

# Design Document

CS 461 Fall 2018

Group 19 BrewHops: Ninkasi Brewing, Automating the brewing process

Brennan Douglas

douglbre@oregonstate.edu

Dan Van Horn

vanhornd@oregonstate.edu

Henry Peterson

peterhen@oregonstate.edu

Bailey Singleton

singletb@oregonstate.edu

November 19th, 2018

## Abstract

During the beer brewing process there are many variables that need to be tested and tracked. This helps determine when the beer is ready and how it compares to other batches. Ninkasi — a brewery in Eugene, Oregon — is using a single large excel spreadsheet to track and store all their information. As they have grown, this has become increasingly unwieldy. There is already the beginning of a solution which includes a small set of applications designed to manage this data automatically while allowing for editing and data visualization. The following describes the plan to complete the system.

## CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	System Purpose . . . . .	1
1.2	System Scope . . . . .	1
1.3	System Overview . . . . .	1
	1.3.1 System context . . . . .	1
	1.3.2 System functions . . . . .	1
	1.3.3 User characteristics . . . . .	1
1.4	Definitions . . . . .	2
<b>2</b>	<b>System Architecture</b>	<b>3</b>
<b>3</b>	<b>Design</b>	<b>3</b>
3.1	Database . . . . .	3
	3.1.1 Existing . . . . .	4
	3.1.2 Updates . . . . .	4
3.2	Front-end . . . . .	4
	3.2.1 View . . . . .	4
	3.2.2 Communication . . . . .	6
	3.2.3 Testing . . . . .	6
3.3	API . . . . .	7
	3.3.1 Typescript Conversion . . . . .	7
	3.3.2 New Functionality . . . . .	7
	3.3.3 Testing . . . . .	7
3.4	Dev-Ops . . . . .	8
	3.4.1 Container . . . . .	8
	3.4.2 Cloud Storage and Load Balancing . . . . .	8
	3.4.3 Web Hosting . . . . .	8
<b>4</b>	<b>Appendix</b>	<b>9</b>
4.1	Desktop Design . . . . .	9
4.2	Mobile Design . . . . .	11
	<b>References</b>	<b>13</b>

## 1 INTRODUCTION

The design for the Ninkasi Brewhops application is outlined based upon the requirements that have been laid out. The application includes four major components that will be broken down in the following way: database, front-end, API, and dev-ops. The technologies chosen for the design expand on those chosen in the technology reviews that have been conducted for each software component.

### 1.1 System Purpose

The purpose of the automated brewing project is to make a more sustainable system for Ninkasi to track, store, and visualize brewing data. As it stands, there is an existing application that has a working database that models the data needed by Ninkasi. The app has some rudimentary pages to display and enter this data. Our extension on the system will improve upon this and add new features.

Upon completion, the application will allow users to enter data for a given beer batch on a mobile or desktop device. The data will then be automatically added to a set of graphs to help visualize the lifecycle of a specific batch. The application will track all data history associated with these batches, the tanks they are in, and the brand they are a part of. This will allow deep analytic looks into the brewing process over years worth of data.

### 1.2 System Scope

The system currently stores all the data associated with each batch of beer that Ninkasi brews in a relational database. The web application serves two purposes: to manage the data (add and update), and to display the data. Along with this there is a concept of user roles, so a basic login and security system is set in place. An extension that this project adds is importing data from Ninkasi's alcolyzer (alcohol content analyzer), in the form of CSV files. This project also adds new forms of data visualization. Specifically, a graph that shows the fermentation curves for each batch. Finally, the technologies being used will be upgraded to bring them to modern standards. These are broken down into three epics:

- Maintenance
- Data Flow
- User Interface

### 1.3 System Overview

#### 1.3.1 System context

Previously, Ninkasi was using large ever expanding excel spreadsheets to track their data brewing data. This single spreadsheet would be passed around by email to different people who would add data and edit data. With the growth of the business, it has become much more cumbersome and error prone.

#### 1.3.2 System functions

The system stands to improve this process via the web application. The current application is not production ready and reaching that point will be the foremost goal during the Maintenance epic. During that phase, a more complete docker-ized version of the application will be developed and reported bugs will be fixed. The Data Flow through the application will be improved by adding new features to allow quicker data entry. Finally, the User Interface will be improved to allow for easier use and better understanding of the website.

#### 1.3.3 User characteristics

There are two types of users: administrators and standard users. The admin will be able to manage all other users and every facet of the application data. They will be on par with a manager who is overseeing the brewing process for a particular brand, or more than one. The standard user will be the tester who makes the measurements on each batch and inputs the data. They will be able to enter data and see the visualizations but won't have much control over the application.

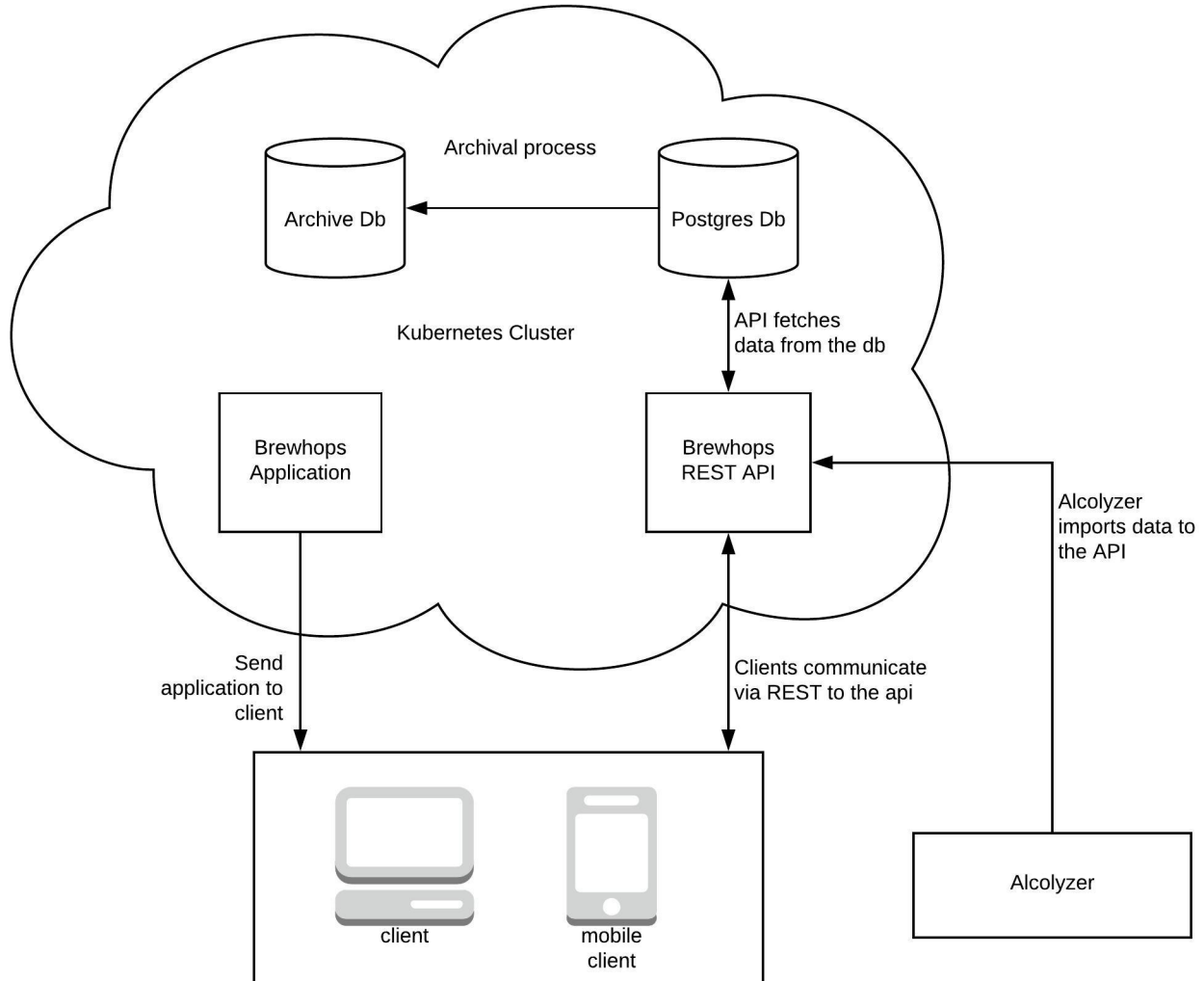
Ninkasi wants a streamlined data entry and visualization process. It needs to be easy to use when a desktop computer isn't available. Most of the data entry will be performed via a tablet. Finally, they want parts of the process to be automated.

## 1.4 Definitions

- We — refers to our team name BrewHops
- Epic — a big chunk of work that has one common objective.
- User Story — a simple description of a software feature from an end-user perspective.
- Agile Software Development — an approach to software development under which requirements and solutions evolve through the collaborative effort of self-organizing and cross-functional teams and their customer(s)/end user(s)
- Dev-Ops — Development Operations, a set of practices that automates the processes between software development and IT teams, in order that they can build, test, and release software faster and more reliably.
- Docker — a container platform for normalizing the application's run-time environment [1]
- Kubernetes — a container (e.g. docker) scripting platform [2]
- K8s — shorthand for Kubernetes
- Vue — the web framework that the project uses [3]
- TypeScript — a typed version of JavaScript [4]
- Microservices — a variant of the service-based architecture where software components are loosely coupled and small.
- Relational Database — a database structured to recognize relations among stored items of information
- Application Programming Interface (API) — a system of software tools and resources that enables developers to create applications
- Single File Component (SFC) — a file for web development that contains markup, functionality, and styling code.
- Single Page Application (SPA) — an app that loads a single HTML page and dynamically changes that page as the user interacts with the app

## 2 SYSTEM ARCHITECTURE

Fig. 1. The system architecture for Ninkasi Brewhops



The BrewHops system is designed with microservice architecture, a variant of the service-based architecture. The three main software components, front-end, API, and database, are loosely coupled and do not rely on each other to run. Individual teammates will be able to develop and scale these pieces without fear of effecting other work.

The front-end is only responsible for asking the API for data and displaying it. If the data needs to change the front-end will pass that work onto the API. The API is the liaison between the front-end and database. It can create, read, update and delete data from the database. The database itself is only responsible for storage. The BrewHops team has experience designing microservice systems from industry. The reason that the team has chosen this pattern is because of its compatibility with cloud hosting services and simplicity rather than combined team experience.

## 3 DESIGN

### 3.1 Database

This section lays out the design of the database. It includes all changes that need to be made to the current schema to account for the software requirements that have been laid out.

### 3.1.1 Existing

The existing database already has the core entities that are needed to keep track of all of the required information. The different batches of beer, their version measurements, recipes, and tanks (the containers that the batches are brewed in) all already exist. Different entities for providing a rudimentary task system also exist, these include: Tasks, Actions, and Employees. The employee entity also represents the user object with user name and password fields. The layout of the database can be seen in the ER diagram at figure 2.

### 3.1.2 Updates

First, security needs to be updated. Plain text passwords shouldn't be kept in the database as it is major security vulnerability. To resolve this, the password attribute on Employee will be converted into a password hash. Upon log in, the given password can be hashed again and compared against the stored hash. This still allows the user to log in but no longer stores their password in plain text.

Next, new functionality defined by the requirements document requires some additions. Every edit that is made to one of the tables needs to be tracked so that they can be audited. The person who made the changes also needs to be tracked. To accomplish this auditing every table will have a "last\_updated\_by" column added to it which will hold the id of the Employee entity who last updated that row. The edits will then be kept track of by implementing "shadow" tables for each entity. A shadow table is a table that is exactly identical to the table schema it is "shadowing". The shadow table makes a new row every time a change occurs to its counter part. The action associated with this change is tracked (e.g. insert, update, and deletes) and the previous/updated values are stored in the new row (which contains a reference to the row that was changed). These shadow tables' naming schema will match the name of their respective table with "\_history" appended onto it. These shadow tables will be implemented with trigger on each of the possible actions: insert, update, delete.

An example of an updated table definition with a shadow table is:

```

1  CREATE TABLE IF NOT EXISTS recipes (
2      id                SERIAL          NOT NULL PRIMARY KEY,
3      name              VARCHAR(30)    NOT NULL,
4      airplane_code     VARCHAR(50)    NOT NULL,
5      yeast             INT            NULL,
6      instructions      JSONB          NOT NULL,
7      last_updated_by   SERIAL          NOT NULL
8  );
9
10 CREATE TABLE IF NOT EXISTS recipes_history (
11     id                INT            NOT NULL PRIMARY KEY,
12     recipe_id         SERIAL          NOT NULL,
13     name              VARCHAR(30)    NOT NULL,
14     airplane_code     VARCHAR(50)    NOT NULL,
15     yeast             INT            NULL,
16     instructions      JSONB          NOT NULL,
17     last_updated_by   SERIAL          NOT NULL
18 );

```

Here a new column was added to the recipe table, "last\_updated\_by" that will hold the id of the last employee to update the row. Then the shadow table, "recipes\_history", was created. The id represents the id of the shadow table and the id that represents the recipes id has been moved to "recipe\_id". Foreign key relationships on these ids have purposely been left off as it causes issues when every table has a reference to one other one. And there is no foreign key for the shadow table's reference to the recipe row as when the row gets deleted it will no longer exist but the shadow table should still hold the data.

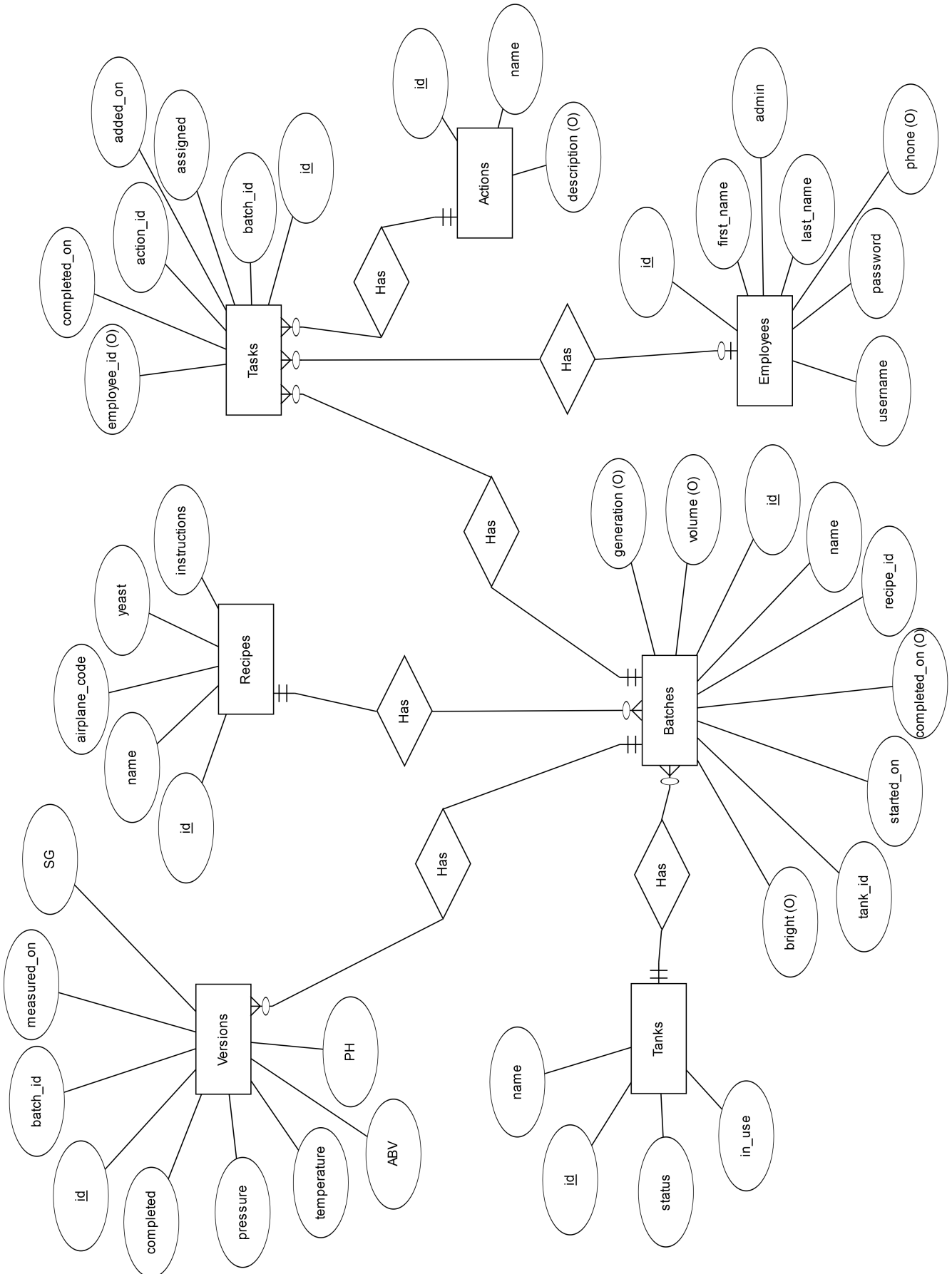
## 3.2 Front-end

### 3.2.1 View

The front-end application will aim to be as light as possible and rely on the external API rather than its own server to authenticate users and access data. It will be written in Vue.js, taking advantage of the single file component (SFC) pattern. An SFC file contains three sections: the view template, logic, and styling. This is contrary to traditional web design where those three sections, namely HTML, CSS, and JavaScript, were largely separated into their own files.

The app will be a single page application (SPA) that will simply switch out SFCs when the user navigates to different pages. The main page component will render its sub-components, also SFCs, preserving separation of concerns. Data and functionality will flow through the application from the top down, as shown in figure 3. This means that any data

Fig. 2. ER diagram for the existing database schema.

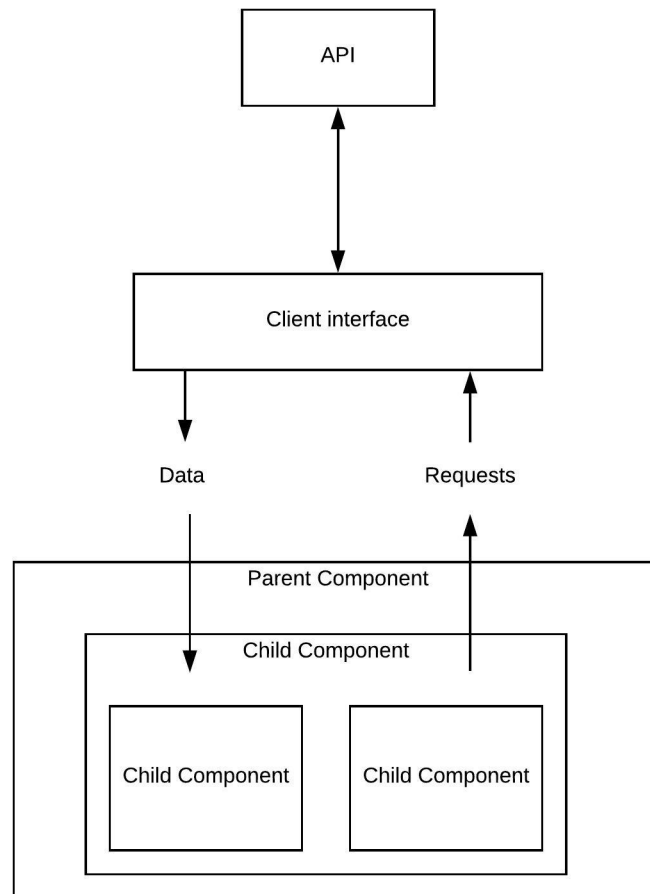


a component needs to render itself will be passed through by its parent. Additionally, if the parent needs to perform some action based on its child component, a function is passed to the child that it can call to notify its parent.

### 3.2.2 Communication

A client interface will be defined to encapsulate the logic for communication with the API. With TypeScript we can define the types that will be returned by the interface, apply static type-checking to it, and by extension to the whole application. The static type-checking will greatly decrease the development time of the front-end and eliminate time spent tracking down data-based bugs. All error handling will be taken care of inside the interface, so view logic remains simple.

Fig. 3. Example of the data flow through view components



### 3.2.3 Testing

Testing the front-end app is a relatively new trend in web development and it has been made very easy with libraries like Jest, written by Facebook. Jest can send mocked data through the app and take snapshots of the rendered components. When front-end code changes, the tests will fail in an expected way, then the developer can confirm the changes made with the new snapshot. When tests fail unexpectedly, this is still good, because Jest is catching some unwanted side effects that need to be addressed.

In addition to snapshot testing, functional testing is available to test any visual dynamic behavior. For example, if there is a drop-down menu, a good unit test for it would be that it opens and closed accordingly when clicked. Although it is tedious test for a human to do, Jest allows us to automate it, so the team benefits from additional unit testing without spending the extra time on it.



### 3.3 API

#### 3.3.1 Typescript Conversion

We will convert the back-end code from pure Javascript to Typescript because it is a super-set of JavaScript and therefore backwards compatible. The process of converting will be fairly quick. This will primarily involve defining types and adding those types to the appropriate variable declarations and functions. Types provide better readability for those coming to the project as well as ensure robustness in code as it is written. The following is an example of a JavaScript function compared with its TypeScript counterpart.

JavaScript:

```

1  buildUpdateString (keys, values) {
2      keys = keys.split(',')
3      let query = ``
4      let idx = 1
5      for (var i in keys) {
6          let key = keys[i]
7          query += `${key} = \${idx}, `
8          idx++
9      }
10     query = query.substring(0, query.length - 2)
11     return {
12         query,
13         idx
14     }
15 }
```

TypeScript:

```

1  buildUpdateString (keys: string, values: string): IUpdateString {
2      let key_list:string[] = keys.split(',')
3      let query: string = ``
4      let idx: number = 1
5      for (let key in key_list) {
6          query += `${key} = \${idx}, `
7          idx++
8      }
9      query = query.substring(0, query.length - 2)
10     return {
11         query,
12         idx
13     }
14 }
```

In the situation above, the type `IUpdateString` would also have to be defined, but it would also be used in other functions that return the same type of value.

#### 3.3.2 New Functionality

Due to updates being made in the database and the front-end, we will have to similarly update the back-end so that data can be passed between the two without issue. This functionality will mirror much of the logic already implemented and will be written directly in TypeScript. It will have query functions to update and retrieve data from new database tables as well as REST protocol functions to update the user interface and receive changes.

#### 3.3.3 Testing

We will write testing for the back-end using Jest. This requires that we create a mock database for these functions to hit, running different types of valid and invalid data through them. We will make sure that all types of valid data presents us with correct results and that invalid data is handled properly such that it does not crash the application. This will improve the consistency of the application by showing that at least all of the known cases can be handled. It will also help with future development because team members can check that, when they make changes, the application still behaves in the expected fashion.

3.4 Dev-Ops

3.4.1 Container

Docker will be used as our container for each instance of our application. The Docker containers will allow us to compartmentalize instances, giving us the ability to test and deploy multiple in a very short amount of time. It will help considerably if the website gets heavy traffic, preventing it from being bogged down with many requests. This will be the first part of the Dev-Ops tasks that will need to be setup as this is the most basic building block of our web application.

These containers will be set up on virtual machines hosted on Amazon Web Services, which are then load balanced. Once the Docker containers are fully functional and tested, we can then move on to setting up the load balancer.

3.4.2 Cloud Storage and Load Balancing

We will handle load balancing and storage through Amazon Web Services. The containers will need to live somewhere, and so will the data they collect. Using Amazon Web Services, we will be able to spin up our containers on the cloud, and use Amazon’s Elastic Load Balancing tools to direct traffic. With multiple containers getting an even amount of traffic through the load balancer, we should not have a problem with constant up-time and little to no downtime on our site.

Amazon Web Services will also be where we can store our database of information on tanks, batches, history, etc. The information will be available to all of the Docker containers, and will be accessed through the client interface to the API. See **section 3.2.2** for more information on sending and receiving information to the API.

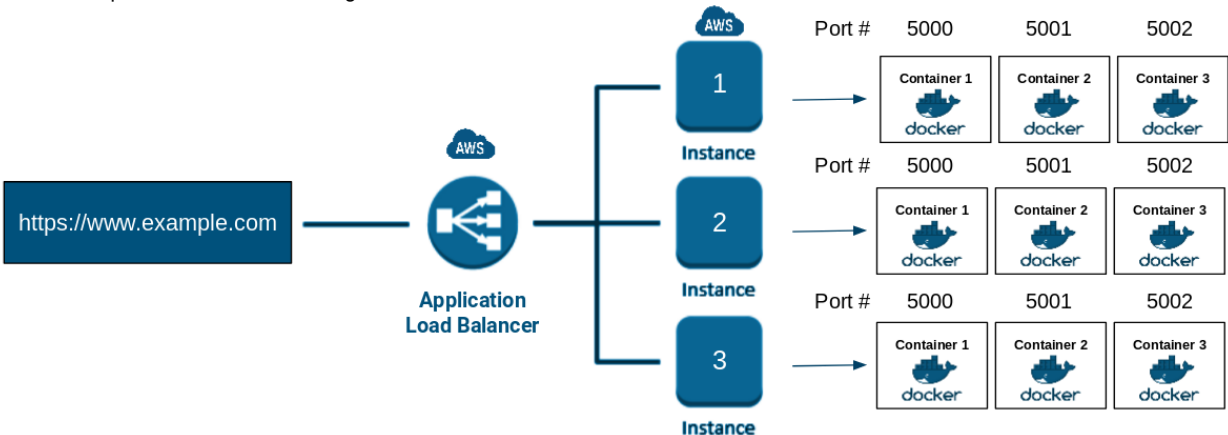
Once we have load balanced Docker containers, we can test traffic throughput and make sure our load balancer can handle enough if not more traffic than we anticipate. Once we move past the load balanced container and have it all set up correctly, we can host it on Amazon Web Service’s EC2.

3.4.3 Web Hosting

As of now, the website is hosted on a previous student’s Oregon State ENGR server. Our goal is to take the site down, and set it up on a more professional, business-grade environment. Similarly to our Cloud Storage and Load Balancing, we will be using Amazon’s EC2 hosting option to distribute our website. Amazon’s EC2 gives us a lot of easy options when working in unison with their elastic load balancer. EC2 hosting will take requests and forward them to our load balancer, which will be the elastic load balancer. From there, the load balancer will decide on which Docker Container the user should go to, based on traffic.

Web hosting can be tested easily, using Amazon Web Services built-in diagnostic tool, and also by checking the container id’s when going through each instance. It can be usability tested also by checking and seeing if the sites load properly.

Fig. 4. Left side represents our Web Hosting URL. It then cascades down to the Load Balancer and then to each instance and container




4 APPENDIX

4.1 Desktop Design

Fig. 5. The Login page for desktop

Ninkasi Brew Hops



Login

username

password

Submit

Fig. 6. The desktop homepage

Logout

Home

Data Entry

Tank

Tank

Action

No Action

Recipe

Recipe

pH

ABV

Bright

Pressure

Generation

Volume

Specific Gravity

Temperature

mm/dd/yyyy --:--:--

Time Measured

Batch Name

Submit

Batch Histories

Tank Info

F1 PRISM 4726-4731 36°F Crash	F3 PRISM 4794-4798 41.1°F Crash	F5 IPA 4759-4764 34°F Transfer	F6 PRISM 4765-4770 36°F Crash
F7 TRI 4748-4753 35°F Transfer	F8 MIS 4775-4776 60°F Transfer	F9 IRA 4784-4788 40.2°F Crash	F4 TRI 4778-4783 35.9°F Crash
F2 RAIN 4653-4656 31°F OK	F2 OKT 4711-4715 36°F Crash	F5 - 30 Jul	F5 - 30 Jul IRA 4806-4808 68°F OK
F8 - 30 Jul IPA 4809-4812 68°F OK	F7 - 1 Aug PRISM 4822-4827 68°F Set to 50		

Fig. 7. The tank information page

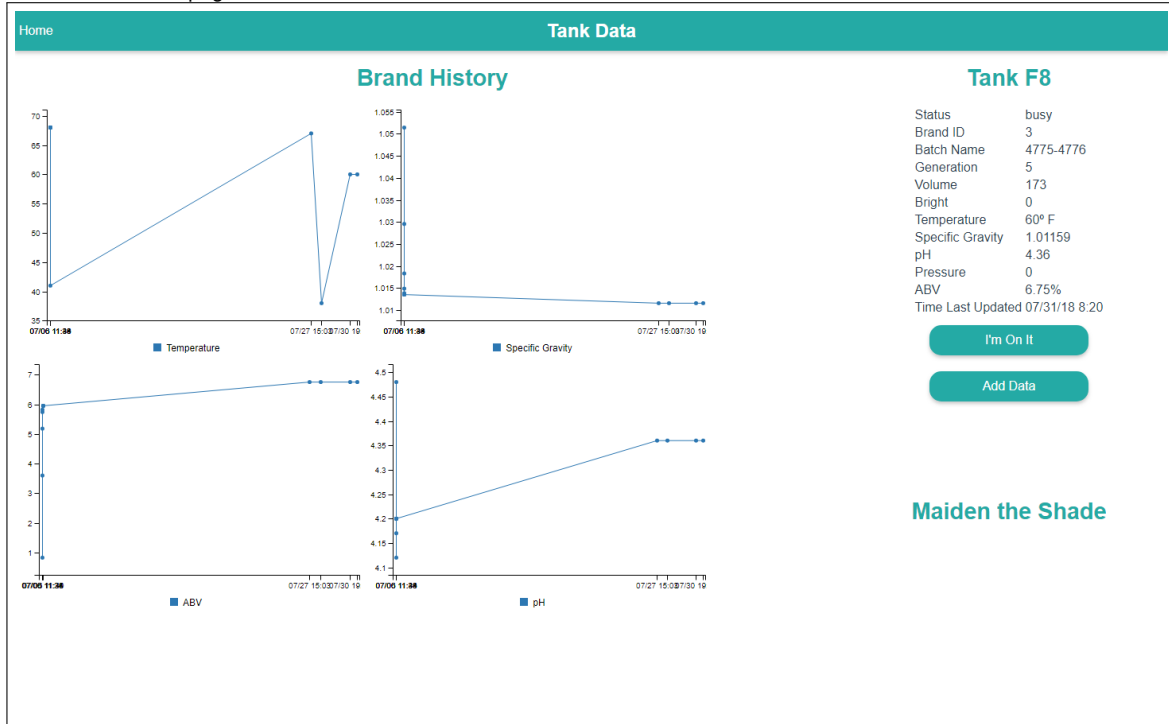


Fig. 8. The data entry page

Home **Data Entry**

Tank Action Recipe

Tank No Action Recipe

pH ABV Bright

Pressure Generation Volume

Specific Gravity Temperature

mm/dd/yyyy --:--:--


Time Measured

Batch Name

[Submit](#)

4.2 Mobile Design

Ninkasi Brew Hops



Login

username

password

Submit

LogoutHome

Menu

Tank Monitoring

Data Entry

Schedule

HomeTank Data

Tank F6

Statusbusy

Brand ID2

Batch Name4765-4770

Generation5

Volume560.6

Bright0

Temperature36° F

Specific Gravity1.01474

pH4.21

Pressure0

ABV5.78%

Time Last Updated 08/02/18 9:22

I'm On It

Add Data

Prismatic

Mosaic Cryo.25

Simcoe Cryo.15

Mobile login page

Mobile home page

Mobile tank page

Create New Tank

New Tank Number

Status on Tank

☐ Tank is in use

Submit

Update Tank Status

Tank Number

Status on Tank

Submit

Create New User

First Name

Last Name

Username

Password

Re-enter password

☐ User is admin

Submit

Create New Brand

Brand Name

Airport Code

Yeast

Dry Hop/Adjunct

Rate

Dry Hop/Adjunct

Rate

Dry Hop/Adjunct

Rate

Dry Hop/Adjunct

Rate

Mobile admin page

Data Entry

Tank

Tank

Action

No Action

Recipe

Recipe

pH

ABV

Bright

Pressure

Generation

Volume

Specific Gravity

Temperature

mm/dd/yyyy --:-- --  
Time Measured

Batch Name

Submit

Batch Histories

Tank Info

F1 PRISM 4726-4731 Crash	F3 PRISM 4794-4798 Crash
F5 IPA 4759-4764 Transfer	F6 PRISM 4765-4770 Crash
F7 TRI 4748-4753 Transfer	F8 MIS 4775-4776 Transfer
F9 IRA 40.2°F	F4 TRI 35.9°F

Mobile data entry page

Tank Info

F1 PRISM 4726-4731 Crash	F3 PRISM 4794-4798 Crash
F5 IPA 4759-4764 Transfer	F6 PRISM 4765-4770 Crash
F7 TRI 4748-4753 Transfer	F8 MIS 4775-4776 Transfer
F9 IRA 4784-4788 Crash	F4 TRI 4778-4783 Crash
F2 RAIN 4653-4656 OK	F2 OKT 4711-4715 Crash
F5 - 30 Jul	F5 - 30 Jul

Mobile tank info page

## REFERENCES

- [1] "What is a container," Oct 2018. [Online]. Available: <https://www.docker.com/resources/what-container>
- [2] "What is kubernetes?" [Online]. Available: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>
- [3] "React a javascript library for building user interfaces." [Online]. Available: <https://reactjs.org/>
- [4] "Get typescript." [Online]. Available: <https://www.typescriptlang.org/>