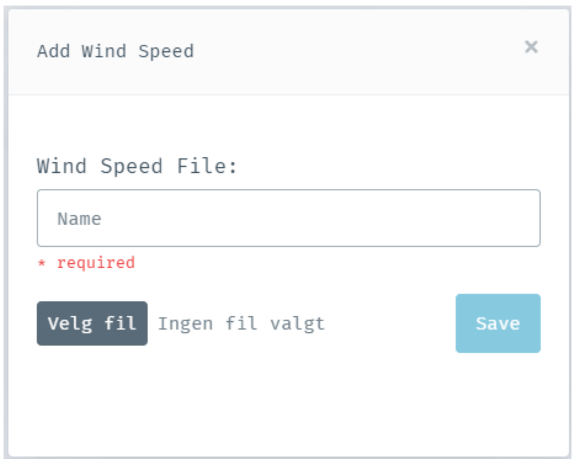
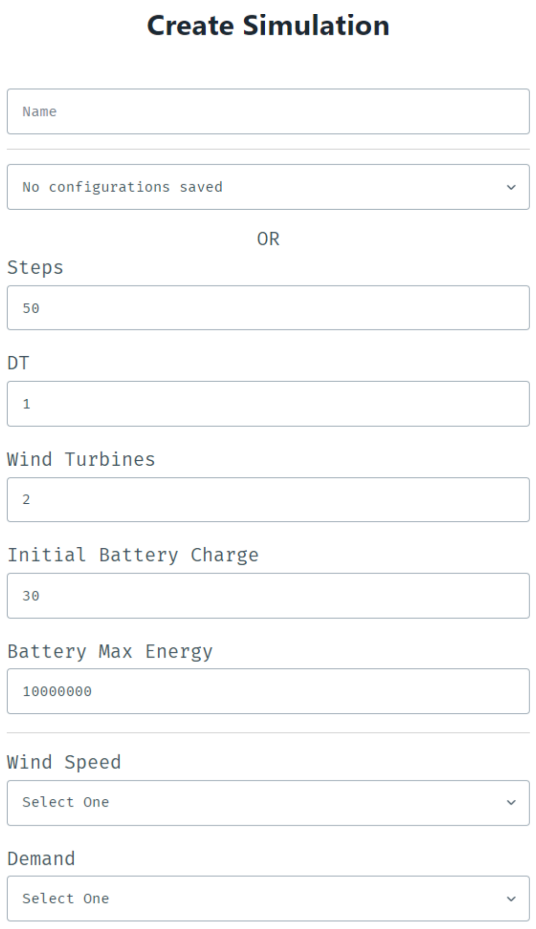


Figure 4.3: This is how you upload a wind-speed file, add a .txt file.

After configuring the desired settings, the user can proceed to create a simulation by selecting the "add simulation" option. The user can then specify the number of steps or iterations to go through, as well as any other specific data values required for the simulation.



Create Simulation

Name	
No configurations saved	
OR	
Steps	50
DT	1
Wind Turbines	2
Initial Battery Charge	30
Battery Max Energy	10000000
Wind Speed	Select One
Demand	Select One

Figure 4.4: Here you have all the values to create a simulation.

Upon completing the input of all relevant information, the user can initiate the simulation by selecting the "create" button. This will generate two tables with graphs that represent the simulation. The user can manipulate the simulation by selecting the "start, stop, fast forward, and slow down" options. The simulation result can also be downloaded, and the configuration can be edited to generate different results.

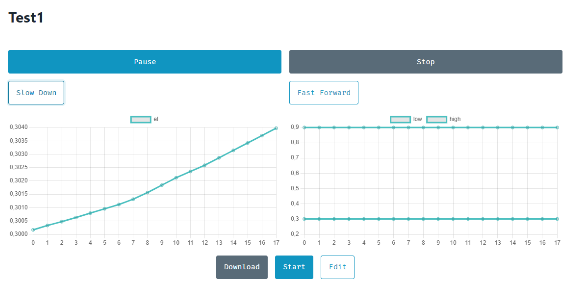


Figure 4.5: Here you have the simulation in progress.

After completing a simulation, the user can return to their profile page to select their next action, whether it be adding additional wind-speed files or creating and saving a new simulation with similar configurations. The previously generated simulation will also be available on the profile page, with the option to delete it if required.

We are happy with the application we developed. There were times when the communication between us and the supervisors was a bit challenging, but we ended up completing all the sub-tasks that we were assigned. Visually, it is a simple application that is functional. Thanks to Pico.css, the front-end was not something we had to worry about. Using Sveltekit and FastAPI came naturally to us, with the challenge being GraphQL. Since GraphQL was a new type of technology for us, it took some time to get used to it and utilize its potential.

4.2 Results

All of our assessment are based on the teams opinions and none of them are based on any quantitative measure.

Whilst designing our architecture we used multiple concepts and principles from different, however related, architectural styles in order to fit our needs. Our most important goal was to create a modular backend API. We managed to achieve this goal, and the mental model illustrated in figure 3.6 fits very well with the final implementation. In order to verify this we track the dependencies in each file. This modular architecture was vital to make the system both flexible and maintainable. However this modularity is highly dependent on each developer being disciplined and understanding how changes may affect the modularity. We believe this is largely due to our inexperience of using Python.

Python posed multiple challenges when implementing the architecture. The biggest challenge we encountered was Python being a dynamic programming language. Because of this, whenever we made changes to a the signature of a function or method, we wouldn't get a warning in our code editor. This led to us having to manually verify that the dependent code was changed to fit the new signature. However, the way Python handles modules helped to prevent us from creating circular references in our code.

All in all if we could've chosen any programming language for our backend we probably would've chosen to use a strongly typed language. Most likely C# as the entire development team is more experienced with the language. However, many of the challenges that we faced could be mitigated by utilizing automated tests. The lack of automated tests is a big flaw in our system. During development we mitigated testing our application to increase our development speed, but in hindsight it most likely slowed us down. This led to manually testing the application every time we made extensive changes to our code base, an approach that proved to be error prone and time consuming.

The rest of the technical architecture ended up being a great fit for the application. Using SvelteKit combined with Pico.css and TailwindCSS allowed us to create a beautiful UI and UX easily. Pico.css was especially useful as we didn't have to spend an extensive amount of time to create a beautiful and cohesive styling for the website. Using Pico.css is one of the technologies that enhanced our frontend development speed comprehensively.

Another tool that was important in our development was Houdini. Houdini utilizes Svelte being a compiler and GraphQL APIs being strongly typed to auto generate types and code to interact with the backend API. Houdini's most significant advantage was it ensuring type safety between our backend and frontend. Any changes that was made to our API was instantly reflected in our frontend.

The finished application ended up being more extensive than the initial requirements. As previously mentioned, throughout our development process we refined and redefined the project requirements which led to a better understanding of the problems the application ought to solve. Consequently the application also became a resource management tool for demand and wind speed input files.

Simultaneously we also had to reduce the architectural complexity in order to enhance our development speed. Doing so made it possible for us to fulfill all the requirements. The biggest architectural compromise was to switch the persistence method to be entirely file based. This generally reduces any read/write performance, but we hope this persistence method is sufficient during the early stages of the application. During manual testing of the application we have not encountered any performance issues, however these tests were not extensive enough to test the applications performance when the directories become large. The biggest disadvantages that we see from using a file base persistence method is losing out on the scalability of using Cassandra and losing the data integrity that an RDBMS offer.

Another architectural compromise we made to reduce complexity was not to utilize a messaging system to react to domain events. In order to preserve the boundaries between each module, we had to directly invoke the service layer of the downstream context from the resolver. We wanted to utilize a messaging system, however due time constraints we decided to mitigate having to implement this.

4.3 Development Process

Our development process did not go as we expected. Even though we wanted to be Agile, we cannot say we achieved this goal. The design phase of the development process took a little longer than desirable due to our lack of knowledge of the domain. Since we wanted to understand the domain and the requirements clearly we had many discussions with the domain experts, our supervisors, early in the development stage. However, they did not have a clear idea of exactly what they wanted whilst we wanted a clear understanding before making any architectural choices.

This led to weeks where we couldn't manage to produce any useful progression. Ultimately we decided to make architectural choices based on the knowledge we had at the time. After this point we got a better idea of what the desired goal was as it became easier for our domain experts to envision what the application would do. This led to us continuously making design choices throughout the development process.

However, because we didn't have a clear understanding of the different use cases and feature set when we started it became difficult to clearly define which tasks needed to be implemented in order to fulfill the requirements. Therefore we were not able to continuously integrate with our code in a successful manner, rather each integration often ended up introducing many code changes.

4.4 Improvements

Overall we assess the outcome of the application as successful. It is able to fulfill all the requirements set for it and we managed to successfully implement our architectural goals. This being said, there are many areas where the application could improve.

Firstly, and most importantly, the lack of automated tests decreases the application's maintainability significantly. Adding automated tests to the application brings more benefits as the application grows larger or becomes more complex.

Secondly, our domain model does not resemble the ubiquitous language properly. In our domain model we identify Configurations, Demands and WindSpeed as three supporting domains. However, a better domain model would be to unite these three subdomains into a single Resource Management subdomain. Within the Resource Management subdomain Configurations, Demands and WindSpeed would remain as aggregates however. In our opinion this would make the domain model resemble the domain more accurately, however we believe that the current domain model is sufficiently understandable. Additionally our current domain model offers more granularity and may be preferable as a result of that. Therefore, the current domain should be reassessed in order to figure out if it is sufficient or needs to be changed.

As previously mentioned we do not employ any messaging system. This is something that may be necessary to further establish the modularity of the application, however this introduces a lot of complexity and potentially costs. Because of that it is important to carefully evaluate the necessity of introducing a messaging system. If not implemented all developers have to be mindful of how changes made affect the modularity of the system.

A suggestion is to use a dependency injection framework to incorporate dependency injection and dependency inversion. These design patterns enhances the testability of the code and reduces tight coupling between components [42]. Due to time constraints, complexity and inexperience with Python web development we did not prioritize utilizing these design patterns.

Chapter 5

Conclusion

In conclusion, we have successfully developed a web application for configuring and displaying power system simulations, using the latest technologies to ensure optimal speed and performance. Despite the absence of a database, we were able to create an efficient and well-structured application that can be easily maintained and updated.

We understand that there is always room for improvement, and we acknowledge that the use of automated testing and a database would greatly enhance the application's functionality and ease of use. However, we are proud of what we have accomplished in the given timeframe, and we are confident that with further development, this application can become a valuable tool for the energy industry.

Through this project, we have gained a deeper understanding of the importance of writing clean, readable code that is easily maintainable and understandable by other developers. We have also demonstrated our ability to adapt to new technologies and overcome challenges, ultimately delivering a successful project.

Overall, we are proud of the work we have done and the skills we have developed throughout this project. We are excited to see where this application goes in the future.

Bibliography

[1] Cassandra applications. https://data-flair.training/blogs/cassandra-applications/. Accessed at: 11/05/2023.

[2] Design a ddd-oriented microservice. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice.

[3] Fastapi. https://fastapi.tiangolo.com/.

[4] Graphql. https://graphql.org/. Accessed at: 29/04/2023.

[5] Houdini or: How i learned to stop worrying and love graphql. https://houdinigraphql.com/.

[6] Introduction to the server side. https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction.

[7] It all starts with customer identity (ciam). https://auth0.com/.

[8] Json encoder and decoder. https://docs.python.org/3/library/json.html. Accessed at: 07/05/2023.

[9] Minimal css framework for semantic html. https://picocss.com/. Accessed at: 06/05/2023.

[10] Refresh token rotation. https://auth0.com/docs/secure/tokens/refresh-tokens/refresh-token-rotation.

[11] Strawberry is a new graphql library for python 3, inspired by dataclasses. https://strawberry.rocks/. Accessed at: 29/04/2023.

[12] Use containers to build, share and run your applications. https://www.docker.com/resources/what-container/. Accessed at: 01/05/2023.

[13] web development, streamlined. https://kit.svelte.dev/.

[14] What is terraform? https://developer.hashicorp.com/terraform/intro. Accessed at: 02/05/2023.

[15] The oauth 2.0 authorization framework. https://datatracker.ietf.org/doc/html/rfc6749, 2012.

[16] Openid connect core 1.0 incorporating errata set 1. https://openid.net/specs/openid-connect-core-1_0.html, 2014.

[17] Developing the ubiquitous language. https://thedomaindrivendesign.io/developing-the-ubiquitous-language/, 2019.

[18] What is a hyperscaler? https://www.redhat.com/en/topics/cloud/what-is-a-hyperscaler, 2022.

[19] Azure container apps overview. https://learn.microsoft.com/en-us/azure/container-apps/overview, 2023. Accessed at: 01/05/2023.

[20] This page is an overview of kubernetes. https://kubernetes.io/docs/concepts/overview/, 2023. Accessed at: 02/05/2023.

[21] Ian F Alexander and Ljerka Beus-Dukic. *Discovering requirements: how to specify products and services.* John Wiley & Sons, 2009.

[22] Steven A.Lowe. An introduction to event storming: The easy way to achieve domain-driven design. https://techbeacon.com/app-dev-testing/introduction-event-storming-easy-way-achieve-domain-driven-design.

[23] Stephen J. Bigelow. Microsoft azure. https://www.techtarget.com/searchcloudcomputing/definition/Windows-Azure.

[24] B. Boehm and V.R. Basili. Top 10 list [software development]. *Computer*, 34(1):135–137, 2001.

[25] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[26] Alana Brandão. Domain-Driven Desing: Context Mapping and Tactical Design, October 2021.

[27] Lan Cao and Balasubramaniam Ramesh. Agile requirements engineering practices: An empirical study. *IEEE Software*, 25(1):60–67, 2008.

[28] Armon Dadgar. Infrastructure as code: What is it? why is it important? `https://www.hashicorp.com/resources/what-is-infrastructure-as-code`, 2018.

[29] T. DeMarco and T. Lister. Risk management during requirements. *IEEE Software*, 20(5):99–101, 2003.

[30] Kate Eby. Agile methodologies: What is best for project management and software development? `https://www.smartsheet.com/content/agile-methodologies`, 2022.

[31] Ellen Gottesdiener and Mary Gorman. *Discover to deliver: agile product planning and analysis.* EBG Consulting, Incorporated, 2012.

[32] Eben Hewitt. *Cassandra: the definitive guide.* " O'Reilly Media, Inc.", 2010.

[33] Stanislav Kozlovski. A short overview of object oriented software design. `https://www.freecodecamp.org/news/a-short-overview-of-object-oriented-software-design-c7aa0a622c83/`, 2018.

[34] C. Larman and V.R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, 2003.

[35] James Montemagno. Domain events: Design and implementation. `https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-events-design-implementation`, 2022.

[36] Peter Onyisi, ATLAS Collaboration, et al. Event-driven messaging for offline data quality monitoring at atlas. In *Journal of Physics: Conference Series*, volume 664, page 062045. IOP Publishing, 2015.

[37] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, 1986.

[38] Sten Pittet. Continuous integration vs. delivery vs. deployment. `https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment`. Accessed at: 30/04/2023.

[39] Kosovare Sahatqija, Jaumin Ajdari, Xhemal Zenuni, Bujar Raufi, and Florije Ismaili. Comparison between relational and nosql databases, 2018.

[40] H. Sharp, A. Finkelstein, and G. Galal. Stakeholder identification in the requirements engineering process. In *Proceedings. Tenth International Workshop on Database and Expert Systems Applications. DEXA 99*, pages 387–391, 1999.

[41] Ian Sommerville. *Software Engineering.* 2016.

[42] Kelvin Tan. Overcoming the Tight Coupling Anti-Pattern in Swift Development, May 2023.

[43] William Ulrich. Introduction to architecture-driven modernization. In *Information Systems Transformation*, pages 3–34. Elsevier, 2010.

[44] Zhu Wei-ping, Li Ming-xin, and Chen Huan. Using mongodb to implement textbook management system instead of mysql, 2011.

[45] Karl Wiegers and Joy Beatty. *Software requirements.* Pearson Education, 2013.

Appendix A

Source Code

GitHub

Website