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 @arc42. 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.

Requirements Overview

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, @BrewersFriend, or @BeerSmith.

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.

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.

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, @BrewersFriend, or @BeerSmith.

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).
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 @arc42. The quality requirements will be further detailed in section 10, "Quality Requirements," of this document.

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:

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.

Table: Stakeholders and their expectations for the application.

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

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:

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.
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.

Id	Requirement	Description
FR8	**Recipe Library**	The application should be able to import and export recipes based on the BeerXML standard. [@beerxml]
FR9	**Realtime Data Collection**	The application should be able to collect realtime data from iSpindel.

Table: Functional requirements for the application.

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

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:

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

Table: Non-functional requirements for the application.

> **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.

\clearpage

Architecture Constraints

Technical Constraints

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.

Table: Technical constraints for the application.

Organizational Constraints

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.

Table: Organizational constraints for the application.

Conventions

The following conventions will be used in the documentation:

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.

Table: Conventions for the application.

Dependencies

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

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.

Table: Dependencies for the application.

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:

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.

Table: Risks for the application.

\clearpage

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.

Software Development Methodology

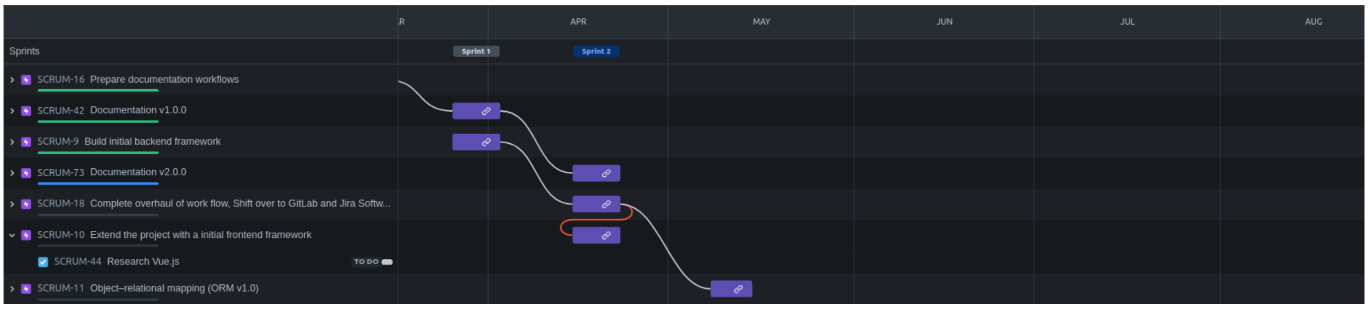
We're following an agile development process, using the Scrum framework with one-week sprints. Git is our version control system, hosted on GitLab for collaboration and code maintenance.

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.



SCRUM-16 Prepare documentation workflows
SCRUM-42 Documentation v1.0.0
SCRUM-9 Build initial backend framework
SCRUM-73 Documentation v2.0.0
SCRUM-18 Complete overhaul of work flow, Shift over to GitLab and Jira Softw...
SCRUM-10 Extend the project with a initial frontend framework
SCRUM-44 Research Vue.js
SCRUM-11 Object–relational mapping (ORM v1.0)

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

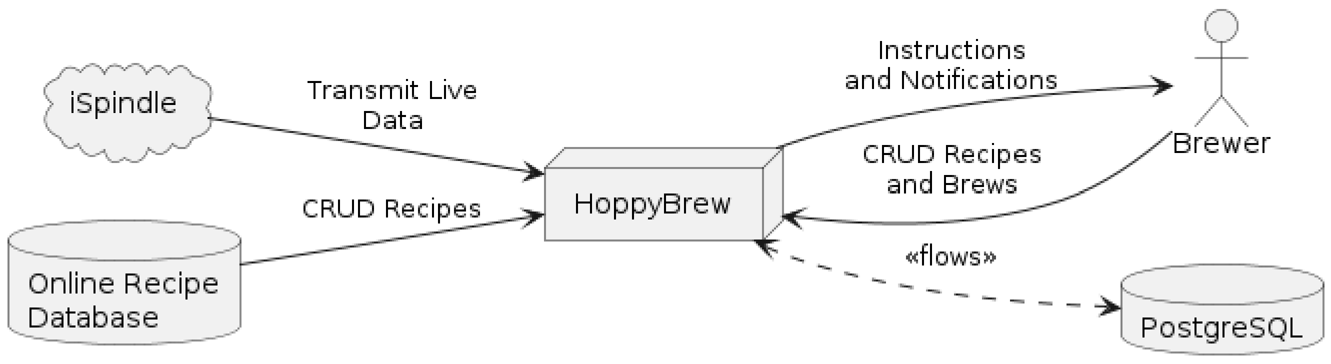
Table: Phases and sprints for the project.

System Scope and Context

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.

```
@startuml 02-Context-Vew-Business

left to right direction

cloud iSpindle
actor Brewer
database "Online Recipe\nDatabase" as RecipeDB

node "HoppyBrew" as HoppyBrew
database "PostgreSQL" as PostgreSQL

RecipeDB --> HoppyBrew : CRUD Recipes
iSpindle --> HoppyBrew : Transmit Live\nData
Brewer --> HoppyBrew : CRUD Recipes\nand Brews
HoppyBrew <..> PostgreSQL : <>
HoppyBrew --> Brewer : Instructions\nand Notifications
@enduml
```



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.

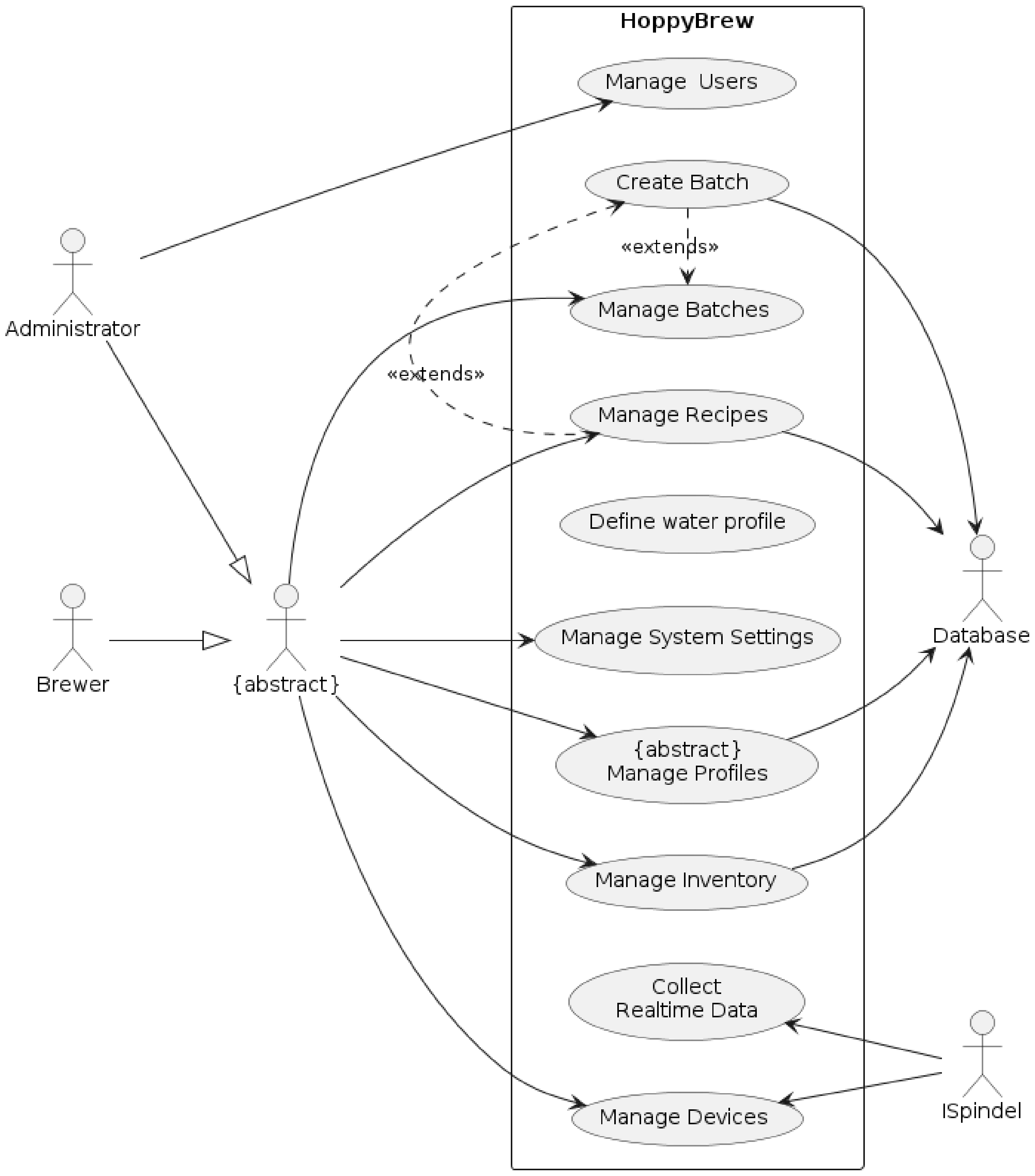
HoppyBrew: HoppyBrew is the system that is being developed. It is responsible for managing brewing recipes and brew logs. HoppyBrew interacts with the Brewer, Administrator, iSpindel, and Online Recipe Database to provide the functionality of the system.

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.

Use Case Diagram

The following use cases have been identified for the application:

```
@startuml 01-Use-Case-Diagram

left to right direction

actor Administrator as Admin
actor Brewer as Brewer
actor Database as DB
actor ISpindel as ISpindel
actor "{abstract}" as AbstractUser

Admin --|> AbstractUser
Brewer --|> AbstractUser

rectangle "HoppyBrew" as HoppyBrew {
    usecase "Manage  Users" as ManageUsers
    usecase "Manage Recipes" as ManageRecipes
    usecase "Define water profile" as DefineWaterProfile
    usecase "Create Batch" as CreateBatch
    usecase "Manage Batches" as ManageBatches
    usecase "{abstract}\nManage Profiles" as ManageProfiles
    usecase "Manage Devices" as ManageDevices
    usecase "Manage Inventory" as ManageInventory
    usecase "Manage System Settings" as ManageSystemSettings
    usecase "Collect\nRealtime Data" as CollectRealtimeData

    Admin --> ManageUsers
    AbstractUser --> ManageRecipes
    AbstractUser --> ManageBatches
    AbstractUser --> ManageProfiles
    AbstractUser --> ManageDevices
    AbstractUser --> ManageInventory
    AbstractUser --> ManageSystemSettings

    CreateBatch .> ManageBatches : <>

    ManageRecipes .> CreateBatch : <>
    CreateBatch --> DB

}

    ManageInventory --> DB
    ManageRecipes --> DB
    ManageProfiles --> DB

    ManageDevices <-- ispindel collectrealtimedata <-- @enduml < code>
```



Actors

The following actors are involved in the use cases:

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.

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

Use Cases

The following use cases are supported by 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.

Table: High-level use cases for the application.

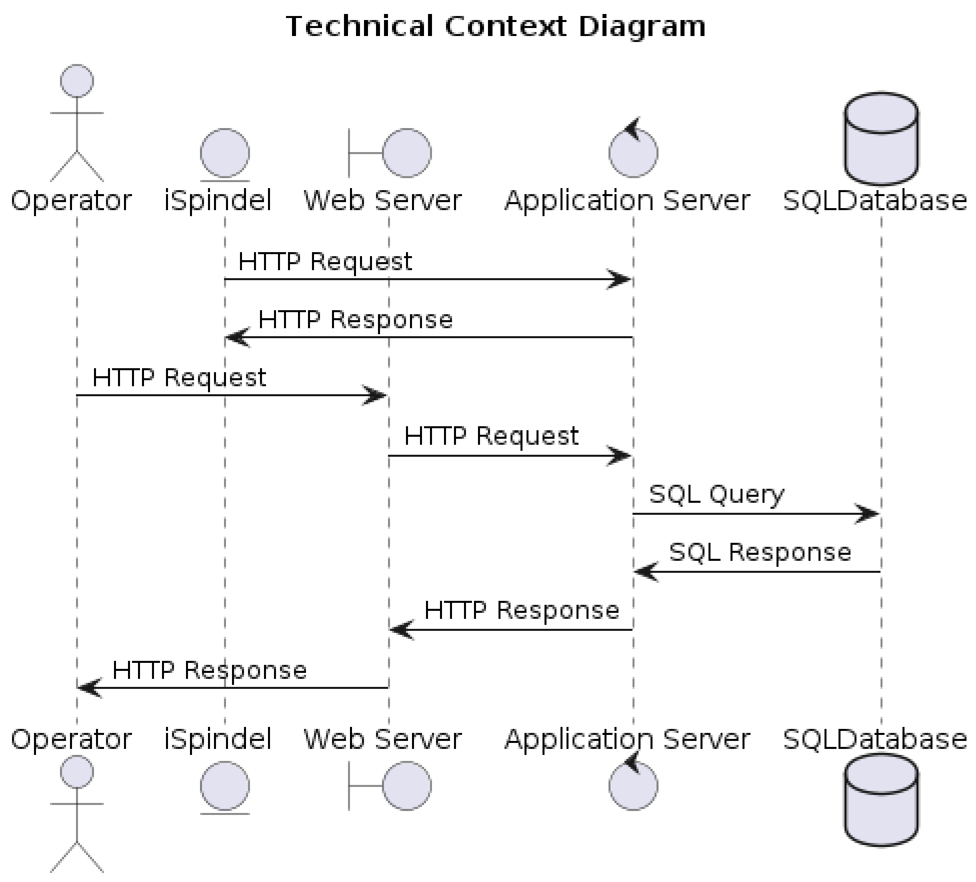
Technical Context

From a technical perspective, the system interacts with several external systems and services. The system is dependent on the iSpindel for collecting real-time data from the brewing process. The system is also dependent on a database for storing and managing data. The system uses GitHub for version control and collaboration. Finally, the system uses Docker for containerization and deployment.

TODO: Create a new diagram for the technical context view.

The technical context diagram provides a high-level overview of the technical environment in which the system operates. The diagram shows the system and its interactions with external systems and services. The diagram also shows the dependencies of the system on external systems and services.

```
@startuml 03-Context-View-Technical

title Technical Context Diagram

actor user as "Operator"
entity ISpindel as "iSpindel"
boundary WebServer as "Web Server"
control AppServer as "Application Server"
database Database as "SQLDatabase"

ISpindel -> AppServer : HTTP Request
AppServer -> ISpindel : HTTP Response

user -> WebServer : HTTP Request
WebServer -> AppServer : HTTP Request
AppServer -> Database : SQL Query
Database -> AppServer : SQL Response
AppServer -> WebServer : HTTP Response
WebServer -> user : HTTP Response

@enduml
```



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.

\clearpage

Solution Strategy

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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

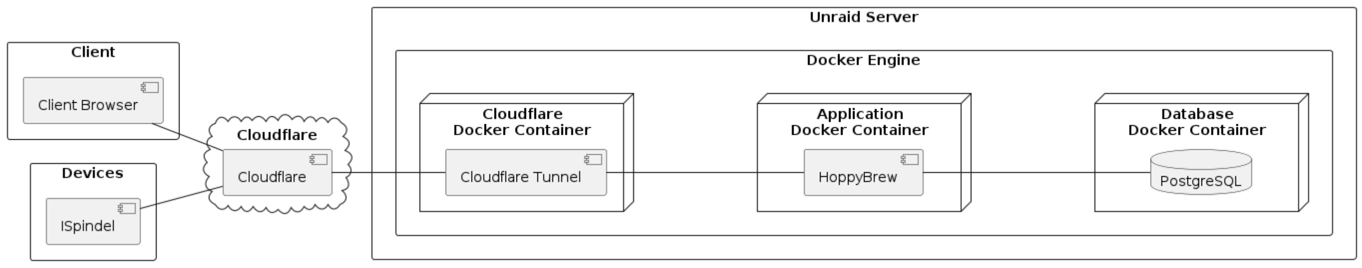
application.
4. Documentation: The application will implement documentation, to provide guidance and support for developers and users. This will help explain the architecture and design of the application, as well as provide instructions for installation and usage.
5. Compliance: The application will 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.

\clearpage

Building Block View

Blackbox Overall System

```

@startuml 05-black-box-overall-system

left to right direction

rectangle "Client" {
    component "Client Browser" as client_browser
}

rectangle "Devices" {
    component "ISpindel" as iSpindel
}

cloud "Cloudflare" {
    component "Cloudflare" as cloudflare
}
rectangle "Unraid Server" {
    rectangle "Docker Engine" {

        node "Cloudflare\nDocker Container" {
            component "Cloudflare Tunnel" as cloudflareTunnel
        }

        node "Application\nDocker Container" {
            component "HoppyBrew" as hoppybrew
        }

        node "Database\nDocker Container" {
            database "PostgreSQL" as postgres
        }
    }
}


client_browser -- cloudflare
iSpindel -- cloudflare
cloudflare -- cloudflareTunnel
cloudflareTunnel -- hoppybrew
hoppybrew -- postgres

@enduml
```



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.

Cloudflare

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

ISpindel

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

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.

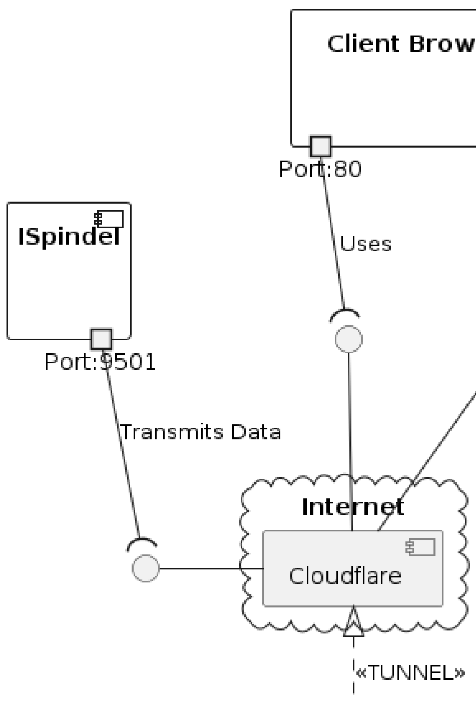
PostgreSQL

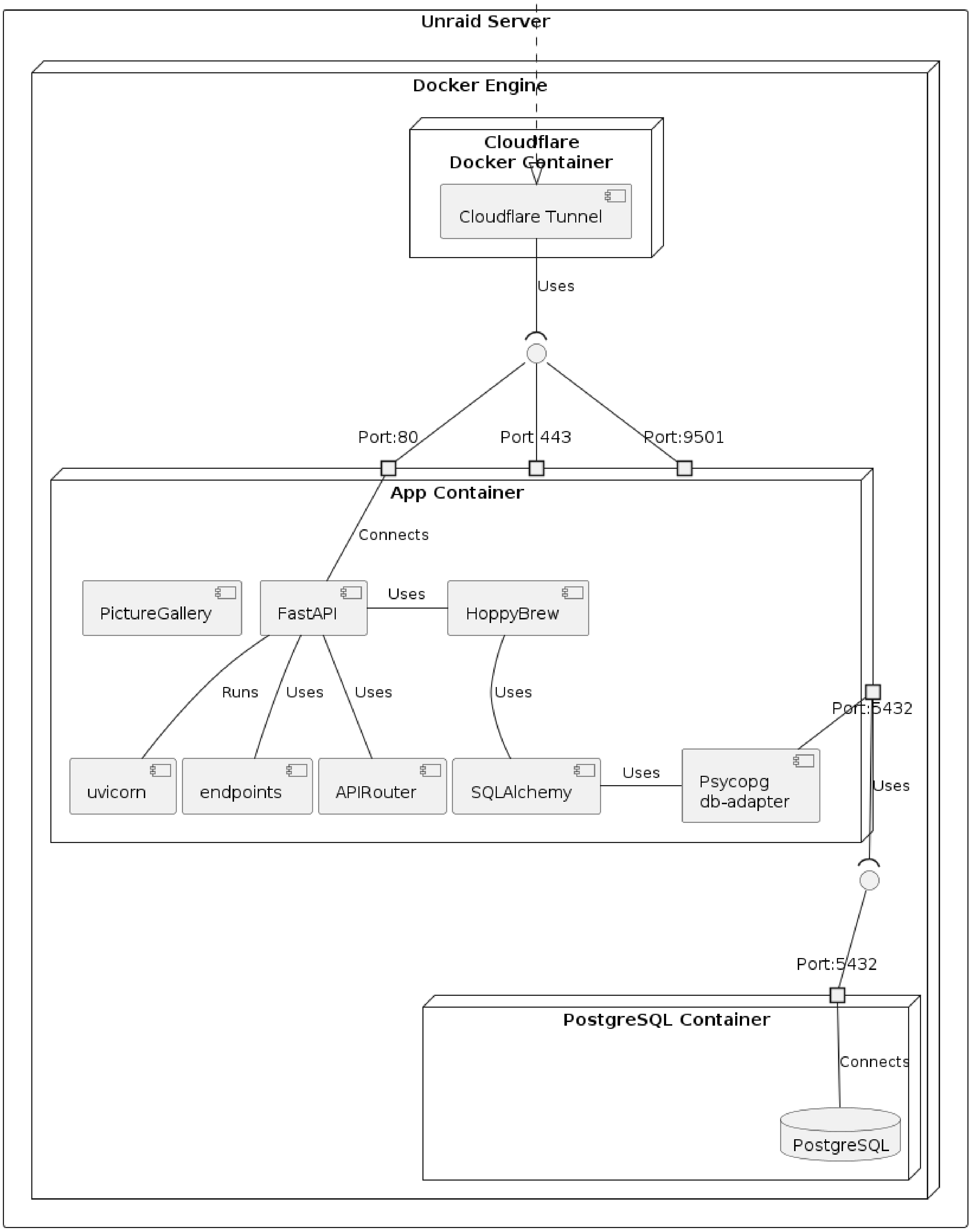
The PostgreSQL database is responsible for storing the data. It communicates with the HoppyBrew application to receive data.

Whitebox Overall System

```
@startuml 04-white-box-overall-system

title White Box Overall System

interface " " as I01
interface " " as I02
interface " " as I03

component "Client Browser" as ClientBrowser{
    portout "Port:80" as ClientPort80
    portout "Port:443" as ClientPort443
}
component "ISpindel" as ISpindel{
    portout "Port:9501" as ISpindel_port
}

cloud "Internet" {
    component "Cloudflare" as cloudflare
}

ClientPort80 -down-( I01 : Uses
I01 -down- cloudflare

ClientPort443 -down-( I02 : Uses
I02 -down- cloudflare

ISpindel_port --( I03 : Transmits Data
I03 - cloudflare


rectangle "Unraid Server" {
    node "Docker Engine" {
        node "Cloudflare\nDocker Container" {
            component "Cloudflare Tunnel" as CloudflareTunnel
        }

        cloudflare <|..|> CloudflareTunnel : <>

        node "App Container" as Application_Container {
            component "HoppyBrew" as HoppyBrew
            component "Psycopg\ndb-adapter" as db_adapter
            component "FastAPI" as api
            component "uvicorn" as uvicorn
            component "endpoints" as endpoints
            component "APIRouter" as APIRouter
            component "PictureGallery" as PictureGallery
            component "SQLAlchemy" as SQLAlchemy

            portin "Port:80" as port80
            portin "Port:443" as port443
            portin "Port:9501" as port9501
            portout "Port:5432" as port5432

            api - HoppyBrew : Uses
            HoppyBrew -- SQLAlchemy : Uses
            SQLAlchemy - db_adapter : Uses
            api -- uvicorn  : Runs
            api -- endpoints  : Uses
            api -- APIRouter  : Uses

            port80 -down- api : Connects
            db_adapter -down- port5432 : Connects
        }

        interface " " as I04
        CloudflareTunnel --( I04 : Uses
        I04 -- port80
        I04 -- port443
        I04 -- port9501

        node "PostgreSQL Container" {
            database "PostgreSQL" as PostgreSQL
            portin "Port:5432" as PostgreSQL_port5432
            PostgreSQL_port5432 - PostgreSQL : Connects
        }

        interface " " as I05
        port5432 -down-( I05 : Uses
        I05 -down- PostgreSQL_port5432
    }
}
@enduml
```

White Box Overall System





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:

PictureGallery (Blackbox)

Intent/Responsibility:

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

Interfaces:

Interface	Description
REST interface	/api/PictureGallery/*

Table: PictureGallery interfaces.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

\clearpage

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.

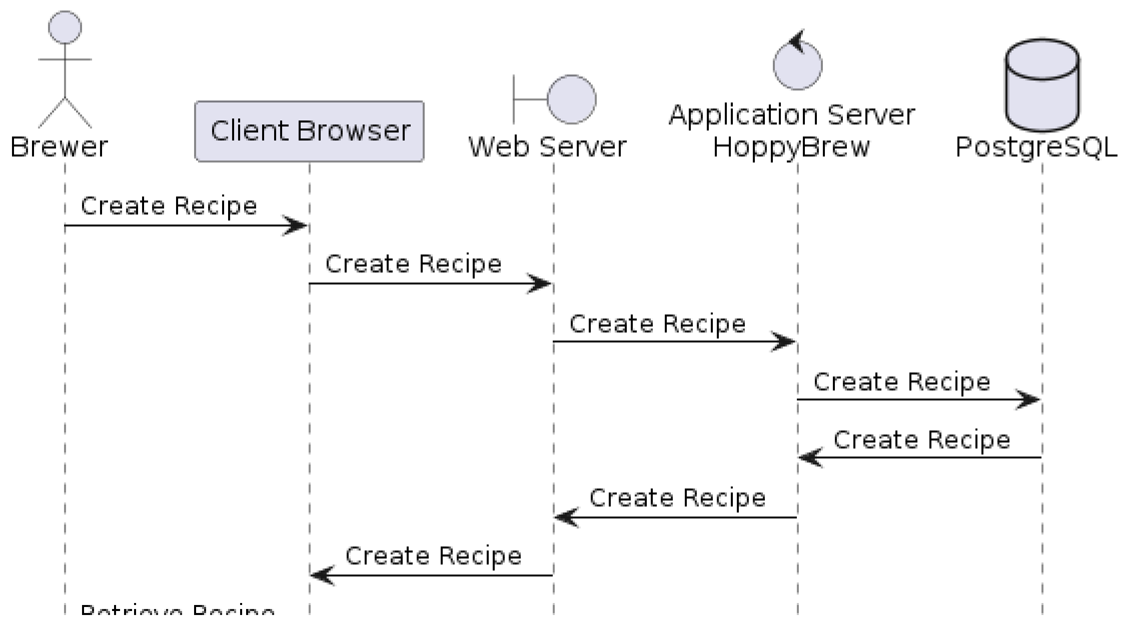
CRUD Recipe

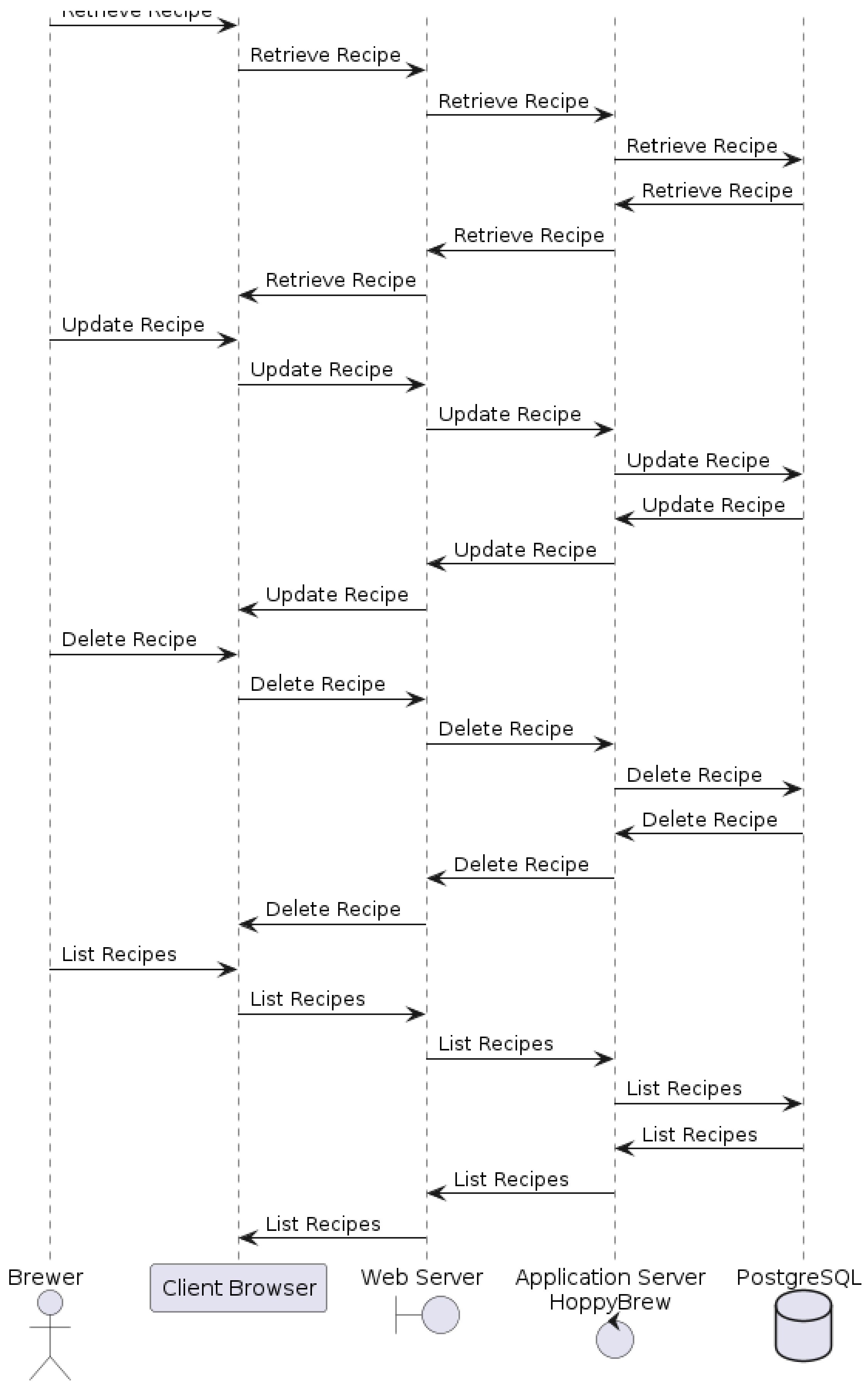
The sequence diagram illustrates CRUD operations for the Recipe entity, showcasing the standard pattern for all CRUD operations. It starts with the Brewer creating a new Recipe, which is then stored in the database. Subsequently, the Brewer can perform actions such as retrieving, updating, and deleting 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.

```
@startuml 06-Runtime-View-CRUD-Recipe

actor Brewer as Brewer
participant "Client Browser" as ClientBrowser
boundary WebServer as "Web Server"
control AppServer as "Application Server\nHoppyBrew"
database "PostgreSQL" as PostgreSQL

Brewer -> ClientBrowser : Create Recipe
ClientBrowser -> WebServer : Create Recipe
WebServer -> AppServer : Create Recipe
AppServer -> PostgreSQL : Create Recipe
PostgreSQL -> AppServer : Create Recipe
AppServer -> WebServer : Create Recipe
```

```
WebServer -> ClientBrowser : Create Recipe

Brewer -> ClientBrowser : Retrieve Recipe
ClientBrowser -> WebServer : Retrieve Recipe
WebServer -> AppServer : Retrieve Recipe
AppServer -> PostgreSQL : Retrieve Recipe
PostgreSQL -> AppServer : Retrieve Recipe
AppServer -> WebServer : Retrieve Recipe
WebServer -> ClientBrowser : Retrieve Recipe

Brewer -> ClientBrowser : Update Recipe
ClientBrowser -> WebServer : Update Recipe
WebServer -> AppServer : Update Recipe
AppServer -> PostgreSQL : Update Recipe
PostgreSQL -> AppServer : Update Recipe
AppServer -> WebServer : Update Recipe
WebServer -> ClientBrowser : Update Recipe

Brewer -> ClientBrowser : Delete Recipe
ClientBrowser -> WebServer : Delete Recipe
WebServer -> AppServer : Delete Recipe
AppServer -> PostgreSQL : Delete Recipe
PostgreSQL -> AppServer : Delete Recipe
AppServer -> WebServer : Delete Recipe
WebServer -> ClientBrowser : Delete Recipe

Brewer -> ClientBrowser : List Recipes
ClientBrowser -> WebServer : List Recipes
WebServer -> AppServer : List Recipes
AppServer -> PostgreSQL : List Recipes
PostgreSQL -> AppServer : List Recipes
AppServer -> WebServer : List Recipes
WebServer -> ClientBrowser : List Recipes

@enduml
```





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.

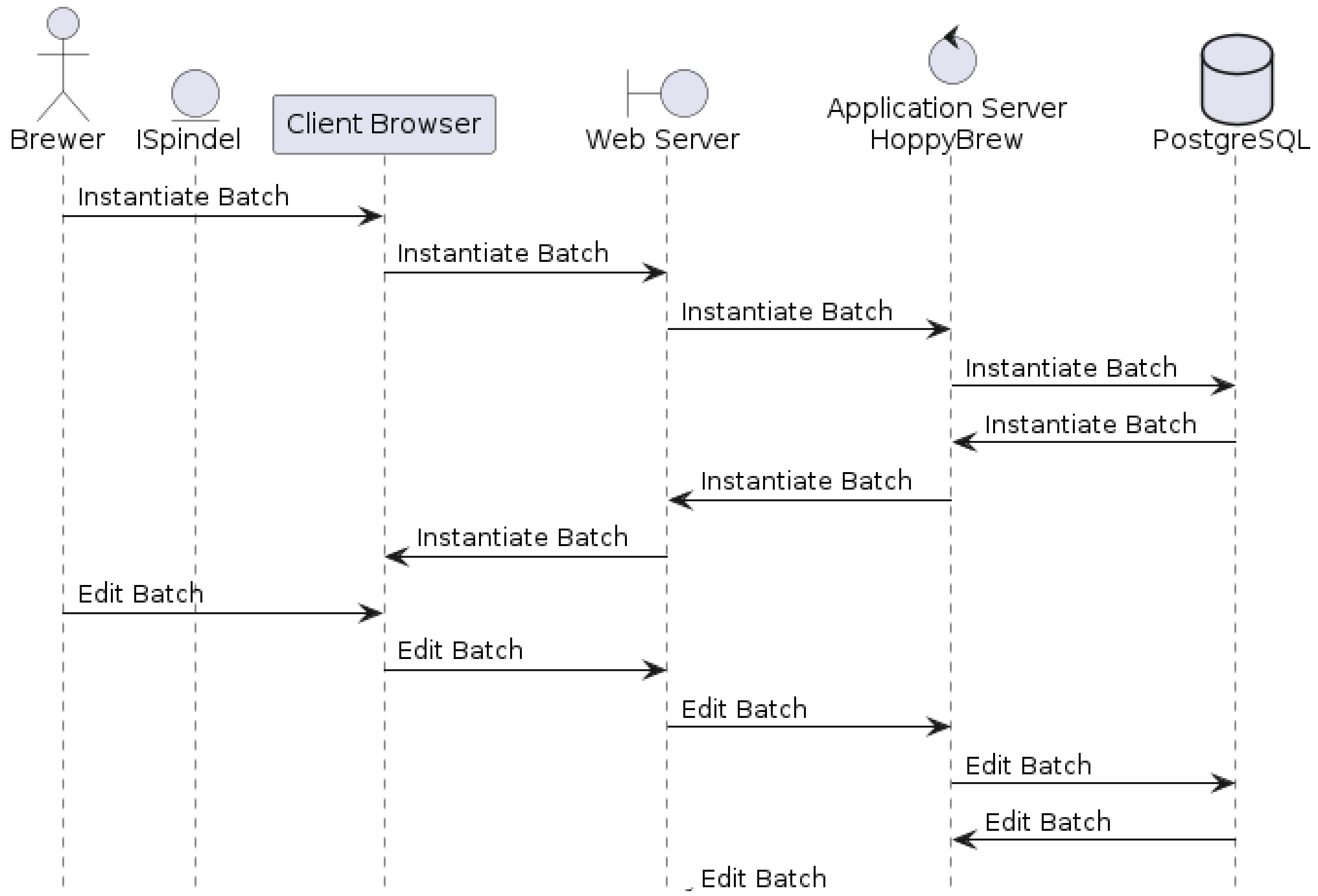
A new Batch is instantiated by the Brewer from an existing Recipe. The Brewer can then edit the Batch, instance without affecting the original Recipe. The Brewer can also delete the Batch if it is no longer needed. The Brewer can also list all Batches stored in the database.

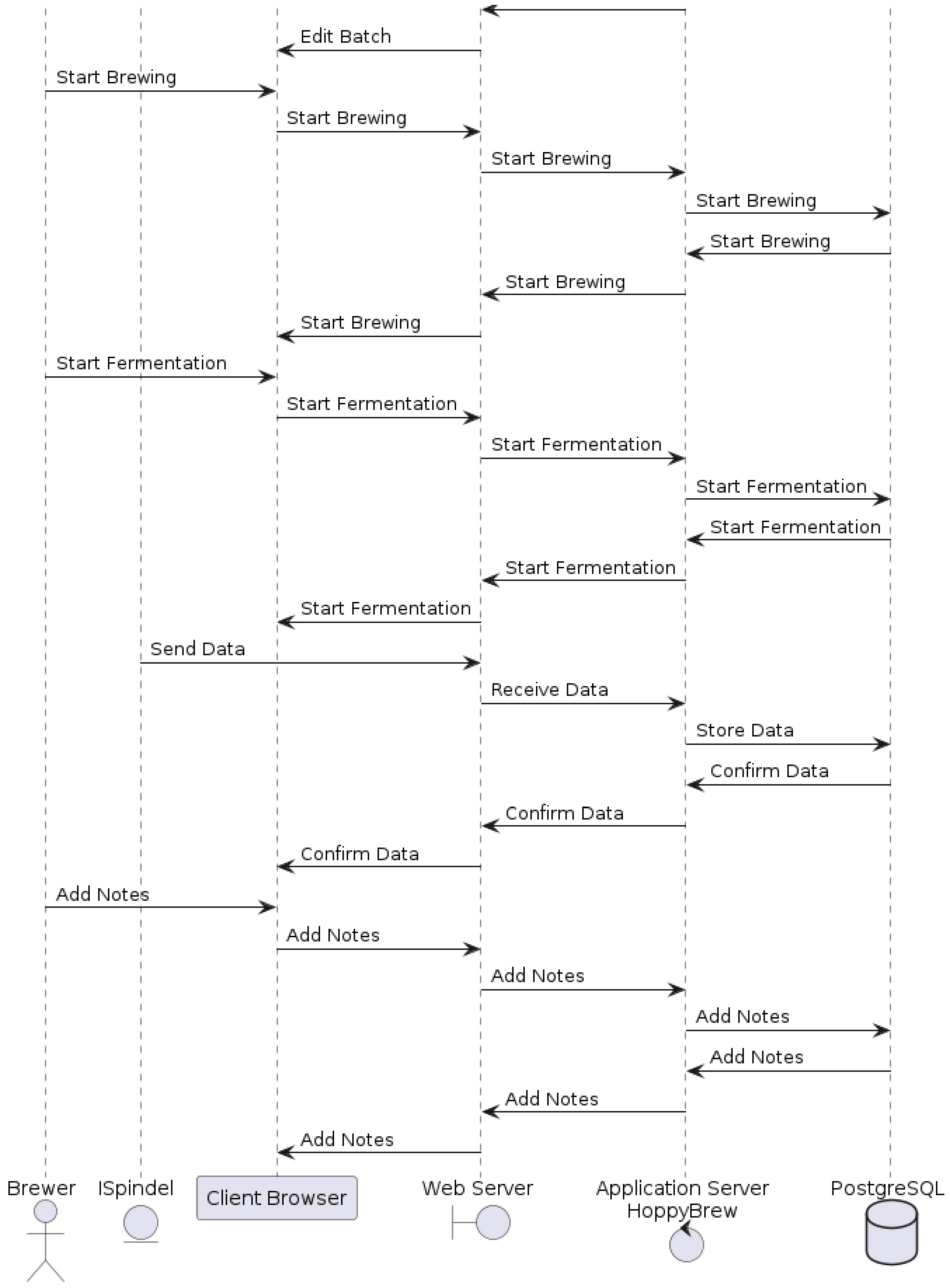
A batch is intended to represent a single brewing process, which is why the batch can undergo Three core brewing stages, which then are followed by a variety of additional stages that are optional like conditioning, archiving, etc.

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.

```
@startuml 07-Runtime-View-CRUD-Batch

actor Brewer as Brewer
entity "ISpindel" as ISpindel
participant "Client Browser" as ClientBrowser
boundary WebServer as "Web Server"
control AppServer as "Application Server\nHoppyBrew"
database "PostgreSQL" as PostgreSQL

Brewer -> ClientBrowser : Instantiate Batch
ClientBrowser -> WebServer : Instantiate Batch
WebServer -> AppServer : Instantiate Batch
AppServer -> PostgreSQL : Instantiate Batch
PostgreSQL -> AppServer : Instantiate Batch
AppServer -> WebServer : Instantiate Batch
WebServer -> ClientBrowser : Instantiate Batch

Brewer -> ClientBrowser : Edit Batch
ClientBrowser -> WebServer : Edit Batch
WebServer -> AppServer : Edit Batch
AppServer -> PostgreSQL : Edit Batch
PostgreSQL -> AppServer : Edit Batch
AppServer -> WebServer : Edit Batch
WebServer -> ClientBrowser : Edit Batch

Brewer -> ClientBrowser : Start Brewing
ClientBrowser -> WebServer : Start Brewing
WebServer -> AppServer : Start Brewing
AppServer -> PostgreSQL : Start Brewing
PostgreSQL -> AppServer : Start Brewing
```

```
AppServer -> WebServer : Start Brewing
WebServer -> ClientBrowser : Start Brewing

Brewer -> ClientBrowser : Start Fermentation
ClientBrowser -> WebServer : Start Fermentation
WebServer -> AppServer : Start Fermentation
AppServer -> PostgreSQL : Start Fermentation
PostgreSQL -> AppServer : Start Fermentation
AppServer -> WebServer : Start Fermentation
WebServer -> ClientBrowser : Start Fermentation
ISpindel -> WebServer : Send Data
WebServer -> AppServer : Receive Data
AppServer -> PostgreSQL : Store Data
PostgreSQL -> AppServer : Confirm Data
AppServer -> WebServer : Confirm Data
WebServer -> ClientBrowser : Confirm Data

Brewer -> ClientBrowser : Add Notes
ClientBrowser -> WebServer : Add Notes
WebServer -> AppServer : Add Notes
AppServer -> PostgreSQL : Add Notes
PostgreSQL -> AppServer : Add Notes
AppServer -> WebServer : Add Notes
WebServer -> ClientBrowser : Add Notes


@enduml
```





Deployment View

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

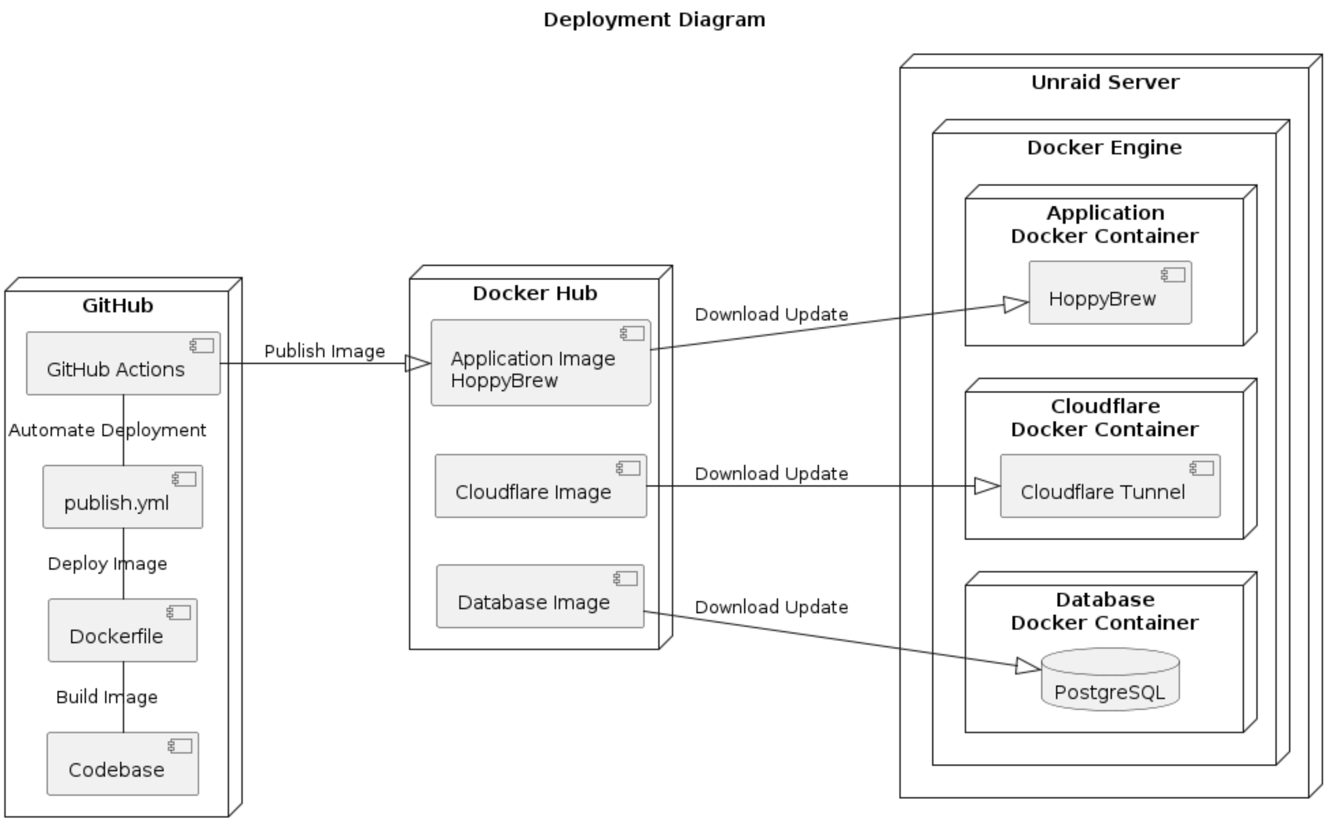
The deployment view describes how the application is deployed and managed in a production environment. The application is deployed using Docker and Docker Compose, with PostgreSQL as the database technology. The codebase is stored in a central repository on GitHub, with a Dockerfile for building the 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.

TODO: Need to figure out how i automate update on the Unraid server when a new version is pushed to Docker Hub.

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.

```
@startuml 08-Deployment-View
title Deployment Diagram

left to right direction

node "GitHub" as github {
    component "Codebase" as codebase
    component "Dockerfile" as dockerfile
    component "publish.yml" as publish
    component "GitHub Actions" as githubActions
}

codebase - dockerfile : Build Image
dockerfile - publish : Deploy Image
publish - githubActions : Automate Deployment


node "Docker Hub" as dockerHub {
    component "Application Image\nHoppyBrew" as appImage
    component "Cloudflare Image" as cloudflareImage
    component "Database Image" as databaseImage
}

node "Unraid Server" as unraid {
    node "Docker Engine" {
        node "Cloudflare\nDocker Container" {
            component "Cloudflare Tunnel" as cloudflareTunnel
        }

        node "Application\nDocker Container" {
            component "HoppyBrew" as hoppybrew
        }

        node "Database\nDocker Container" {
            database "PostgreSQL" as postgres
        }
    }
}

githubActions --|> appImage : Publish Image
appImage --|> hoppybrew : Download Update
cloudflareImage --|> cloudflareTunnel : Download Update
databaseImage --|> postgres : Download Update

@enduml
```



\clearpage

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.

Entity-Relationship Diagram

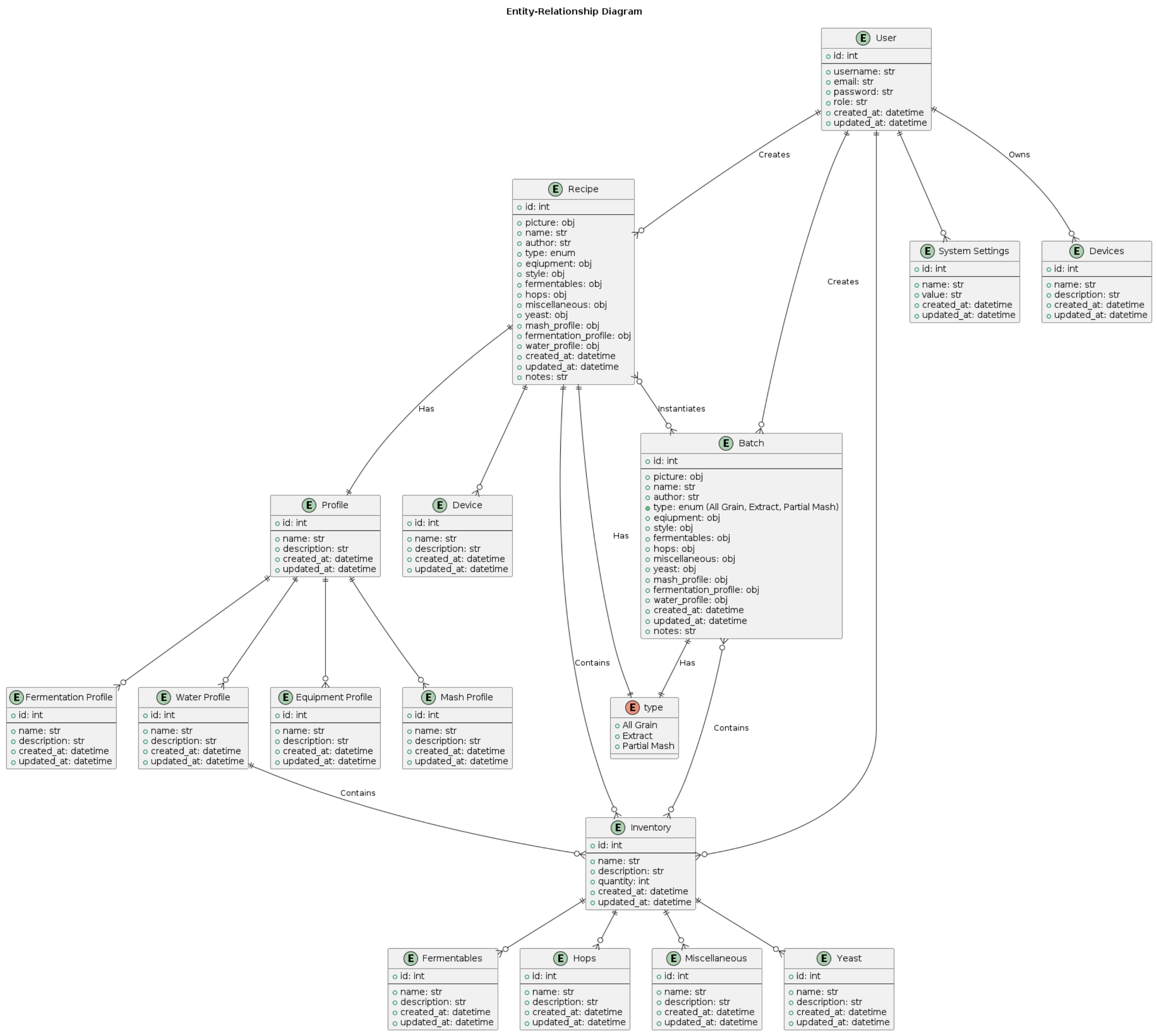
```
@startuml 09-ER-Diagram

title Entity-Relationship Diagram

entity "User" as user {
    + id: int
    --
    + username: str
    + email: str
    + password: str
    + role: str
    + created_at: datetime
    + updated_at: datetime
}

entity "Recipe" as recipe {
    + id: int
    --
    + picture: obj
    + name: str
    + author: str
    + type: enum
    + eqiupment: obj
    + style: obj
    + fermentables: obj
    + hops: obj
    + miscellaneous: obj
    + yeast: obj
    + mash_profile: obj
    + fermentation_profile: obj
    + water_profile: obj
    + created_at: datetime
    + updated_at: datetime
    + notes: str
}

enum "type" as type {
    + All Grain
    + Extract
    + Partial Mash
}

entity "Batch" as batch {
    + id: int
    --
```

```
        + picture: obj
        + name: str
        + author: str
        + type: enum (All Grain, Extract, Partial Mash)
        + eqiupment: obj
        + style: obj
        + fermentables: obj
        + hops: obj
        + miscellaneous: obj
        + yeast: obj
        + mash_profile: obj
        + fermentation_profile: obj
        + water_profile: obj
        + created_at: datetime
        + updated_at: datetime
        + notes: str
    }

    entity "Profile" as profile {
        + id: int
        --
        + name: str
        + description: str
        + created_at: datetime
        + updated_at: datetime
    }

    entity "Device" as device {
        + id: int
        --
        + name: str
        + description: str
        + created_at: datetime
        + updated_at: datetime
    }

    entity "Inventory" as inventory {
        + id: int
        --
        + name: str
        + description: str
        + quantity: int
        + created_at: datetime
        + updated_at: datetime
    }

    entity "System Settings" as systemSettings {
        + id: int
        --
        + name: str
        + value: str
        + created_at: datetime
        + updated_at: datetime
    }
```

```
entity "Equipment Profile" as equipmentProfile {
    + id: int
    --
    + name: str
    + description: str
    + created_at: datetime
    + updated_at: datetime
}

entity "Mash Profile" as mashProfile {
    + id: int
    --
    + name: str
    + description: str
    + created_at: datetime
    + updated_at: datetime
}

entity "Fermentation Profile" as fermentationProfile {
    + id: int
    --
    + name: str
    + description: str
    + created_at: datetime
    + updated_at: datetime
}

entity "Water Profile" as waterProfile {
    + id: int
    --
    + name: str
    + description: str
    + created_at: datetime
    + updated_at: datetime
}

entity "Devices" as devices {
    + id: int
    --
    + name: str
    + description: str
    + created_at: datetime
    + updated_at: datetime
}

entity "Fermentables" as fermentables {
    + id: int
    --
    + name: str
    + description: str
    + created_at: datetime
    + updated_at: datetime
}
```

```
entity "Hops" as hops {
    + id: int
    --
    + name: str
    + description: str
    + created_at: datetime
    + updated_at: datetime
}

entity "Miscellaneous" as miscellaneous {
    + id: int
    --
    + name: str
    + description: str
    + created_at: datetime
    + updated_at: datetime
}

entity "Yeast" as yeast {
    + id: int
    --
    + name: str
    + description: str
    + created_at: datetime
    + updated_at: datetime
}

recipe ||--|| type : "Has"
batch ||--|| type : "Has"

user ||--o{ recipe : "Creates"
recipe }o--o{ batch : "Instantiates"
user ||--o{ batch : "Creates"
user ||--o{ devices : "Owns"
recipe ||--|| profile : "Has"
recipe ||--o{ device
user ||--o{ inventory
user ||--o{ systemSettings

profile ||--o{ equipmentProfile
profile ||--o{ mashProfile
profile ||--o{ fermentationProfile
profile ||--o{ waterProfile

inventory ||--o{ fermentables
inventory ||--o{ hops
inventory ||--o{ miscellaneous
inventory ||--o{ yeast

batch }o--o{ inventory : "Contains"

waterProfile ||--o{ inventory : "Contains"
```

```
recipe ||--o{ inventory : "Contains"

@enduml
```



\clearpage

Architecture Decisions

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.

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.

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.

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.

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.

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.

\clearpage

Quality Requirements

Quality Tree

Quality Scenarios

\clearpage

Risks and Technical Debts

\clearpage

Glossary

Term	Definition
Actor	in use case parlance, are parties outside the system that interact with the system. They may be users or other systems. Each actor defines a coherent set of roles users of the system can play (UML, 1999). Cockburn (1997) distinguishes between primary and secondary actors. A primary actor is one having a goal requiring the assistance of the system. A secondary actor is one from which the system needs assistance to satisfy its goal.
Architecture	The term software architecture is used both to refer to the high-level structure of software systems and the specialist discipline or field distinct from that of software engineering. The architecture of a software system identifies a set of components that collaborate to achieve the system goals. The architecture specifies the “externally visible” properties of the components-i.e., those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on (Bass et al., 1998). It also specifies the relationships among the components and how they interact.

Term	Definition
Conceptual Architecture	The intent of the conceptual architecture is to direct attention at an appropriate decomposition of the system without delving into the details of interface specification and type information. Moreover, it provides a useful vehicle for communicating the architecture to non-technical audiences, such as management, marketing, and many users. The conceptual architecture identifies the system components, the responsibilities of each component, and interconnections between components. The structural choices are driven by the system qualities, and the rationale section articulates and documents this connection between the architectural requirements and the structures (components and connectors or communication/co-ordination mechanisms) of the architecture.
Features	Features are the differentiating functionality of a product. This functionality may not be available in other products, or it may not be available with the same quality characteristics.
Functional Requirements	Functional requirements capture the intended behavior of the system-or what the system will do. This behavior may be expressed as services, tasks or functions the system is required to perform.
Logical Architecture	The logical architecture is the detailed architecture specification, precisely defining the component interfaces and connection mechanisms and protocols. It is used by the component designers and developers.
Meta-architecture	The meta-architecture is a set of high-level decisions that will strongly influence the structure of the system, but is not itself the structure of the system. The meta-architecture, through style, patterns of composition or interaction, principles, and philosophy, rules certain structural choices out, and guides selection decisions and trade-offs among others. By choosing communication or co-ordination mechanisms that are repeatedly applied across the architecture, a consistent approach is ensured and this simplifies the architecture. (See Bredemeyer Consulting and Bredemeyer Consulting.)
Non-functional Requirements	Non-functional requirements or system qualities, capture required properties of the system, such as performance, security, maintainability, etc.-in other words, how well some behavioral or structural aspect of the system should be accomplished.
Product Line	Product lines consist of basically similar products with different cost/feature variations per product.
Product Family	Product families include a number of product lines targeted at somewhat different markets or usage situations. What makes the product lines part of a family, are some common elements of functionality and identity.
Qualities	System qualities, or non-functional requirements, capture required properties of the system, such as performance, security, maintainability, etc.-in other words, how well some behavioral or structural aspect of the system should be accomplished.

Term	Definition
Scenario	A scenario is an instance of a use case, and represents a single path through the use case. Thus, one may construct a scenario for the main flow through the use case, and other scenarios for each possible variation of flow through the use case (e.g., triggered by error conditions, security breaches, etc.). Scenarios may be depicted using sequence diagrams.
Use Case	A use case defines a goal-oriented set of interactions between external actors and the system under consideration. That is, use cases capture who (actors) does what (interactions) with the system, for what purpose (goal). A complete set of use cases specifies all the different ways to use the system, and thus defines all behavior required of the system--without dealing with the internal structure of the system.
Use Case Diagram	A use case diagram is a graphical representation of the use cases and their relationships to the actors.
User Interface	The user interface is the part of the system with which the users interact. It includes all the screens, forms, reports, and so on that the users use to interact with the system.
User Story	A user story is a short, simple description of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.
View	A view is a representation of a whole system from the perspective of a related set of concerns. Views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers, system engineers, and project managers. Views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers, system engineers, and project managers.
Viewpoint	A viewpoint is a specification of the conventions for constructing and using a view. A viewpoint specifies not only the kinds of models that are to be constructed, but also the rules governing the construction of those models.
Viewtype	A viewtype is a template for a view. It specifies the types of models that are to be constructed, and the rules governing the construction of those models.
Work Product	A work product is a document or model that is produced as part of a software development process. Work products are used to capture and communicate information about the system being developed.

\clearpage

Bibliography