Software Engineering
< DLMCSPSE01 />

Project: HoppyBrew

Concept Phase

International University of Applyied Sciences

Written by
Asbjorn Bordoy

On the
18-Mar-2024

Revision 2.0.0

Abstract

The project aims to develop a comprehensive Beer Brewing Recipe Manager system, catering to brewing enthusiasts and homebrewers. This system facilitates the management of brewing processes and associated data through intuitive interfaces and robust functionalities. Users can create, share, and manage beer recipes, customize water and equipment profiles, schedule brewing sessions, monitor fermentation in real-time, generate reports, and more. The system ensures a seamless user experience by integrating with external devices like ISpindel for data collection and leveraging a database for secure storage and retrieval of brewing-related information. With an emphasis on user-friendly design and versatile features, the Beer Brewing Recipe Manager fosters innovation and tradition in the art of homebrewing.

Contents

List of Tables

List of Figures

Revision History

Table 1: Revision History

Revision	Date	Description	Author
1.0.0	02-Apr-2024	Preliminary Release in Concept Phase written in R-Markdown	Asbjorn Bordoy
2.0.0	15-Apr-2024	Complete overhaul of the document to include all phases of the project and written in Markdown	Asbjorn Bordoy

1 # Introduction and Goals

This documentation: is intended to provide a high-level overview of the HoppyBrew application. The document is based on the template provided by Hruschka and Starke (n.d.). Arc42 is a template for documenting software architectures. It is based on the ISO/IEC/IEEE 42010 standard, which is the international standard for documenting software architectures. The template is designed to be flexible and adaptable, and to be used in a wide range of software development projects. The template is divided into a number of sections, each of which covers a different aspect of the software architecture. The sections are designed to be used in a modular fashion, so that they can be used individually or in combination with other sections. The template is also designed to be easy to use, with a clear and consistent structure, and with a focus on the most important aspects of software architecture.

1.1 Requirements Overview

1.1.1 What is HoppyBrew?

HoppyBrew is a web application designed to help beer brewing enthusiasts manage their brewing recipes and brew logs. The application is intended to provide a user-friendly and intuitive interface for managing brewing recipes and brew logs. The application is designed to be compatible with a wide range of devices and browsers, and to integrate with other brewing tools and services, such as `iSpindel`. The application is targeted at beer brewing enthusiasts who want to manage their brewing recipes and brew logs in a simple and efficient way without the need for overpriced subscription fees like those at "Brewfather" (n.d.), "Brewers Friend" (n.d.), or "Beersmith" (n.d.).

1.1.2 Why HoppyBrew?

The distinguishing feature of HoppyBrew compared to other brewing applications is that it is open-source and free to use. This means users can utilize the application without having to pay any subscription fees.

1.1.3 Main Features:

The main features of the application are as follows:

- Store and manage brewing recipes.
- Create and manage batches based on recipes.
- Define water profiles for brewing recipes.
- Manage brewing profiles, such as mash profiles, fermentation profiles, equipment profiles, and beer style profiles.
- Interface with external devices, such as iSpindel, to collect real-time data.
- Manage inventory items, such as fermentables, hops, yeast, and miscellaneous items.
- Import and export recipes based on the BeerXML standard.

1.1.4 Target Audience:

The target audience for the application includes a large group of beer brewing enthusiasts and self-hosting enthusiasts who want to manage their brewing recipes and brew logs in a simple and efficient way, without the need for overpriced subscription fees like those at "Brewfather" (n.d.), "Brewers Friend" (n.d.), or "Beersmith" (n.d.).

1.2 Quality Goals

The top three quality goals for the architecture and design, whose fulfillment is of highest importance to the major stakeholders of HoppyBrew, have been identified as follows:

Priority	Quality	Motivation
1	Usability	The application should be easy to use and intuitive, with a clean and modern user interface.
2	Compatibility	The application should be compatible with a wide range of devices and browsers (mobile, desktop, tablet).

Priority	Quality	Motivation
3	Integration	The application should integrate with other brewing tools and services, such as iSpindel.

Table: Quality goals and priorities for the application.

These quality goals are derived from the `ISO/IEC 25010 quality model` as referenced in Hruschka and Starke (n.d.). The quality requirements will be further detailed in section 10, "Quality Requirements," of this document.

1.3 Stakeholders

In the architecture and design process of HoppyBrew, stakeholders play a pivotal role, providing essential requirements and constraints. Given that this project is part of a school assignment, the stakeholders are limited to the following individuals and their expectations:

Table 3: Stakeholders and their expectations for the application.

Priority	Name/Category	Expectations
Primary	Beer-brewer Enthusiast	Wants a user-friendly and intuitive application for managing brewing recipes and brew logs.
Secondary	Self-hosting Enthusiast and devellopers	Wants a high-quality, open-source application that is easy to maintain and extend.
Tertiary	The Software Architect	Wants a well-documented and well-structured application that meets the requirements of the assignment. On sucessful completion of the project, the Architect will be engolfed in a sense of pride and accomplishment.

It's important to recognize that since this is just a school project, the stakeholders are restricted to my own different personas.

1.3.1 Functional Requirements

The functional requirements for the application are based on the use cases that have been identified for the application. The use cases are intended to provide a high-level overview of the functionality that the application should support. The use cases are based on the requirements provided by the stakeholders and are intended to guide the architecture and design process in a way that ensures that the application meets the expectations of the stakeholders.

It is important to note that the difference between a use case and a functional requirement is that a use case describes a specific interaction between a user and the system, while a functional requirement describes a specific function or feature that the system should support. And although the use cases are based on the requirements provided by the stakeholders, the functional requirements are based on the use cases. Which means that the functional requirements are taking a step further and are more detailed than the use cases.

The functional requirements for the application are as follows:

Table 4: Functional requirements for the application.

Id	Requirement	Description
FR1	**User Management**	The application should support CRUD operations for users.
FR2	**Recipe Management**	The application should support CRUD operations for recipes.

Id	Requirement	Description
FR3	**Batch Management**	The application should support CRUD operations for batches.
FR4	**{Profile Management}**	The application should support profile management
FR4.1	*Water Profile*	The application should support CRUD operations for water profiles.
FR4.2	*Mash Profile*	The application should support CRUD operations for mash profiles.
FR4.3	*Fermentation Profile*	The application should support CRUD operations for fermentation profiles.
FR4.4	*Equipment Profile*	The application should support CRUD operations for equipment profiles.
FR4.5	*Beer Style Profile*	The application should support CRUD operations for beer style profiles.
FR5	**Device Management**	The application should support CRUD operations for devices.
FR5.1	*iSpindel*	The application should support CRUD operations for iSpindel.
FR6	**Inventory Management**	The application should support CRUD operations for inventory items.
FR6.1	*Fermentables*	The application should support CRUD operations for fermentables.
FR6.2	*Hops*	The application should support CRUD operations for hops.
FR6.3	*Yeast*	The application should support CRUD operations for yeast.
FR6.4	*Miscellaneous*	The application should support CRUD operations for miscellaneous items.
FR7	**System Settings**	The application should support system settings.
FR8	**Recipe Library**	The application should be able to import and export recipes based on the BeerXML standard. ("BeerXML," n.d.)
FR9	**Realtime Data Collection**	The application should be able to collect realtime data from iSpindel.

Note!

- Requirements indicated with `{}` are abstract requirements that are further detailed in their respective sub-requirements.
- CRUD stands for Create, Read, Update, and Delete.

1.3.2 Non-functional Requirements

The non-functional requirements for the application are based on the quality goals that have been identified for the application. The quality goals are intended to provide a high-level overview of the most important quality attributes for the application, and to guide the architecture and design process in a way that ensures that these quality attributes are met. Unlike the functional requirements, the non-functional requirements not always so easily measurable, and are often more subjective in nature.

The non-functional requirements for the application are as follows:

Table 5: Non-functional requirements for the application.

Id	Requirement	Description
NFR1	**Usability**	The application should be easy to use and intuitive, with a clean and modern user interface.

Id	Requirement	Description
NFR2	**Compatibility**	The application should be compatible with a wide range of devices and browsers.
NFR3	**Integration**	The application should integrate with other brewing tools and services, such as iSpindel.
NFR4	**Performance**	The application should be fast and responsive, with minimal latency.
NFR5	**Scalability**	The application should be able to handle a large number of users and data.
NFR6	**Reliability**	The application should be reliable and available, with minimal downtime.
NFR7	**Security**	The application should be secure, with user authentication and authorization.
NFR8	**Maintainability**	The application should be easy to maintain and extend, with clean and modular code.
NFR9	**Documentation**	The application should be well-documented, with clear and concise documentation.
NFR10	**Open-source**	The application should be open-source, with a permissive license.

Note!

- The non-functional requirements are based on the quality goals that have been identified for the application.
- The non-functional requirements are intended to provide a high-level overview of the most important quality attributes for the application.
- The non-functional requirements are intended to guide the architecture and design process in a way that ensures that these quality attributes are met.

2 Architecture Constraints

2.1 Technical Constraints

Table 6: Technical constraints for the application.

Id	Constraint	Background and / or Motivation
TC1	**Technology Stack**	The application will be developed using a specific technology stack, including Vue.js, Python, FastAPI, PostgreSQL, Docker, and Docker Compose. This is to ensure consistency and compatibility across different components of the application.
TC2	**Open-source**	The application will be open-source, with a permissive license. This is to ensure that the application is freely available and can be used and modified by anyone.
TC3	**Compatibility**	The application should be compatible with a wide range of devices and browsers. However, in this project, we will focus on developing the application for desktop browsers.
TC4	**Integration**	The application should integrate with other brewing tools and services, such as iSpindel. However, in this project, we will focus on integrating with iSpindel only.
TC5	**Performance**	No aspects related to performance will be considered in this project.
TC6	**Scalability**	No aspects related to scalability will be considered in this project.
TC7	**Reliability**	No aspects related to reliability will be considered in this project.
TC8	**Security**	No aspects related to security will be considered in this project.
TC9	**Maintainability**	No aspects related to maintainability will be considered in this project.

2.2 Organizational Constraints

Table 7: Organizational constraints for the application.

Id	Constraint	Background and / or Motiqation
OC1	**Deadline**	As a school project, worth 5 credits, a time constraint is imposed on the project. The project is expected to take about 150 hours to complete.
OC2	**Feature Limitations**	As a school project, some parts of the application will be left out. For example, not every type of action for all the different parts of the app will be implemented. If one type of action can be done for one part, it will be assumed it can be done for all the others.
OC3	**User Interface Simplification**	To keep things simple, the user interface will be made as basic as possible while still showing how the app works. This means it won't look super fancy or work perfectly on every device, but it'll get the job done.

2.3 Conventions

The following conventions will be used in the documentation:

Table 8: Conventions for the application.

Id	Convention	Background and / or Motivation
C1	**Language**	The documentation will be written inNaming Conventions**
C2	**Architecture**	The documentation will follow the arc42 template. This is to ensure that the documentation is well-structured and consistent.
C3	**Version Control**	The application will use Git for version control. This is to ensure that the codebase is well-maintained and up-to-date.
C4	**Code Style**	The application will follow a consistent code style, based on the PEP 8 style guide for Python. This is to ensure that the code is clean and readable.

2.4 Dependencies

The application is dependent on the following external services and tools:

Table 9: Dependencies for the application.

Id	Dependency	Description
D1	**iSpindel**	The application is dependent on iSpindel for collecting realtime data from the brewing process.
D2	**Database**	The application is dependent on a database for storing and managing data.
D3	**GitHub**	The application is dependent on GitHub for version control and collaboration.
D4	**Docker**	The application is dependent on Docker for containerization and deployment.

2.5 Risks

There are a number of risks associated with this project, which could potentially impact the success of the project. The following risks have been identified for the application:

Table 10: Risks for the application.

Id	Risk	Description
R1	**Technical Risks**	There is a risk that the technologies used in the project are not suitable for the requirements of the application.
R2	**Time Risks**	There is a risk that the project will take longer than expected to complete, due to unforeseen circumstances.
R3	**Skill development Risks**	There is a risk that the required knowledge for the project cannot be acquired in a reasonable amount of time.

3 Project plan

For this 5-credit school project, we've allotted 150 hours, assuming 1 credit equals 30 hours. We're dividing it into five 40-hour sprints, using Atlassian Jira and a Kanban board for task management.

Due to disruptions, we're aligning progress with hours rather than fixed dates. Despite exceeding estimated time in the first two sprints, setting up Atlassian Jira will help us track time more accurately going forward. We'll add one more sprint to make up for lost time, bringing the total sprints to five.

3.1 Software Development Methodology

We're following an agile development process, using the Scrum framework with one-week sprints. Git is our version control system, hosted on GitLab for collaboration and code maintenance.

3.2 Timeline

The project consists of three major phases:

1. **Conception Phase:** Defining objectives, scope, methodology, and requirements. Lasts two weeks.
2. **Development Phase:** Building the application, testing, deployment, and documentation. Lasts six weeks.
3. **Conclusion Phase:** Evaluating the project, preparing the final report, and presentation. Lasts two weeks.

These phases represent major milestones, though not explicitly depicted in the timeline image below. The sprints align with these milestones.

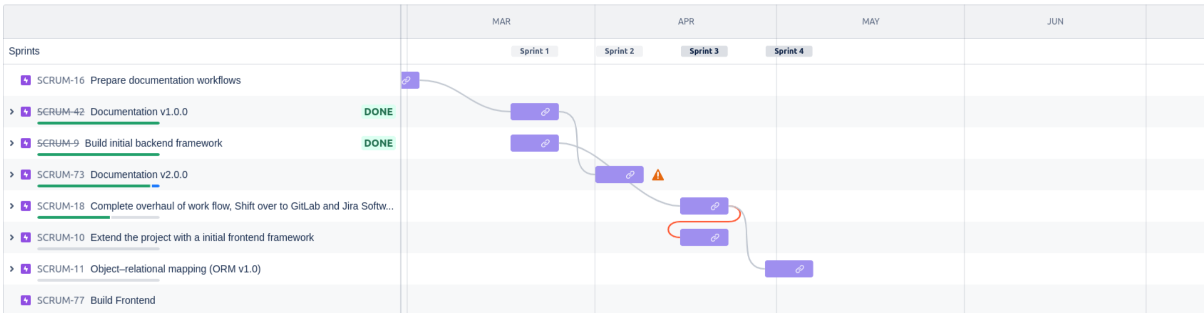


Figure 1: SCRUM Timeline

Table 11: Phases and sprints for the project. # System Scope and Context

Phase	Sprint
Conception Phase	Sprint 1, Sprint 2
Development Phase	Sprint 3, Sprint 4
Conclusion Phase	Sprint 5

3.3 Business Context

As indicated in the business context diagram below, the system only interacts with three external actors, namely the Administrator, The Brewer, and the ISpindel. The Administrator is responsible for managing the system, including adding new users, managing user roles, and monitoring the system. The Brewer is responsible for creating new brews, managing existing brews, and monitoring the progress of the brews. The ISpindel is responsible for collecting real-time data from the brewing process and sending it to the system.

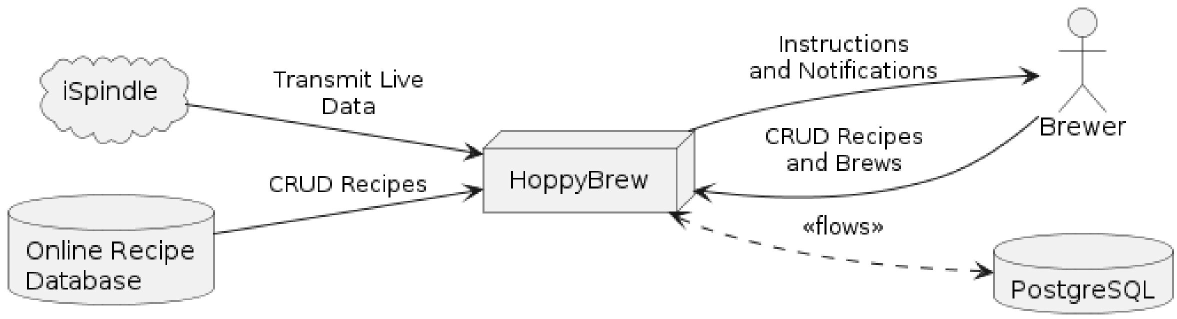


Figure 2: High-level business context diagram for the application.

Brewer: The Brewer is responsible for creating new brews, managing existing brews, and monitoring the progress of the brews. The Brewer interacts with the system to create new recipes, manage existing recipes, and monitor the progress of the brews.

Administrator: The Administrator is responsible for managing the system, including adding new users, managing user roles, and monitoring the system. The Administrator interacts with the system to add new users, manage user roles, and monitor the system.

iSpindel: The iSpindel is responsible for collecting real-time data from the brewing process and sending it to the system. The iSpindel interacts with the system to transmit live data from the brewing process.

Online Recipe Database: The Online Recipe Database is responsible for storing and managing recipes for the system. The Online Recipe Database interacts with the system to store and retrieve recipes.

HoppyBrew: HoppyBrew is the system that is being developed. It is responsible for managing brewing recipes and brew logs. HoppyBrew interacts with the Brewer, Administrator, iSpindel, and Online Recipe Database to provide the functionality of the system.

PostgreSQL: PostgreSQL is the database technology that is being used to store and manage data for the system. PostgreSQL interacts with HoppyBrew to store and retrieve data.

3.4 Use Case Diagram

The following use cases have been identified for the application:

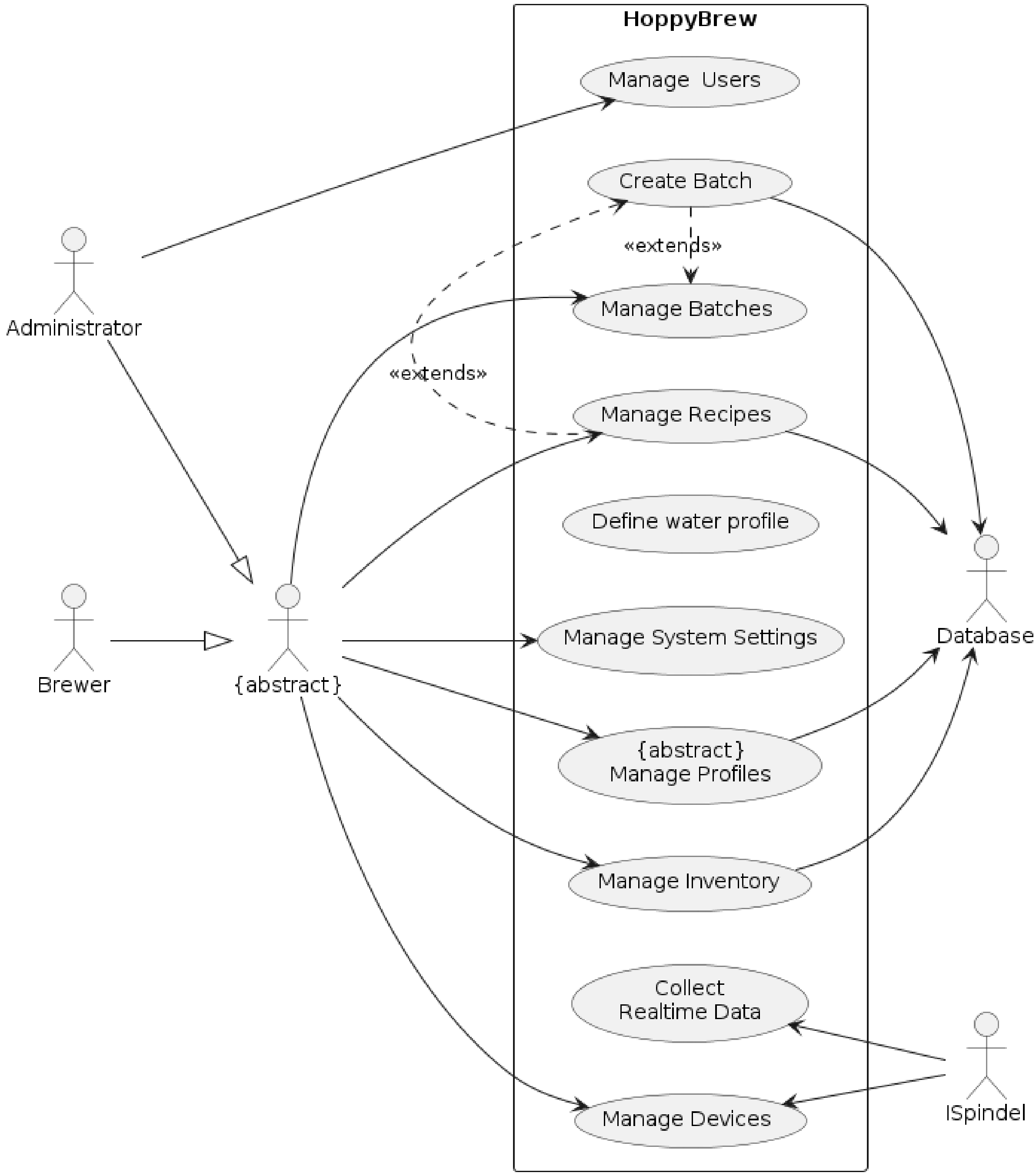


Figure 3: High-level use case diagram for the application.

3.4.0.1 Actors The following actors are involved in the use cases:

Table 12: Actors involved in the use cases for the application.

Actor	Description
Admin	The admin is responsible for managing the application and its users.
Brewer	The brewer is responsible for managing and creating brewing recipes and brews.
ISpindel	The iSpindel is responsible for providing data to the application.

Actor	Description
Database	The database is responsible for storing and managing data for the application.

3.4.0.2 Use Cases The following use cases are supported by the application:

Table 13: High-level use cases for the application.

Id	Use Case	Description
UC1	**Manage Users**	The admin can manage users, including creating, updating, and deleting users.
UC2	**Manage Recipes**	The brewer can manage recipes, including creating, updating, and deleting recipes.
UC3	**Create Batch**	The brewer can create a batch based on a recipe.
UC4	**Manage Batches**	The brewer can manage batches, including creating, updating, and deleting batches.
UC5	**Manage Profiles**	The brewer can manage profiles, including creating, updating, and deleting profiles.
UC6	**Manage Devices**	The admin can manage devices, including creating, updating, and deleting devices.
UC7	**Manage Inventory**	The admin can manage inventory, including creating, updating, and deleting inventory items.
UC8	**Manage System Settings**	The admin can manage system settings, including updating system settings.
UC9	**Collect Realtime Data**	The iSpindel can collect realtime data and send it to the application.

3.5 Technical Context

From a technical perspective, the system interacts with several external systems and services. The system relies on the iSpindel for collecting real-time data from the brewing process. Additionally, it depends on a database for storing and managing data. The system utilizes GitHub for version control and collaboration, and Docker for containerization and deployment.

The technical context diagram provides a high-level overview of the technical environment in which the system operates. It illustrates the system and its interactions with external systems and services, as well as the dependencies on these external systems and services.

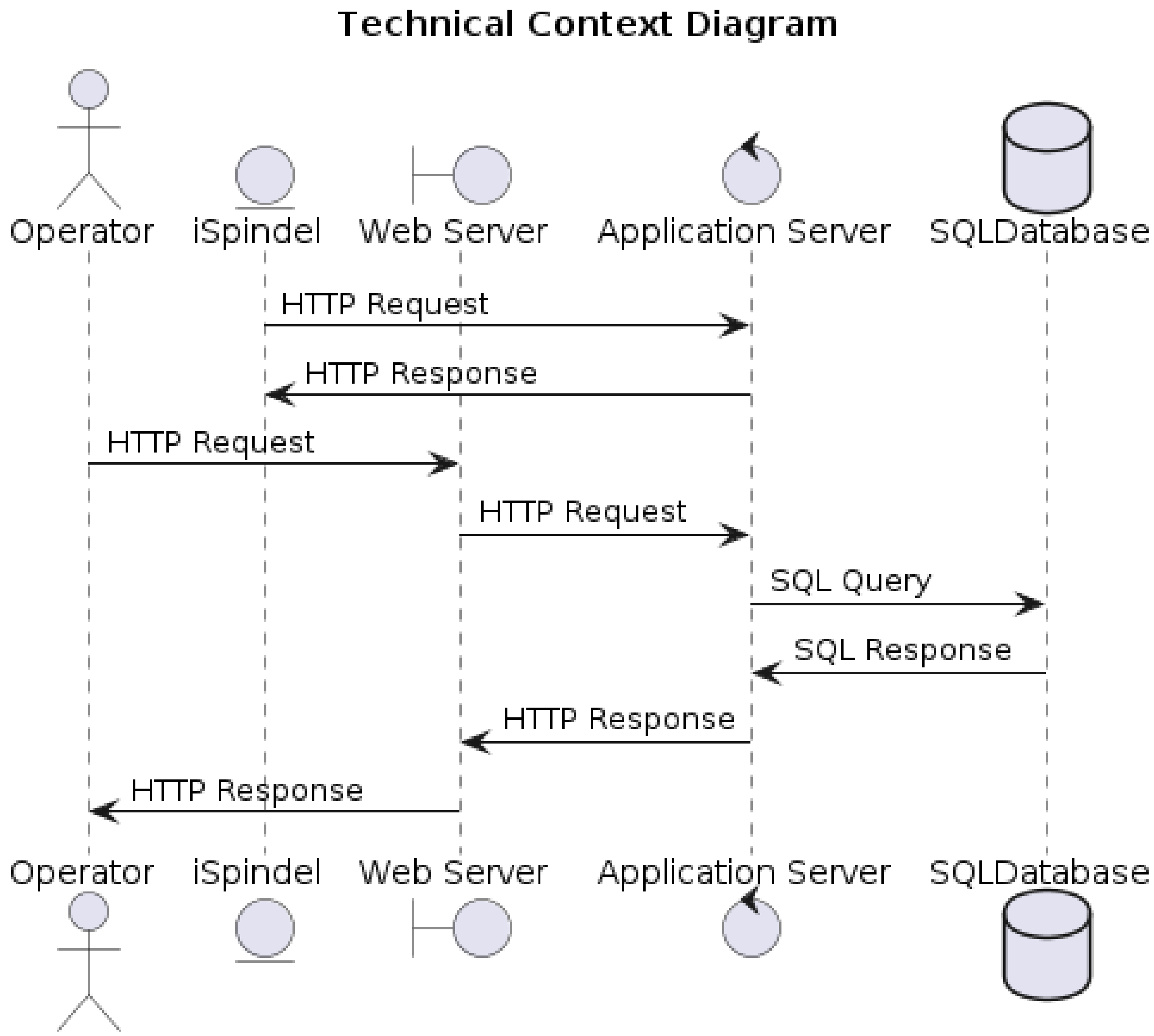


Figure 4: Technical-Context-Vew

Backend: The backend of the application is responsible for processing the data and storing it in the database. The backend communicates with the frontend using RESTful APIs. The backend is developed using Python and FastAPI, with PostgreSQL as the database technology.

Frontend: The frontend of the application is responsible for displaying the data to the user. The frontend communicates with the backend using RESTful APIs.

Database: The database of the application is responsible for storing and managing the data. The database communicates with the backend using SQL queries.

4 Solution Strategy

4.1 Technology Decisions

The application will be encapsulated in a containerized environment using Docker. The application will be developed using Vue.js as the frontend technology, Python and FastAPI as the backend technology, and PostgreSQL as the database technology. The application will be deployed using Docker and Docker Compose. The development of the application will be done using Visual Studio Code as the primary code editor.

4.1.1 Frontend Technology

After evaluating the requirements and constraints of the project, we have decided to use Vue.js as the frontend technology for the application. This is because Vue.js is a lightweight and flexible JavaScript framework that allows for rapid development of single-page applications. It provides a simple and intuitive syntax, making it easy to learn and use, which is ideal for a school project with a tight deadline. Additionally, Vue.js has a large and active community, which means there are plenty of resources and support available for developers. This will help us overcome any challenges we may face during the development process.

4.1.2 Backend Technology

The backend of the application will be developed using Python and FastAPI. FastAPI is a modern web framework for building APIs with Python. It is fast, easy to use, and provides automatic validation and serialization of request and response data. FastAPI is also based on standard Python type hints, which makes it easy to use and understand for developers who are familiar with Python. Additionally, FastAPI has built-in support for asynchronous programming, which allows for greater performance and scalability of the application. This makes it an ideal choice for building the backend of our application.

4.1.3 Database Technology

The application will use PostgreSQL as the database technology. PostgreSQL is a powerful, open-source relational database management system that is widely used in the industry. It provides a rich set of features, including support for complex queries, transactions, and data integrity constraints. PostgreSQL is also highly scalable and reliable, making it suitable for use in production environments. Additionally, PostgreSQL has excellent support for JSON data types, which will be useful for storing and querying the data in our application. Overall, PostgreSQL is a solid choice for the database technology of our application.

4.1.4 Deployment Technology

The application will be deployed using Docker and Docker Compose. Docker is a containerization platform that allows applications to be packaged into lightweight, portable containers that can run on any system. Docker Compose is a tool that allows multiple containers to be managed and orchestrated together. This will allow us to easily deploy and scale the application, as well as ensure consistency across different environments. Additionally, Docker and Docker Compose are widely used in the industry, which means there are plenty of resources and support available for developers. This will help us overcome any challenges we may face during the deployment process.

4.1.5 Development Tools

The development of the application will be done using Visual Studio Code as the primary code editor. Visual Studio Code is a lightweight and powerful code editor that provides a rich set of features, including syntax highlighting, code completion, and debugging tools. It also has a large and active community, which means there are plenty of extensions and support available for developers. This will help us write clean and efficient code, as well as overcome any challenges we may face during the development process.

4.2 Key Quality Goals

The key quality goals of the application are as follows:

1. Performance: The application should be fast and responsive, with low latency and high throughput.
2. Scalability: The application should be able to handle a large number of users and requests, and scale horizontally as needed.
3. Reliability: The application should be highly available and fault-tolerant, with minimal downtime and data loss.

To achieve these quality goals, we will use the following approaches:

1. Performance: We will use FastAPI for the backend, which is a high-performance web framework that is optimized for speed. We will also use PostgreSQL as the database technology, which is highly scalable and reliable. Additionally, we will use Docker and Docker Compose for deployment, which will allow us to easily scale the application as needed.
2. Reliability: We will use PostgreSQL as the database technology, which is highly reliable and provides support for transactions and data integrity constraints. We will also use Docker and Docker Compose for deployment, which will allow us to easily manage and orchestrate multiple containers. Additionally, we will implement automated testing and monitoring to ensure the reliability of the application.

4.3 Organizational Decisions

The application will be developed by a team of four developers, with each developer responsible for a specific component of the application. The team will follow an agile development process, with regular stand-up meetings, sprint planning, and retrospectives. The team will also use Git as the version control system, with a central repository hosted on GitHub. This will allow for easy collaboration and coordination between team members, as well as ensuring that the codebase is well-maintained and up-to-date.

4.4 Motivation

The motivation for the chosen technology decisions is as follows:

1. Frontend Technology: Vue.js was chosen as the frontend technology because of its lightweight and flexible nature, as well as its large and active community. This will help us develop the frontend of the application quickly and efficiently, with plenty of resources and support available.
2. Backend Technology: FastAPI was chosen as the backend technology because of its speed, ease of use, and support for asynchronous programming. This will help us develop the backend of the application quickly and efficiently, with high performance and scalability.
3. Database Technology: PostgreSQL was chosen as the database technology because of its power, scalability, and reliability. This will help us store and query the data in the application efficiently and reliably, with support for complex queries and transactions.
4. Deployment Technology: Docker and Docker Compose were chosen as the deployment technology because of their containerization capabilities and ease of use. This will help us deploy and scale the application easily and consistently, with support for managing and orchestrating multiple containers.
5. Development Tools: Visual Studio Code was chosen as the primary code editor because of its lightweight and powerful nature, as well as its rich set of features and extensions. This will help us write clean and efficient code, with plenty of resources and support available.

4.5 Solution Approaches

The solution approaches for the application are as follows:

1. Frontend: The frontend of the application will be developed using Vue.js, with a focus on simplicity, usability, and responsiveness. The frontend will communicate with the backend using RESTful APIs, with support for real-time updates and notifications.

2. Backend: The backend of the application will be developed using Python and FastAPI, with a focus on speed, performance, and scalability. The backend will provide a set of APIs for the frontend to interact with, with support for authentication, authorization, and validation.

3. Database: The database of the application will be developed using PostgreSQL, with a focus on reliability, scalability, and data integrity. The database will store the data in a structured and efficient manner, with support for complex queries and transactions.

4. Deployment: The application will be deployed using Docker and Docker Compose, with a focus on consistency, scalability, and reliability. The application will be packaged into lightweight and portable containers, which can run on any system. The containers will be managed and orchestrated using Docker Compose, which will allow for easy deployment and scaling of the application.

4.6 Additional Considerations

Some additional considerations for the application are as follows:

1. Security: The application will implement security best practices, such as encryption, authentication, and authorization. This will help protect the data and privacy of the users, as well as prevent unauthorized access and attacks.
2. Monitoring: The application will implement monitoring and logging, to track the performance and availability of the application. This will help identify and resolve any issues or bottlenecks, as well as ensure the reliability of the application.
3. Testing: The application will implement automated testing, to ensure the quality and correctness of the code. This will help identify and fix any bugs or issues, as well as ensure the stability of the application.
4. Documentation: The application will implement documentation, to provide guidance and support for developers and users. This will help explain the architecture and design of the application, as well as provide instructions for installation and usage.
5. Compliance: The application will comply with relevant laws and regulations, such as data protection and privacy laws. This will help protect the rights and interests of the users, as well as ensure the legal and ethical operation of the application.
6. Accessibility: The application will implement accessibility best practices, to ensure that all users can access and use the application. This will help provide a positive and inclusive experience for all users, regardless of their abilities or disabilities.
7. Localization: The application will implement localization, to support multiple languages and regions. This will help reach a wider audience and improve the usability of the application, as well as provide a more personalized experience for users.

5 Building Block View

5.1 Blackbox Overall System

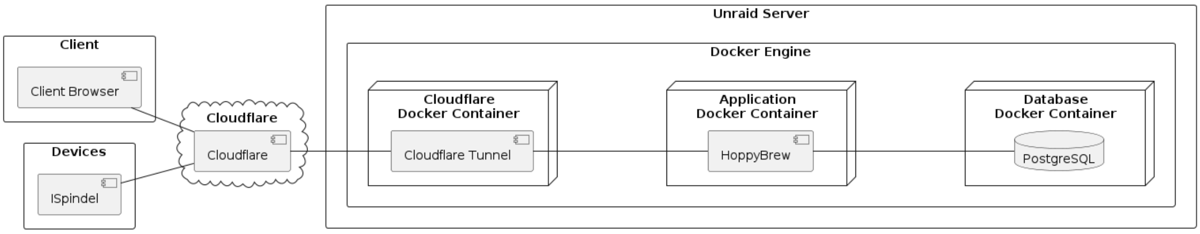


Figure 5: Overview Diagram

5.1.1 Client Browser

The client browser is responsible for displaying the data to the user. It communicates with the Cloudflare service to send and receive data.

5.1.2 Cloudflare

The Cloudflare service is responsible for routing the data between the client browser and the Unraid Server.

5.1.3 ISpindel

The ISpindel is responsible for collecting the data. It communicates with the Cloudflare service to send and receive data.

5.1.4 HoppyBrew

The HoppyBrew application is responsible for processing the data and storing it in the database. It communicates with the Cloudflare service to send and receive data. It also communicates with the PostgreSQL database to store the data. This can also be seen as the business logic of the application.

5.1.5 PostgreSQL

The PostgreSQL database is responsible for storing the data. It communicates with the HoppyBrew application to receive data.

5.2 Whitebox Overall System

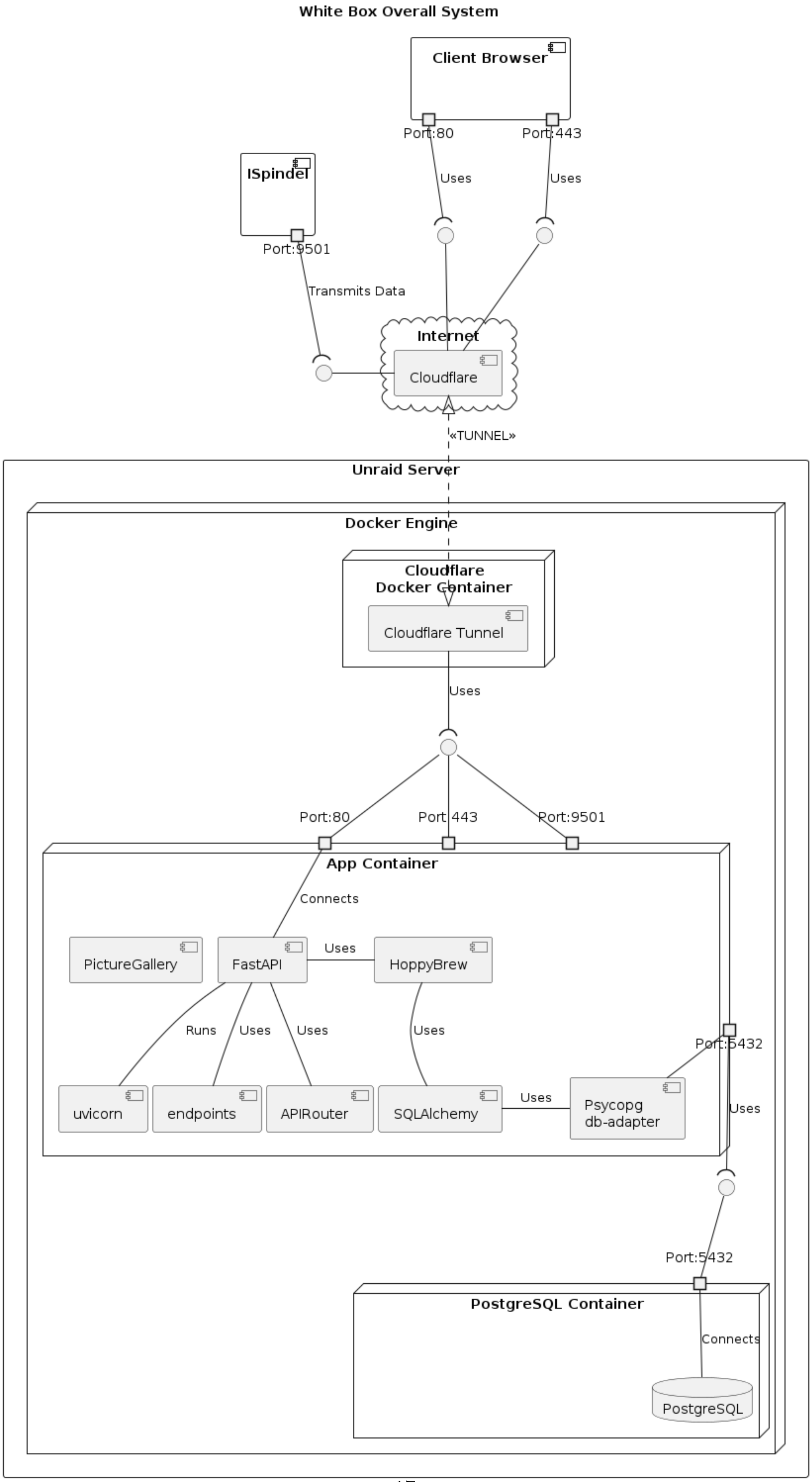


Figure 6: Overview Diagram

The motivation for the decomposition is to separate the concerns of the different parts of the system. The client browser is responsible for displaying the data to the user, the ISpindel is responsible for collecting the data, and the Unraid Server is responsible for processing the data and storing it in the database. The Cloudflare service is responsible for routing the data between the client browser and the Unraid Server.

Contained Building Blocks:

5.2.1 PictureGallery (Blackbox)

Intent/Responsibility:

The PictureGallery component is responsible for managing the pictures in the application.

Interfaces:

Table 14: PictureGallery interfaces.

Interface	**Description**
REST interface	/api/PictureGallery/*

5.2.2 FastAPI (Blackbox)

Intent/Responsibility:

The FastAPI component is responsible for providing the web framework for the application. It provides a set of APIs for the different parts of the application, such as users, recipes, batches, profiles, devices, inventory, and system settings. It communicates with the SQLAlchemy component to store and retrieve data.

5.2.3 APIRouter (Blackbox)

Intent/Responsibility:

The APIRouter component is responsible for routing the requests to the different parts of the application. It provides a set of routes for the different parts of the application, such as users, recipes, batches, profiles, devices, inventory, and system settings. It communicates with the FastAPI component to handle the requests and responses.

5.2.4 Endpoints (Blackbox)

Intent/Responsibility:

The Endpoints component is responsible for defining the endpoints for the different parts of the application. It provides a set of endpoints for the different parts of the application, such as users, recipes, batches, profiles, devices, inventory, and system settings. It communicates with the FastAPI component to handle the requests and responses.

5.2.5 SQLAlchemy (Blackbox)

Intent/Responsibility:

The SQLAlchemy component is responsible for providing the Object Relational Mapper (ORM) for the application. It provides a high-level interface for interacting with the PostgreSQL database, including support for complex queries, transactions, and data integrity constraints. It communicates with the FastAPI component to store and retrieve data.

5.2.6 Psycopg db-adapter (Blackbox)

Intent/Responsibility:

The Psycopg db-adapter component is responsible for providing the database adapter for the application. It provides a low-level interface for interacting with the PostgreSQL database, including support for connecting, querying, and updating data. It communicates with the SQLAlchemy component to store and retrieve data.

5.2.7 uvicorn (Blackbox)

Intent/Responsibility:

The uvicorn component is responsible for providing the ASGI server for the application. It provides a high-performance server for handling the requests and responses of the application. It communicates with the FastAPI component to run the application.

5.2.8 HoppyBrew (Blackbox)

Intent/Responsibility:

The HoppyBrew component is responsible for providing the business logic of the application. It provides a set of APIs for the different parts of the application, such as users, recipes, batches, profiles, devices, inventory, and system settings. It communicates with the SQLAlchemy component to store and retrieve data.

5.2.9 Cloudflare (Blackbox)

Intent/Responsibility:

The Cloudflare component is responsible for providing the routing service for the application. It provides a secure and reliable connection between the client browser and the Unraid Server. It communicates with the Cloudflare Tunnel component to route the data between the client browser and the Unraid Server.

5.2.10 Cloudflare Tunnel (Blackbox)

Intent/Responsibility:

The Cloudflare Tunnel component is responsible for providing the tunnel for the application. It provides a secure and reliable connection between the client browser and the Unraid Server. It communicates with the Cloudflare service to route the data between the client browser and the Unraid Server.

5.2.11 PostgreSQL (Blackbox)

Intent/Responsibility:

The PostgreSQL component is responsible for providing the database technology for the application. It provides a powerful and reliable relational database management system for storing and managing the data of the application. It communicates with the HoppyBrew component to store and retrieve data.

5.2.12 Client Browser (Blackbox)

Intent/Responsibility:

The Client Browser component is responsible for providing the user interface for the application. It provides a set of interfaces for the different parts of the application, such as users, recipes, batches, profiles, devices, inventory, and system settings. It communicates with the Cloudflare service to send and receive data.

5.2.13 ISpindel (Blackbox)

Intent/Responsibility:

The ISpindel component is responsible for providing the data collection service for the application. It provides a set of interfaces for collecting the data from the brewing process. It communicates with the Cloudflare service to send and receive data.

6 Runtime View

User interactions with the system are depicted in the following sequence diagrams.

For the sake of simplicity, we will not include all possible interactions in the sequence diagrams. Instead, we will focus on the core interactions that are essential for understanding the system. And we will not include all components in the sequence diagrams such as cloudflare.

6.1 CRUD Recipe

The sequence diagram illustrates CRUD operations for the Recipe entity, showcasing the standard pattern for all CRUD operations. It starts with the Brewer creating a new Recipe, which is then stored in the database. Subsequently, the Brewer can perform actions such as retrieving, updating, and deleting the Recipe as required. Additionally, there's an option to list all Recipes stored in the database.

The Brewer interacts with the system via the Client Browser, initiating communication with the Cloudflare Tunnel. This tunnel then communicates with the HoppyBrew application. Within the HoppyBrew application, communication with the PostgreSQL database takes place for storing and retrieving data.

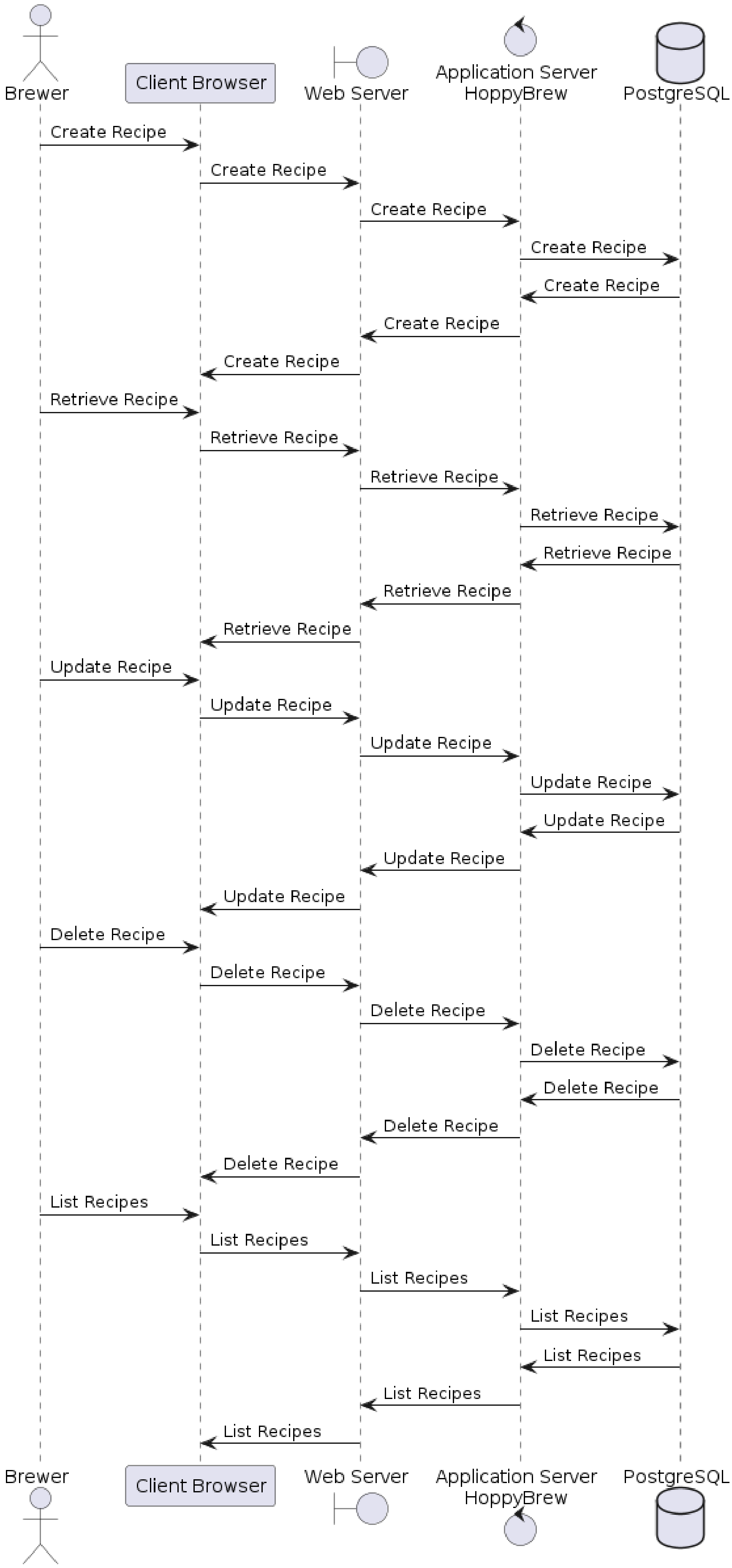


Figure 7: CRUD Recipe

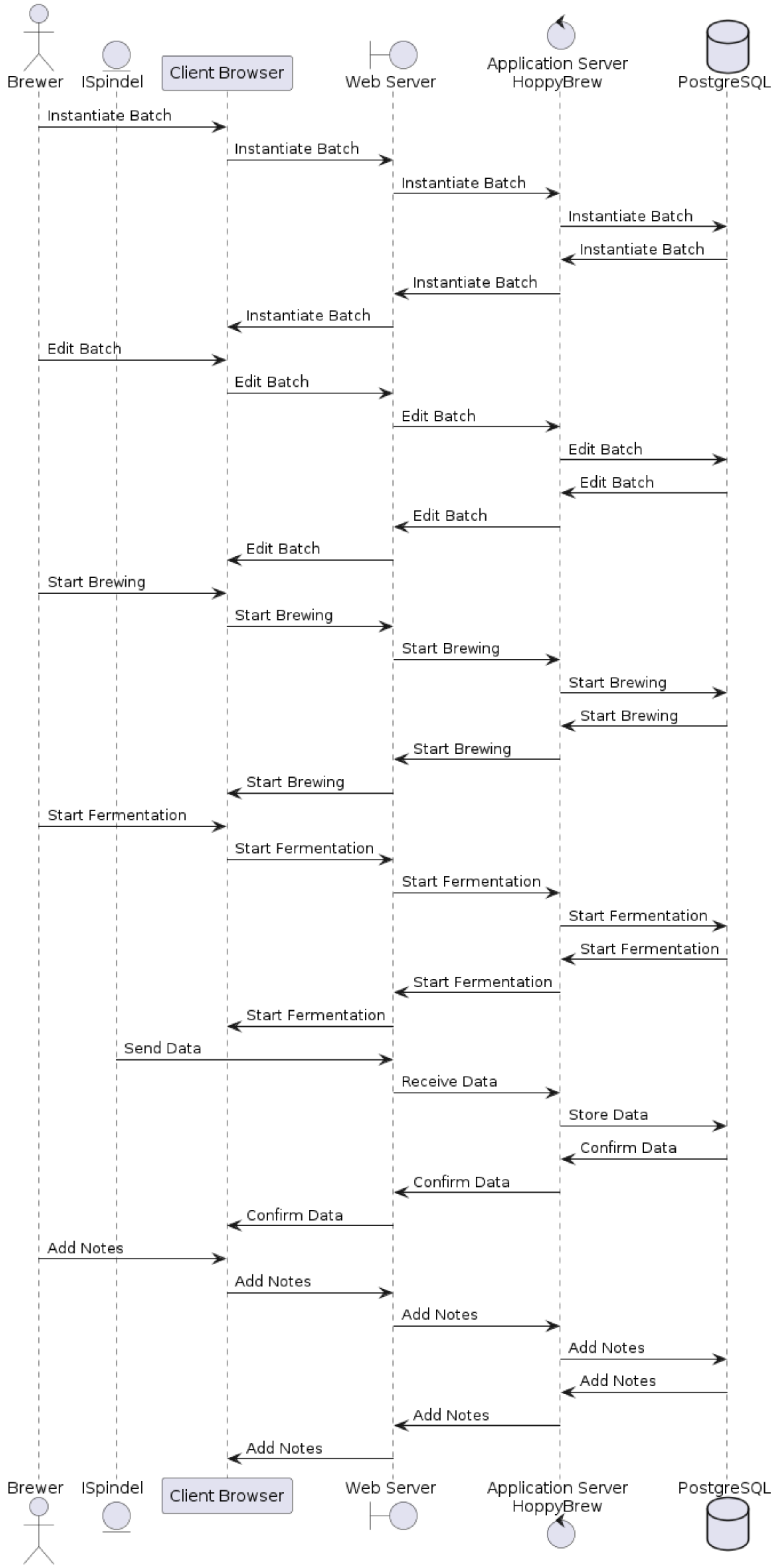
6.2 CRUD Batch

As we already have covered the CRUD operations for the Recipe entity, the CRUD operations for the Batch entity follow a similar pattern and we will therefore not repeat the explanation here.

A new Batch is instantiated by the Brewer from an existing Recipe. The Brewer can then edit the Batch, instance without affecting the original Recipe. The Brewer can also delete the Batch if it is no longer needed. The Brewer can also list all Batches stored in the database.

A batch is intended to represent a single brewing process, which is why the batch can undergo Three core brewing stages, which then are followed by a variety of additional stages that are optional like conditioning, archiving, etc.

1. **Preparation:** The Brewer prepares the ingredients and equipment for the brewing process. This is basically just the initial instantiation of the Recipe as a Batch.
2. **Brewing:** The Brewer starts the brewing process. In this stage, the Brewer is provided with all key information about the Recipe, These are mainly a good overview of the recipe. A countdown timer to indicate when the next ingredient should be added, Required temperatures, PH levels, and other key information. The Brewer can also add notes to the batch.
3. **Fermentation:** The Brewer transfers the wort to the fermenter and adds the yeast. The Brewer can monitor the fermentation process, including the temperature, gravity, and other key metrics. The Brewer can also add notes to the batch. It is at this stage that the ISpindel comes into play, as it collects real-time data from the fermentation process and sends it to the system. The Brewer can monitor this data in real-time. The Brewer can also add notes to the batch.



Deployment View

Key words: Docker, Docker Compose, PostgreSQL, Github, publish.yml, Dockerfile, Github Actions, Continuous Integration, Continuous Deployment, CI/CD, Docker Hub.

The deployment view describes how the application is deployed and managed in a production environment. The application is deployed using Docker and Docker Compose, with PostgreSQL as the database technology. The codebase is stored in a central repository on GitHub, with a Dockerfile for building the application image and a publish.yml file for deploying the application. The deployment process is automated using GitHub Actions, which provides continuous integration and continuous deployment (CI/CD) for the application. The application image is stored in Docker Hub, which provides a registry for storing and managing container images.

6.3 Deployment Diagram

The deployment diagram provides a high-level overview of the deployment architecture of the application. It shows the different components of the application, including the client browser, the ISpindel, the Cloudflare service, the Unraid Server, and the PostgreSQL database. It also shows the communication paths between the components, including the HTTP requests and responses that are sent and received. The deployment diagram helps to visualize how the application is deployed and managed in a production environment, and how the different components interact with each other.

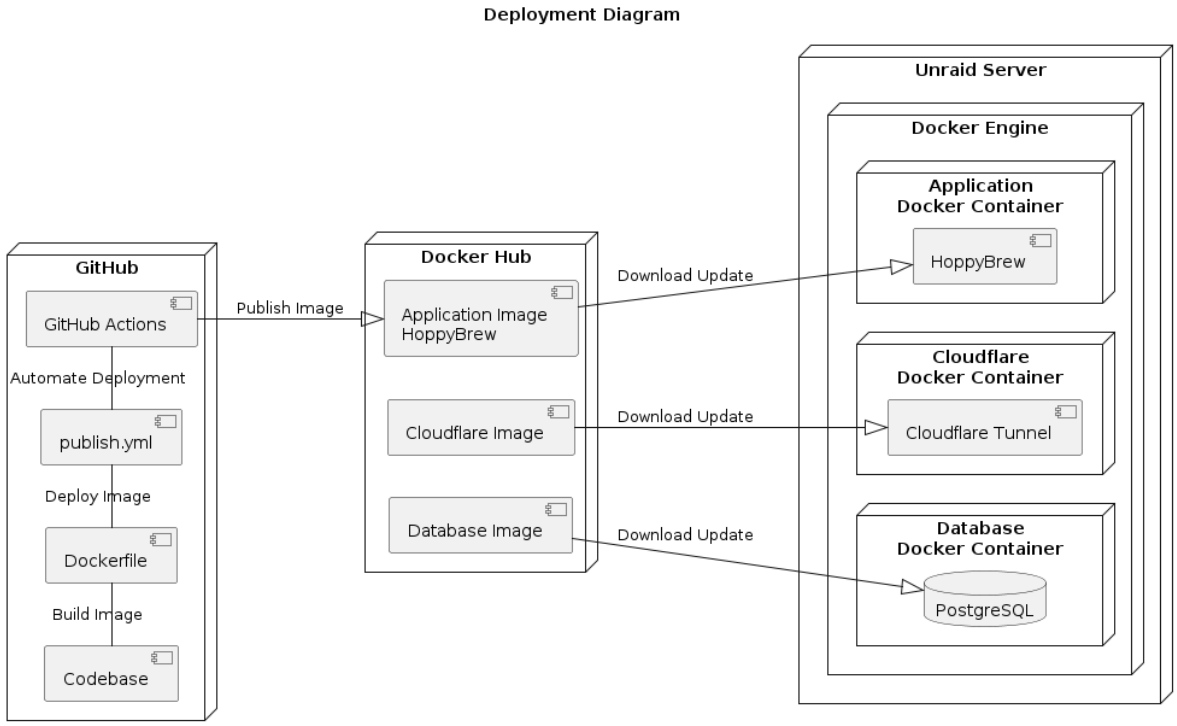


Figure 8: Deployment Diagram

7 Cross-cutting Concepts

HoppyBrew is a datacentric application, which is why everything revolves around the Entity-Relationship (ER-Diagram) model. The ER-Diagram model is a conceptual representation of the data in the application, showing the entities and their relationships. The ER-Diagram model helps to visualize the structure of the data and the relationships between the entities, which is essential for designing and implementing the database schema. The ER-Diagram model also helps to ensure the integrity and consistency of the data, as well as providing a clear and concise representation of the data model.

7.1 Entity-Relationship Diagram

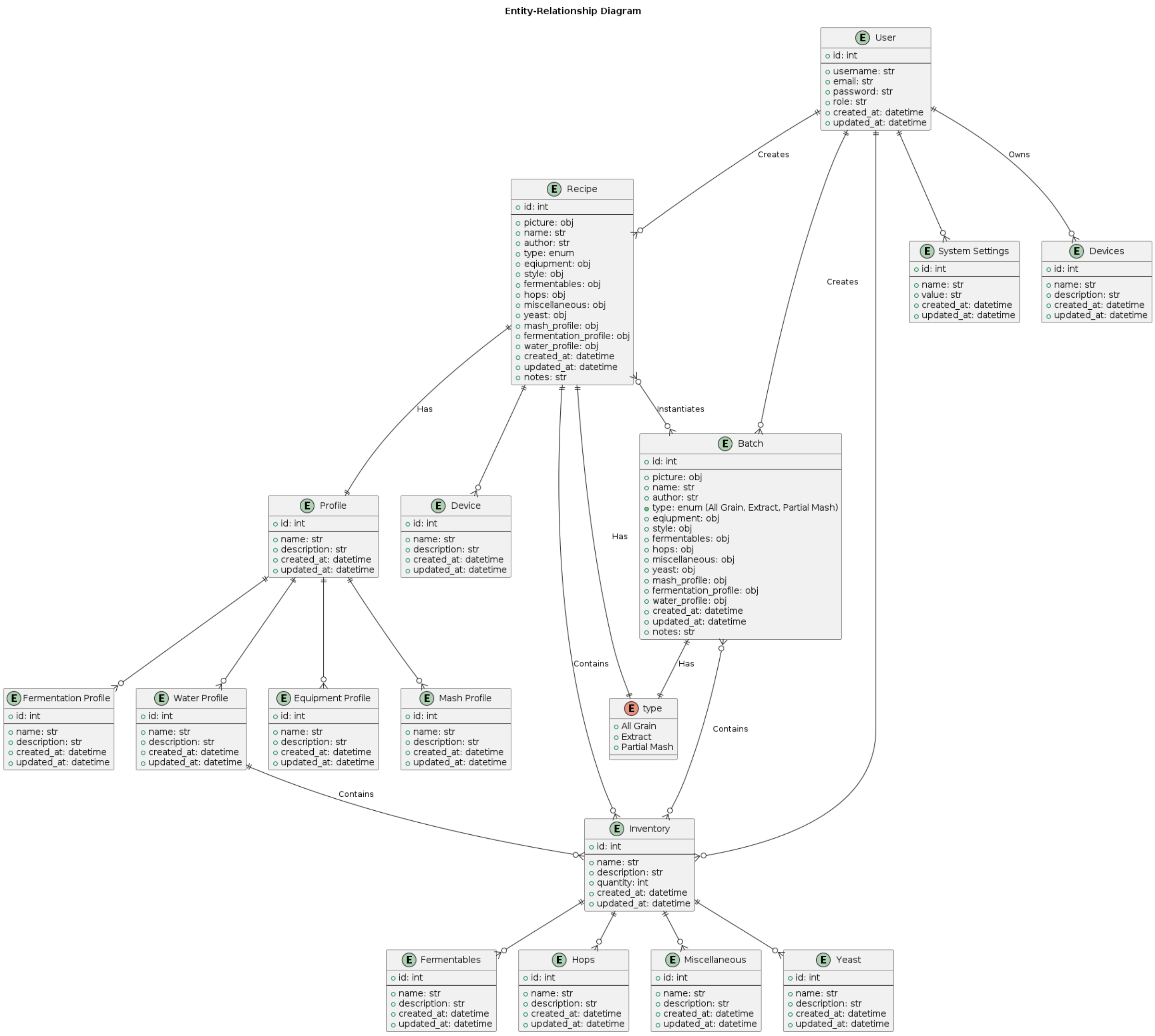


Figure 9: ER-Diagram

8 Architecture Decisions

8.1 Model-View-Controller (MVC) Architecture

We have decided to use the Model-View-Controller (MVC) architecture for the application. The MVC architecture is a software design pattern that separates the application into three main components: the model, the view, and the controller. The model represents the data and business logic of the application, the view represents the user interface, and the controller acts as an intermediary between the model and the view. This separation of concerns helps to improve the maintainability, scalability, and testability of the application. It also allows for easier collaboration between developers, as each component can be developed independently.

8.2 RESTful APIs

We have decided to use RESTful APIs for communication between the frontend and backend of the application. RESTful APIs are a set of architectural constraints that define how web services should be designed and implemented. They are based on the principles of statelessness, uniform interface, and resource-based architecture. RESTful APIs use standard HTTP methods such as GET, POST, PUT, and DELETE to perform operations on resources, and use standard data formats such as JSON or XML to represent data. This makes it easy to develop, test, and maintain the APIs, as well as allowing for interoperability with other systems and services.

8.3 sqlalchemy

We have decided to use sqlalchemy as the Object Relational Mapper (ORM) for the application. sqlalchemy is a powerful and flexible ORM that provides a high-level interface for interacting with databases. It allows developers to work with database objects as Python objects, and provides support for complex queries, transactions, and data integrity constraints. sqlalchemy also provides support for multiple database backends, including PostgreSQL, MySQL, and SQLite, which makes it suitable for a wide range of applications. This will help us interact with the PostgreSQL database in a more efficient and reliable manner, as well as providing a more object-oriented approach to working with data.

8.4 Single Page Application (SPA)

We have decided to use a Single Page Application (SPA) for the frontend of the application. An SPA is a web application that loads a single HTML page and dynamically updates the content as the user interacts with the application. This provides a more responsive and interactive user experience, as the page does not need to be reloaded every time the user performs an action. SPAs are also easier to develop and maintain, as they use a modular and component-based architecture. This allows for better code organization, reusability, and testability, as well as making it easier to scale and extend the application.

8.5 Containerization

We have decided to use containerization for the deployment of the application. Containerization is a lightweight and portable technology that allows applications to be packaged into self-contained units called containers. Containers include everything needed to run the application, including the code, runtime, libraries, and dependencies. This makes it easy to deploy and scale the application, as containers can run on any system that supports containerization. Containerization also provides isolation and security for the application, as each container runs in its own environment and does not interfere with other containers. This helps to ensure consistency and reliability across different environments, as well as making it easier to manage and update the application.

8.6 Continuous Integration and Continuous Deployment (CI/CD)

We have decided to use Continuous Integration and Continuous Deployment (CI/CD) for the development and deployment of the application. CI/CD is a set of practices and tools that automate the process of building, testing, and deploying software. Continuous Integration involves automatically building and testing the code whenever changes are made, to ensure that the code is working correctly. Continuous

Deployment involves automatically deploying the code to production whenever changes are made, to ensure that the code is available to users. This helps to improve the quality, speed, and reliability of the development process, as well as reducing the risk of errors and downtime. It also allows for faster feedback and iteration, as changes can be deployed quickly and easily.

9 Quality Requirements

This section contains all quality requirements in the form of a quality tree with scenarios. The most important ones have already been described in section 1.2 (quality goals). Here, we capture additional quality requirements with lesser priority, which will not create high risks when they are not fully achieved. Since quality requirements significantly influence architectural decisions, it's essential to understand what is truly important to every stakeholder, both concretely and measurably.

The quality tree, as defined in the ATAM (Architecture Tradeoff Analysis Method), consists of quality/evaluation scenarios as leaves. The tree structure with priorities provides an overview for a potentially large number of quality requirements. The quality tree is a high-level overview of the quality goals and requirements, structured as a tree-like refinement of the term "quality," with "quality" or "usefulness" as the root and quality categories as main branches. In any case, the tree should include links to the scenarios of the following section.

9.1 Quality Tree

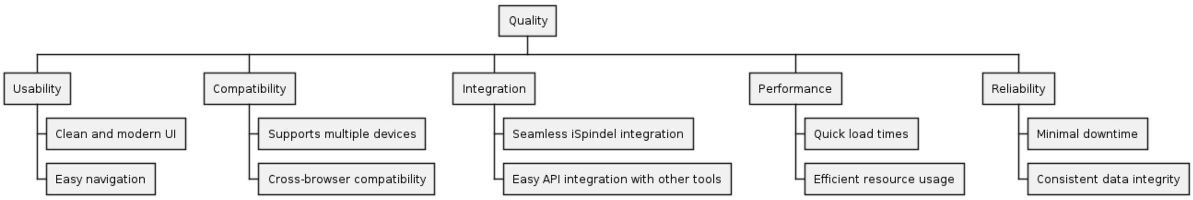


Figure 10: Quality Tree

9.2 Quality Scenarios

To make quality requirements concrete, we use scenarios that describe what should happen when a stimulus arrives at the system. For architects, two kinds of scenarios are important:

- Usage scenarios describe the system's runtime reaction to a certain stimulus. This includes scenarios that describe the system's efficiency or performance.
  - Example: The system reacts to a user's request within one second.
- Change scenarios describe a modification of the system or its immediate environment.
  - Example: Additional functionality is implemented or requirements for a quality attribute change.

Scenarios make quality requirements concrete and allow easier measurement or decision-making on whether they are fulfilled. Especially when assessing your architecture using methods like ATAM, you need to describe your quality goals more precisely down to a level of scenarios that can be discussed and evaluated. Scenarios can be documented in tabular or free-form text.

Table 15: Quality Scenarios.

ID	Description	Stimulus	Response Time	Priority
QS1	User submits a new brewing recipe	Form submission	<1 second	High
QS2	System integrates real-time data from iSpindel	Data arrival	Immediate	High
QS3	User navigates to the recipe library	Click on menu item	<1 second	Medium
QS4	Admin updates system settings	Form submission	<2 seconds	Medium
QS5	New API integration with external brewing tool	API call	<2 seconds	Low
QS6	System updates with new functionality (e.g., new recipe format support)	Deployment	Smooth transition	Medium

10 Risks and Technical Debts

A list of identified technical risks or technical debts is provided, ordered by priority. “Risk management is project management for grown-ups” (Tim Lister, Atlantic Systems Guild). This should be the motto for systematic detection and evaluation of risks and technical debts in the architecture, which will be needed by management stakeholders (e.g., project managers, product owners) as part of the overall risk analysis and measurement planning. The risks and technical debts are listed along with suggested measures to minimize, mitigate, or avoid risks or reduce technical debts.

Table 16: Risks and Technical Debts.

ID	Description	Priority	Suggested Measures
R1	Inadequate performance under high load	High	Implement load testing and optimize code efficiency
R2	Security vulnerabilities due to insufficient validation	High	Conduct regular security audits and implement robust validation checks
R3	Technical debt from rapid feature additions	Medium	Allocate regular refactoring sessions
R4	Integration issues with new brewing tools	Medium	Establish comprehensive integration testing protocols
R5	Delays in data updates from iSpindel	Low	Implement buffering and retry mechanisms

11 Glossary

This glossary contains the most important domain and technical terms that stakeholders use when discussing the system. Clearly defining terms ensures all stakeholders have an identical understanding of these terms and do not use synonyms or homonyms. The glossary includes terms with their definitions and potentially translations if needed.

11.1 Computer Science Terminologies

Table 17: Computer Science Glossary.

Term	Definition
Actor	In use case parlance, parties outside the system that interact with the system. They may be users or other systems.
Architecture	The high-level structure of software systems, identifying a set of components that collaborate to achieve system goals.
Conceptual Architecture	Directs attention to an appropriate decomposition of the system without delving into interface specification and type information.
Features	The differentiating functionality of a product, which may not be available in other products or with the same quality characteristics.
Functional Requirements	Capture the intended behavior of the system or what the system will do, expressed as services, tasks, or functions.
Logical Architecture	Detailed architecture specification, precisely defining component interfaces and connection mechanisms and protocols.
Meta-architecture	High-level decisions influencing the structure of the system, through style, patterns, principles, and philosophy, guiding structural choices.
Non-functional Requirements	Capture required properties of the system, such as performance, security, maintainability, etc.
Product Line	Consists of basically similar products with different cost/feature variations per product.
Product Family	Includes a number of product lines targeted at different markets or usage situations, with common elements of functionality and identity.
Qualities	System qualities or non-functional requirements, capturing properties of the system like performance, security, maintainability, etc.
Scenario	An instance of a use case, representing a single path through the use case, describing interactions between actors and the system.
Use Case	Defines a goal-oriented set of interactions between external actors and the system under consideration, detailing the steps to achieve a goal.
Use Case Diagram	A graphical representation of use cases and their relationships to the actors.
User Interface	The part of the system with which users interact, including screens, forms, reports, and so on.
User Story	A short, simple description of a feature told from the perspective of the person who desires the new capability, usually a user or customer.
View	A representation of a whole system from the perspective of a related set of concerns, used to describe the system from different stakeholders' viewpoints.
Viewpoint	A specification of the conventions for constructing and using a view, defining the kinds of models to be constructed and the rules governing them.
Viewtype	A template for a view, specifying the types of models to be constructed and the rules governing their construction.
Work Product	A document or model produced as part of a software development process, used to capture and communicate information about the system being developed.
Entity-Relationship Diagram (ERD)	A visual representation of the relationships between entities in a database, showing how data is organized and related.

Term	Definition
Object Relational Mapper (ORM)	A programming technique that maps objects from an object-oriented programming language to a relational database.
CRUD	Acronym for Create, Read, Update, Delete – basic operations for managing data.
Quality Attribute	A property or characteristic of the system that affects its overall performance and user experience, such as usability, reliability, or scalability.

11.2 Brewing Specific Terminologies

Table 18: Brewing Glossary.

Term	Definition
Brew Log	A record of the brewing process, including details such as ingredients, timings, and observations.
iSpindel	A digital hydrometer used in brewing to monitor the fermentation process by measuring tilt and temperature.
Recipe	A set of instructions for brewing beer, including ingredients and steps.
Stakeholder	An individual or group with an interest or concern in the project, such as users, developers, or project managers.
IBU	International Bitterness Units, a measure of the bitterness of beer, determined by the amount of hops used in brewing.
ABV	Alcohol by Volume, a measure of the alcohol content of beer, expressed as a percentage of the total volume.
OG	Original Gravity, a measure of the sugar content of beer before fermentation, used to calculate the alcohol content.
FG	Final Gravity, a measure of the sugar content of beer after fermentation, used to calculate the alcohol content.
EBC	European Brewery Convention, a measure of the color of beer, expressed in units of color.
BU/GU	Bitterness Units to Gravity Units ratio, a measure of the balance between bitterness and sweetness in beer.
Sparging	The process of rinsing the grains with hot water to extract sugars for fermentation.
Mashing	The process of soaking malted grains in hot water to extract sugars for fermentation.
Boiling	The process of heating the wort to a boil, adding hops and other ingredients, and sterilizing the liquid.
Fermentation	The process of adding yeast to the wort to convert sugars into alcohol and carbon dioxide.
Conditioning	The process of aging beer after fermentation to develop flavors and carbonation.
RO Water	Reverse Osmosis water, a type of purified water used in brewing to remove impurities and minerals.
Yeast Starter	A small batch of wort used to grow yeast cells before pitching them into the main batch of beer.
Cold Crash	The process of cooling beer to near-freezing temperatures to clarify the liquid before bottling or kegging.
Kegging	The process of transferring beer from a fermenter to a keg for carbonation and serving.
Bottling	The process of transferring beer from a fermenter to bottles for carbonation and storage.
Carbonation	The process of adding carbon dioxide to beer to create bubbles and effervescence.
Lautering	The process of separating the liquid wort from the solid grains after mashing.
Sparge Water	The hot water used to rinse the grains during the sparging process.
Mash Tun	The vessel used for mashing and lautering the grains in the brewing process.

Term	Definition
Fermenter	The vessel used for fermenting the wort after boiling and cooling.
Airlock	A device used to allow carbon dioxide to escape from the fermenter while preventing oxygen from entering.
Hydrometer	A device used to measure the specific gravity of beer, indicating the sugar content and fermentation progress.
Thermometer	A device used to measure the temperature of the wort during brewing and fermentation.
pH Meter	A device used to measure the acidity of the wort and beer during brewing and fermentation.
Hop Spider	A device used to contain hops during the boiling process, preventing them from clogging the equipment.
Immersion Chiller	A device used to cool the wort quickly after boiling, reducing the risk of contamination.

11.3 Conclusion

In conclusion, the development of HoppyBrew has been a challenging yet rewarding journey. Despite not implementing the iSpindel integration due to its complexity and the testing constraints, the project achieved several key milestones and functional implementations.

11.3.1 Achievements

1. **Database and Models:** Successfully created the database, including the models and CMS, which start up correctly. This setup involved ensuring smooth communication between the three Docker containers for the backend, database, and frontend.
2. **CRUD Operations:** Implemented and tested CRUD operations for hops, fermentables, miscellaneous items, yeast, recipes, and batches.
3. **Web Scraping:** Conducted web scraping to pull down various recipes, including those from the Brewer Association, and successfully stored this data in the database.
4. **Import and Export of XML Files:** Implemented functionalities to import and export BeerXML files, allowing seamless data exchange with applications like Brewfather.
5. **Continuous Integration and Deployment (CI/CD):** Set up CI/CD pipelines using GitHub Actions, successfully building and testing Docker containers, and ensuring interconnectedness between containers. This setup also included linting for code quality and partial implementation of deployment processes to Docker Hub.
6. **Documentation:** Developed comprehensive documentation using Markdown, converted to PDF using Pandoc. Despite the technical challenges, this documentation ensures clarity and ease of understanding for future developers.
7. **Makefile:** Worked with makefiles to streamline and automate the build process, which proved to be a powerful tool for managing project build tasks efficiently.

11.3.2 Challenges

1. **Dependency and Version Control:** Faced significant issues with managing dependencies and version control due to discrepancies between different tutorials and updates, leading to frequent application breaks and fixes.
2. **Database Migration:** Encountered extensive challenges with database relationships and migrations, which would have benefited from earlier use of database versioning tools.
3. **Front-end Development:** Although new to Vue.js and Nuxt.js, managed to develop the frontend, despite occasional breaks and the need to frequently rebuild the virtual environment.
4. **Complex Implementations:** Left out complex features such as equipment profiles, water profiles, and mash profiles due to time constraints and their similarity to recipe CRUD operations.
5. **Docker and Docker Compose:** Spent considerable time working with Docker and Docker Compose, realizing their differences. Docker Compose, while useful for local development, was not recommended for the Unraid system, leading to additional complexity in deployment strategies.

11.3.3 Learnings

1. **Scrum Methodology:** Gained a deep appreciation for Scrum, finding its dynamic and agile approach highly effective.
2. **Tools and Technologies:** Learned to use Atlassian Jira for project management, Docker for containerization, FastAPI for backend development, and Vue.js/Nuxt.js for frontend development.
3. **Documentation Practices:** Invested time in building thorough documentation, learning the intricacies of Markdown and conversion tools like Pandoc.
4. **GitHub vs. GitLab:** Initially explored using GitLab for its strong CI/CD capabilities but realized late in the project that the assignment required using GitHub. Upon reflection, found GitHub to be stronger and easier to work with, facilitating smoother project management and CI/CD implementation.
5. **Makefile:** Utilized makefiles to automate build tasks, enhancing the efficiency and manageability of the project's build process.

Overall, the HoppyBrew project provided a comprehensive learning experience, equipping me with valuable skills and knowledge in modern web application development and project management methodologies. Despite some unfinished features and the decision to deprioritize certain functionalities, the project stands as a robust proof of concept, ready for further development and refinement.

Bibliography

"Beersmith." n.d. Beersmith. https://beersmith.com/.

"BeerXML." n.d. BeerXML. http://www.beerxml.com/.

"Brewers Friend." n.d. Brewer's Friend. https://www.brewersfriend.com/.

"Brewfather." n.d. Brewfather. https://web.brewfather.app/.

Hruschka, Dr. Peter, and Dr. Gernot Starke. n.d. "Arc42." arc42. https://arc42.org/.