

All Ground Up



Mateen Ali, Alejandro Almaraz, Jessica Arriaga,
Jett Canavarro, Steven Coffey

<https://github.com/JettCanavarro/CSC468-group-7>

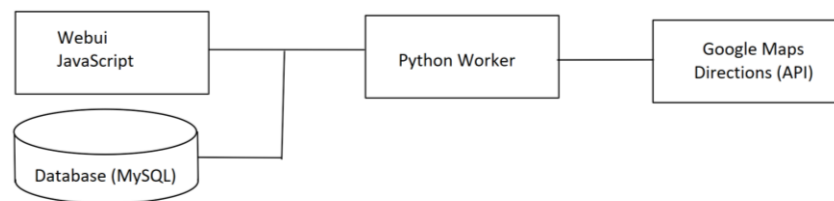
Summary

All Ground Up is a cloud-based web application designed to allow users to review and order-in from their favorite local coffee shops with ease. The application connects customers to local coffee shops with a review driven design. The customers can order directly from the application and the order will be ready for pick up. Vendors can change and update their menu which will be updated directly for the customers to view. As soon as a menu item is updated on a menu it will be available for review by the consumers who order it. Only customers that order a specific item can review it to reduce unsolicited reviews. All Ground Up is created using Docker containers and deployed through Kubernetes. Kubernetes is tasked with the orchestration of our Docker containers for our cloud-based application.

Chapter 1: All Ground Up Vision

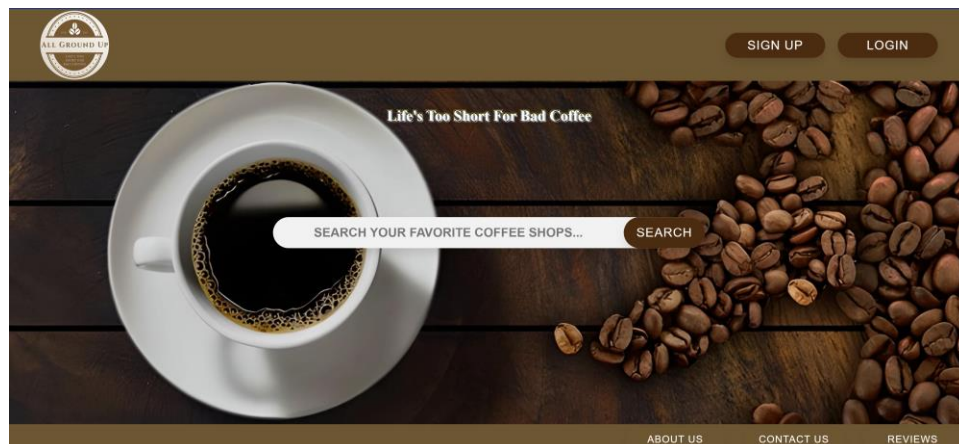
All Ground Up is a cloud-based web application for users to order from their favorite local coffee shops and post reviews from their order. All Ground Up will have platforms for both consumers and vendors. The consumer platform will allow users to order from their local coffee shops with ease and post their reviews on their orders. Users will be able to see all the reviews posted from other customers before ordering, so they can make the best coffee choice for themselves. The customer can place their coffee order on the application for in-store pickup. The order ahead feature allows customers the opportunity to skip the lines and quickly pick up their coffee on the way to their destination. The vendor platform will allow local coffee shops the opportunity to post their menu, offer deals, and fulfill orders that were placed by the customers.

Project Design



Above is the design behind All Ground Up. There are three different main components that go into the project. The web UI, database, and worker will all work seamlessly together to give the user the best experience. The fourth component within our project is the Google Maps API. Within the application, the API is used to give customers directions from their location to their desired coffee shop. This will allow users to have all their needs met within the All Ground Up application.

All Ground Up Application



The All Ground Up application is a web-based application which is shown in the screenshot above. When users go to the main page of the application, the above webpage is the first page they will be visiting. From here, users can either sign up, login, or search for their favorite coffee shops.

Chapter 2: All Ground Up Implementation Proposal

To create All Ground Up, we will need to create three main parts of the application which include the web UI, database, and worker. The front-end web UI will be created using React.js. React.js is a JavaScript development framework used for the creation of web applications. JavaScript is a programming language used to create interactive and dynamic websites. React.js is a free and open-source front-end JavaScript library which allows developers to create simple single-page applications or larger multi-page applications. We chose to use React.js because it is open-source and is a library that allows developers to create web applications with ease.

The back-end portion of the application consists of the database and worker. For the database, we will use MySQL. MySQL is an open-source relational database management system. MySQL provides its own structured query language (SQL) syntax and allows users to manage and manipulate data within MySQL using SQL. The data within MySQL are stored in tables that are related logically. We chose to use MySQL because it is open-source unlike many other database management systems and it is compatible with Python. The database will keep track of each user, transaction, menu, etc. The database design is crucial to helping the web application run smoothly and MySQL is perfect for this task.

For the back-end worker, we will use Python. Python is a programming language that has more simple and straightforward syntax than other programming languages like Java. Python also has many libraries and tools that are already made available for use. Two of these tools that are important for our application are Flask and mysql-connector. Flask is a web framework for Python that provides users with tools, libraries, and technologies to build web applications. This is going to be crucial for interacting with React.js and providing the back-end code for the web UI. Mysql-connector is a library for connecting to MySQL databases, which is necessary for the worker to interact with the database. The back-end Python worker is going to be interacting with the database to pull and push needed information. We chose to use Python for our worker because it has uncomplicated syntax, as well as libraries and tools that connect easily with MySQL and React.js.

The last part of our projects design is to have our application running in the cloud. To do this, there are several important pieces we need to consider. The first includes Docker which is a software platform that allows users to build, test, and run applications quickly on a host machine or in a virtual environment. Docker uses images to build containers that can pull, push, and package software which has everything the software needs to run like libraries and code. The next piece is Kubernetes which is used to support multiple Docker containers running at the same time. Finally, we will use Jenkins to create a CI/CD pipeline. Jenkins is an open-source automated server that allows for continuous integration. CI/CD stands for continuous integration/continuous deployment. This will allow us to automate the process of building, testing, and deploying changes we make to our application. All these pieces will be used together for our application to successfully run within the cloud.

Chapter 3: Building Docker Images

There are three individual docker images that were created to make the app functional. There's a backend image for Python, MYSQL image to store data and a front-end image to run the React app. The steps that are taken to build the docker images for these components are mentioned in detail below:

1. Building the worker (back-end) image:

```
FROM python:alpine

RUN pip install mysql-connector-python
RUN pip install flask
RUN pip install flask_cors

COPY worker.py /
CMD ["python", "worker.py"]
```

For the Python image, we created a Docker image that is built off the python: alpine image. From this image, we were able to add specific installations necessary for our Python worker. We installed mysql-connector, flask and flask_cors since the worker will be interacting with both the database and the web UI. Finally, we copied the worker Python files into the image, so we can run our worker within the container. During our testing, we were able to successfully build the Docker image with our test code, build the container and run test Python code within the Docker container.

```
import mysql.connector as sqlconnector
from flask import Flask, render_template, request
from flask_cors import CORS

app = Flask(__name__)
cors = CORS(app)

connection = sqlconnector.connect(user='user', password='123', host='mysql-container', database='maindb')

@app.route('/api/signup', methods=['POST'])
def signup():
    data = request.get_json()

    firstname = data['firstName']
    lastname = data['lastName']
    username = data['username']
    password = data['password']
    email = data['email']
    phone = data['phoneNumber']
    address = data['address']
    zipcode = data['zipcode']

    cursor = connection.cursor()
    cursor.execute('SELECT ZIP_ID FROM ZIP WHERE ZIP_CODE IN (%s)', (zipcode, ))
    result = cursor.fetchone()
    zip_id = result[0]

    cursor.execute('INSERT INTO CUSTOMER (FIRST_NAME, LAST_NAME, USERNAME, PASS, EMAIL, PHONE, CUSTOMER_ADDRESS, ZIP_ID) VALUES (%s, %s, %s, %s, %s, %s, %s, %s)',
    connection.commit()
    cursor.close()

    return {'status': 'success'}
```

The screenshot above shows the first part of the worker.py file. At the top of the file, there are three import statements of mysql.connector, flask, and flask_cors. These import statements match the three libraries that were installed within the Dockerfile. Below the import statements, the Flask app is created, and the connection is established with the MySQL database. The first function, signup(), within the worker.py file is for the sign-up page. The signup function gets the data from the web UI in a JSON format and then saves each piece of data to a separate variable. The function then gets the zip_id from the database that correlates with the zip code the user entered. Finally, the function inserts the customer into the database with the specific zip_id and returns the status of the function is successful. The Python worker is able to interact with both MySQL and React.js.

2. Building the database (MySQL) image:

```
FROM mysql/mysql-server:latest
#ight bring error if llatest version is different

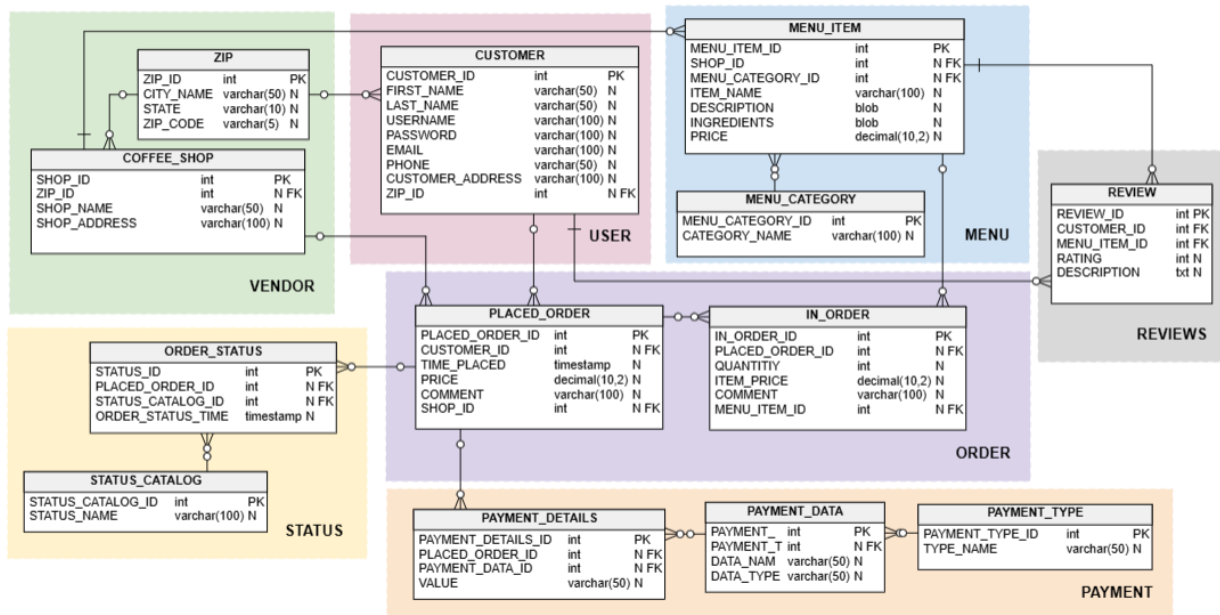
# Set the root password for MySQL
ENV MYSQL_ROOT_PASSWORD=password

# Create a database and user
ENV MYSQL_DATABASE=maindb
ENV MYSQL_USER=user
ENV MYSQL_PASSWORD=123

# Copy the custom SQL script into the container
COPY ./init.sql /docker-entrypoint-initdb.d/
#copy csv file into Dockerfile
#No path required if in same directory for csv file
EXPOSE 3066
```

The Dockerfile for the database image uses the "mysql/mysql-server:latest" base image, sets the root password, creates a database and user, and copies the SQL script into the container. To build the image, navigate to the directory containing the Dockerfile and run the command "docker build -t <image-name>.", replacing <image-name> with the desired name for the image. After building the image, it can be run using the command "docker run -d -p 3306:3306 <image-name>". During initial testing, we were able to successfully build the MySQL image, build the container, and run our SQL files within the container. Within an interactive container, we were also able to work with the database to insert information, select specific tables, and interact with the database successfully.

Database Design



For our data, we are creating a new database to manage all the data needed for All Ground Up. The database design above is the database schema which shows how our database is set up and formatted. We are creating base data for our project by creating CSV files for each table with the data we want to import. The CSV files are an efficient way to load multiple lines of data into our MySQL database with one command. The Python worker will be responsible for collecting and managing our database tables.

```

USE maindb;

CREATE TABLE ZIP (
  ZIP_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  CITY_NAME VARCHAR(50) NULL,
  STATE VARCHAR(10) NULL,
  ZIP_CODE VARCHAR(5) NULL
);

CREATE TABLE COFFEE_SHOP (
  SHOP_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  ZIP_ID INT NULL,
  SHOP_NAME VARCHAR(50) NULL,
  SHOP_ADDRESS VARCHAR(100) NULL,
  FOREIGN KEY (ZIP_ID) REFERENCES ZIP (ZIP_ID)
);

CREATE TABLE CUSTOMER (
  CUSTOMER_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  FIRST_NAME VARCHAR(50) NULL,
  LAST_NAME VARCHAR(50) NULL,
  USERNAME VARCHAR(100) NULL,
  PASS VARCHAR(100) NULL,
  EMAIL VARCHAR(100) NULL,
  PHONE VARCHAR(50) NULL,
  CUSTOMER_ADDRESS VARCHAR(100) NULL,
  ZIP_ID INT NULL,
  FOREIGN KEY (ZIP_ID) REFERENCES ZIP (ZIP_ID)
);

```

Above is a screenshot of the init.sql file. This file has all the code used to create all the tables for the database and reflects the database design. The first line within the file indicates the tables are going to be created within the “maindb” database and all the tables will be created with their corresponding columns. The tables had to be created in a specific order as they have many foreign keys and the connections to other tables can only be made after the table with the primary key is created.

3. Building the front-end image:

```

FROM node:16-alpine as builder

WORKDIR /app

COPY package.json ./
COPY package-lock.json ./
COPY ./ ./

RUN npm ci
RUN npm run build

FROM nginx:1.21.0-alpine as production

ENV NODE_ENV production

COPY --from=builder /app/build /usr/share/nginx/html
COPY ./nginx/nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

```

The Dockerfile for the front-end image uses the "node:16-alpine" base image, sets the working directory to "/app", copies the package.json, package-lock.json, and all other files from the current directory to "/app", installs the required packages, and builds the application. It then uses the "nginx:1.21.0-alpine"

base image, sets the environment variable "NODE_ENV" to "production", copies the build artifacts to Nginx, and sets up the Nginx configuration. To build the image, navigate to the directory containing the Dockerfile and run the command "docker build -t <image-name>.", replacing <image-name> with the desired name for the image. After building the image, it can be run using the command "docker run -d -p 80:80 <image-name>".

```
import React, { useState } from 'react';
import './MainPage.css';
import logo from './img/agu logo 4.png';
import backgroundImage from './img/background.png';

function MainPage(props) {
  const { handlePageChange } = props;
  const backgroundStyle = {
    backgroundImage: `url(${backgroundImage})`,
  };
};

const [showAboutUs, setShowAboutUs] = useState(false);

const toggleAboutUs = () => {
  setShowAboutUs(!showAboutUs);
};

return (
  <div className="MainPage">
    <div className="Header">
      <img src={logo} alt="All Ground Up Logo" />
      <div className="Buttons">
        <button onClick={() => handlePageChange('SignUp')}>Sign Up</button>
        <button onClick={() => handlePageChange('Login')}>Login</button>
      </div>
    </div>
    <div className="Container" style={backgroundStyle}>
      <div className="SearchContainer">
        <form className="SearchBar">
          <input type="text" placeholder="Search your favorite local coffee shops..." />
          <button type="submit">Search</button>
        </form>
      </div>
      <div className="Slogan">
        <h2>"Life's Too Short For Bad Coffee"</h2>
      </div>
    </div>
    <div className="Footer">
      <div className="FooterButtons">
        <button className="FooterButton" onClick={toggleAboutUs}>About Us</button>
        <button className="FooterButton">Contact Us</button>
        <button className="FooterButton">Reviews</button>
      </div>
    </div>
    {showAboutUs &&
      <div className="AboutUsModal">
        <div className="AboutUsContent">
          <span className="CloseButton" onClick={toggleAboutUs}>&times;</span>
          <p>All Ground Up is a cloud-based web application designed to allow users to order coffee delivery from their</p>
        </div>
      </div>
    }
  </div>
);
}
```

```
.MainPage {
  height: 100vh;
  display: flex;
  flex-direction: column;
  background-image: linear-gradient(to bottom right, #7a6f41, #c4a777);
}

.Header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  height: 16%;
  padding: 0 50px;
  background-color: #745730;
}

.Header img {
  width: 110px;
  height: 110px;
  border-radius: 50%;
}

.Buttons {
  display: flex;
}

button {
  width: 160px;
  height: 40px;
  margin-left: 20px;
  font-size: 20px;
  text-transform: uppercase;
  letter-spacing: 1px;
  font-weight: 500;
  color: #fff;
  background-color: #542c0f;
  border: none;
  border-radius: 25px;
  box-shadow: 0px 8px 15px rgba(0, 0, 0, 0.1);
  transition: all 0.3s ease 0s;
  cursor: pointer;
  outline: none;
}

button:hover {
  background-color: #724a34;
}
```

The web UI portion of the application is made up of many files. The files above are the ones that make up the main page of our application. The file above on the left is the MainPage.js file and the file above on the right is the MainPage.css file. The JavaScript (js) file is used to create the page and make the page functional while the Cascading Style Sheets (css) file is used to style the page. Most of our JS files have a corresponding CSS file so the formatting is visually appealing for our users.

Chapter 4: Connecting Components with Docker

The first way we worked to connect our components was through a Docker network, `allgroundup`. First, we built images of the React app web UI, the MySQL database, and the Python worker, named `webui`, `mysql`, and `worker`, respectively.

```
ja958931@head:~/CSC468-group-7$ docker run --name mysql_cont --network allgroundup -e MYSQL_ROOT_PASSWORD=password -d mysql
c5ab5727a8bc2af759351aab029a6c6c2cdfb57c9bf9feb0e7f8c9161037e69d
ja958931@head:~/CSC468-group-7$ docker run --name webui_cont -d -p 8090:80 --net allgroundup webui
0d9c71498c7cddbda1f5bc0f22cdecafe7a45532453df1c1166a9da4ba030ebc1
ja958931@head:~/CSC468-group-7$ docker run --name worker_cont --network allgroundup -d -it worker sh
419c4aa53728ca2d863cfd5f3b05db33459ad0d7168ab50c505454833d43b5c7
```

We ran the three commands above to create the three containers on our `allgroundup` Docker network. Each of the commands to build the running containers within the Docker network worked successfully.

```
{
  "Name": "allgroundup",
  "Id": "d5f70c56523356de971de92cc669bc231587b59e330ceb81a6f6e86cdf1d348",
  "Created": "2023-05-07T20:29:06.344508498-06:00",
  "Scope": "local",
  "Driver": "bridge",
  "EnableIPv6": false,
  "IPAM": {
    "Driver": "default",
    "Options": {},
    "Config": [
      {
        "Subnet": "172.20.0.0/16",
        "Gateway": "172.20.0.1"
      }
    ]
  },
  "Internal": false,
  "Attachable": false,
  "Ingress": false,
  "ConfigFrom": {
    "Network": ""
  },
  "ConfigOnly": false,
  "Containers": {
    "0d9c71498c7cddbda1f5bc0f22cdecafe7a45532453df1c1166a9da4ba030ebc1": {
      "Name": "webui_cont",
      "EndpointID": "5063e2330fbdc85107fd5d92b9114bab3134aafe7f332b0286c53f089ff40478",
      "MacAddress": "02:42:ac:14:00:03",
      "IPv4Address": "172.20.0.3/16",
      "IPv6Address": ""
    },
    "419c4aa53728ca2d863cfd5f3b05db33459ad0d7168ab50c505454833d43b5c7": {
      "Name": "worker_cont",
      "EndpointID": "432ff3dc4a7d5b34970985d4686a8ce4e0f66b59ca330552ae73d9ca289e2d61",
      "MacAddress": "02:42:ac:14:00:04",
      "IPv4Address": "172.20.0.4/16",
      "IPv6Address": ""
    },
    "c5ab5727a8bc2af759351aab029a6c6c2cdfb57c9bf9feb0e7f8c9161037e69d": {
      "Name": "mysql_cont",
      "EndpointID": "fc1b6dc8ac899646c247b039a2cd8c06f0a59a66d5321c25b3044cae151633fa",
      "MacAddress": "02:42:ac:14:00:02",
      "IPv4Address": "172.20.0.2/16",
      "IPv6Address": ""
    }
  }
}
```

To make sure the containers were on `allgroundup`, we entered the command `'docker network inspect allgroundup'`, which showed the containers are listed and were indeed running on the same network. Above is a screenshot of the output of the command with all three containers listed under the containers section.

```

ja958931@head:~/CSC468-group-7$ docker exec -ti webui_cont ping mysql_cont
PING mysql_cont (172.20.0.2): 56 data bytes
64 bytes from 172.20.0.2: seq=0 ttl=64 time=0.331 ms
64 bytes from 172.20.0.2: seq=1 ttl=64 time=0.203 ms
64 bytes from 172.20.0.2: seq=2 ttl=64 time=0.210 ms
64 bytes from 172.20.0.2: seq=3 ttl=64 time=0.205 ms
64 bytes from 172.20.0.2: seq=4 ttl=64 time=0.197 ms
64 bytes from 172.20.0.2: seq=5 ttl=64 time=0.304 ms
^C
--- mysql_cont ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max = 0.197/0.241/0.331 ms
ja958931@head:~/CSC468-group-7$ docker exec -ti worker_cont ping mysql_cont
PING mysql_cont (172.20.0.2): 56 data bytes
64 bytes from 172.20.0.2: seq=0 ttl=64 time=0.308 ms
64 bytes from 172.20.0.2: seq=1 ttl=64 time=0.309 ms
64 bytes from 172.20.0.2: seq=2 ttl=64 time=0.284 ms
64 bytes from 172.20.0.2: seq=3 ttl=64 time=0.258 ms
64 bytes from 172.20.0.2: seq=4 ttl=64 time=0.238 ms
64 bytes from 172.20.0.2: seq=5 ttl=64 time=0.207 ms
^C
--- mysql_cont ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max = 0.207/0.250/0.308 ms
ja958931@head:~/CSC468-group-7$ docker exec -ti worker_cont ping webui_cont
PING webui_cont (172.20.0.3): 56 data bytes
64 bytes from 172.20.0.3: seq=0 ttl=64 time=0.298 ms
64 bytes from 172.20.0.3: seq=1 ttl=64 time=0.220 ms
64 bytes from 172.20.0.3: seq=2 ttl=64 time=0.270 ms
64 bytes from 172.20.0.3: seq=3 ttl=64 time=0.217 ms
64 bytes from 172.20.0.3: seq=4 ttl=64 time=0.204 ms
^C
--- webui_cont ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.204/0.241/0.298 ms
ja958931@head:~/CSC468-group-7$ docker exec -ti webui_cont ping worker_cont
PING worker_cont (172.20.0.4): 56 data bytes
64 bytes from 172.20.0.4: seq=0 ttl=64 time=0.218 ms
64 bytes from 172.20.0.4: seq=1 ttl=64 time=0.199 ms
64 bytes from 172.20.0.4: seq=2 ttl=64 time=0.213 ms
64 bytes from 172.20.0.4: seq=3 ttl=64 time=0.205 ms
^C
--- worker_cont ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.199/0.208/0.218 ms
ja958931@head:~/CSC468-group-7$

```

To ensure the containers could communicate, the command `'docker exec -ti webui_cont ping mysql_cont'` was entered. The screenshot above shows these two containers were able to successfully speak to one another. We ran the same command multiple times with a different combination of the three containers to make sure all the containers were able to communicate properly. As seen in the screenshot above, all the containers were able to successfully ping each other. To see further how the containers interacted, we created a test python script that creates a SQL table and adds data to the test table.

```

import mysql.connector as sqlconnector
connection = sqlconnector.connect(user='user', password='123', host='mysql_cont', database='maindb')
cursor = connection.cursor()
cursor.execute('CREATE TABLE ZIPTEST (ZIP_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY, CITY_NAME VARCHAR(50) NULL, STATE VARCHAR(10) NULL, ZIP_CODE VARCHAR(5) NULL)')

city = "West Chester"
state = "PA"
zip_code = "19383"

cursor.execute('INSERT INTO ZIPTEST (CITY_NAME, STATE, ZIP_CODE) VALUES (%s, %s, %s)', (city, state, zip_code))
connection.commit()
cursor.close()

print("Success")

```

/ # python worker.py
Success

The first screenshot above is the test Python script we created to add a ziptest table into the maindb database in the MySQL container. The second screenshot above is the print statement showing the code was successfully run with no errors. To make sure the Python script actually changed the maindb database, we switched to the MySQL container to check.

```

ja958931@head:~/CSC468-group-7$ docker exec -it mysql_cont bash
bash-4.4# mysql -u user -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 85
Server version: 8.0.32 MySQL Community Server - GPL

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> USE maindb
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

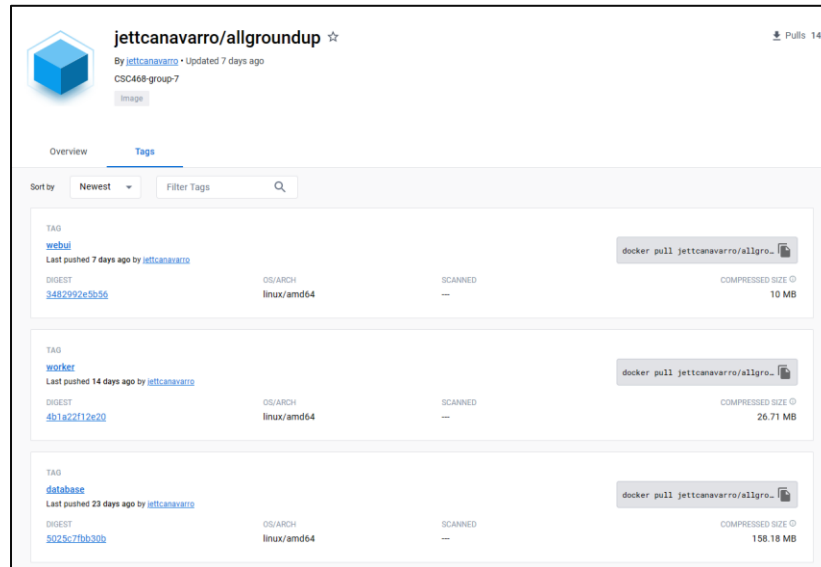
Database changed
mysql> SELECT * FROM ZIPTEST
->
+----+-----+-----+-----+
| ZIP_ID | CITY_NAME | STATE | ZIP_CODE |
+----+-----+-----+-----+
| 1 | West Chester | PA | 19383 |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>

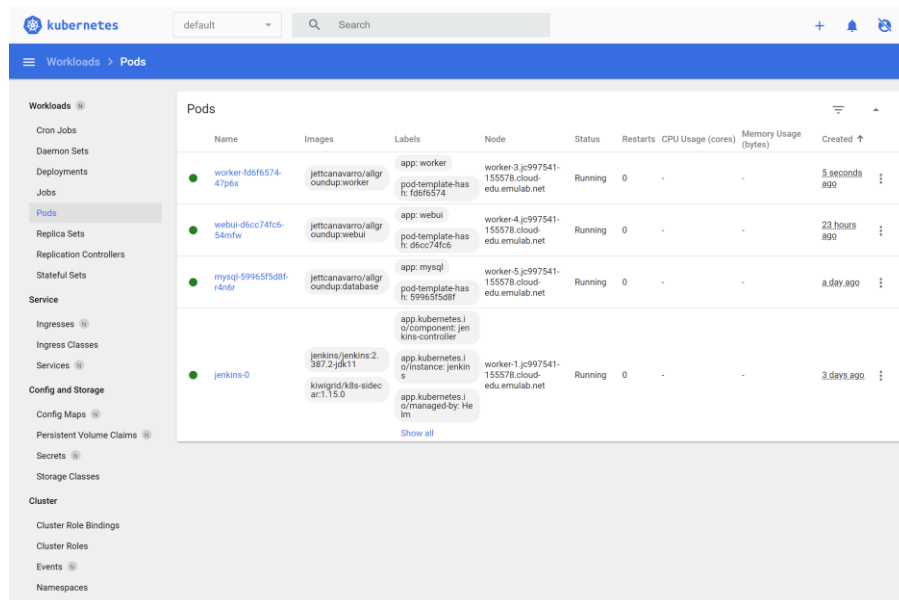
```

When we entered the MySQL container, we were able to see the tables were successfully updated from the Python script. The screenshot above shows the MySQL container being executed with an interactive terminal and the command `'SELECT * FROM ZIPTEST'` being run within the terminal. The test data from the Python script successfully updated the database.

This successful communication allowed our group to continue the development stage and provide fuller functionality for our worker and web UI. We were then able to use Kubernetes to assist with the containerization of our application.



Before we worked with Kubernetes, we uploaded all our Docker images to DockerHub. Above is a screenshot of all the images on Docker Hub under the allgroundup repository.



Above is a screenshot of the Kubernetes dashboard and all of the pods running for our application.

Chapter 5: Final Stage of Project

All Ground Up was able to successfully launch three containers for orchestration by Kubernetes with the CI/CD pipeline. Below is a breakdown of each component and the effort and struggles that went into each one.

1. The Database layout for All Ground Up took some time initially to get implemented properly. We started the process of getting a working database by getting a basic database working without all the tables implemented. We wanted to get proof of concept working before uploading all necessary information. Once we figured out that we could use the base image “mysql/mysql-server:latest”, we were able to set a root password, create a database, user, and copied the SQL script we wrote for testing purposes to build the image. We then opted to use CSV files to quickly and efficiently populate our database with customer and vendor information. The Python worker will handle future database changes to eliminate the need to manually update our database.

```
USE maindb;
CREATE TABLE ZIP (
  ZIP_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  CITY_NAME VARCHAR(50) NULL,
  STATE VARCHAR(10) NULL,
  ZIP_CODE VARCHAR(5) NULL
);

CREATE TABLE COFFEE_SHOP (
  SHOP_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  ZIP_ID INT NULL,
  SHOP_NAME VARCHAR(50) NULL,
  SHOP_ADDRESS VARCHAR(100) NULL,
  FOREIGN KEY (ZIP_ID) REFERENCES ZIP (ZIP_ID)
);

CREATE TABLE CUSTOMER (
  CUSTOMER_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  FIRST_NAME VARCHAR(50) NULL,
  LAST_NAME VARCHAR(50) NULL,
  USERNAME VARCHAR(100) NULL,
  PASS VARCHAR(100) NULL,
  EMAIL VARCHAR(100) NULL,
  PHONE VARCHAR(50) NULL,
  CUSTOMER_ADDRESS VARCHAR(100) NULL,
  ZIP_ID INT NULL,
  FOREIGN KEY (ZIP_ID) REFERENCES ZIP (ZIP_ID)
);

CREATE TABLE STATUS_CATALOG (
  STATUS_CATALOG_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  STATUS_NAME VARCHAR(100) NULL
);

CREATE TABLE PLACED_ORDER (
  PLACED_ORDER_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  CUSTOMER_ID INT NULL,
  ORDER_TIME TIMESTAMP NULL,
  PRICE DECIMAL(10, 2) NULL,
  COMMENT VARCHAR(100) NULL,
  SHOP_ID INT NULL,
  FOREIGN KEY (CUSTOMER_ID) REFERENCES CUSTOMER (CUSTOMER_ID),
  FOREIGN KEY (SHOP_ID) REFERENCES COFFEE_SHOP (SHOP_ID)
);
```

Above is a screenshot of the first part of the init.sql file which has all the create statements for our tables. This is the basis for the whole database relies on the creation of the table.

customer_id	first_name	last_name	username	pass	email	phone	customer_address	zip_id
1	Jessica	Arriaga	coffee123	BuyCoffee01	coffee123@gmail.com	2123052244	844 S Campus Drive	1
2	Jett	Canavarrro	cloud123	Cloud01	cloudlover@gmail.com	7173342434	844 S Campus Drive	1
3	Mateen	Ali	user123	User01	user123@gmail.com	2234567789	844 S Campus Drive	1
4	Alejandro	Almaraz	computer123	Computer01	computer123@gmail.com	2237899900	844 S Campus Drive	1
5	Steven	Coffey	games123	Games01	games123@gmail.com	7174845566	844 S Campus Drive	1

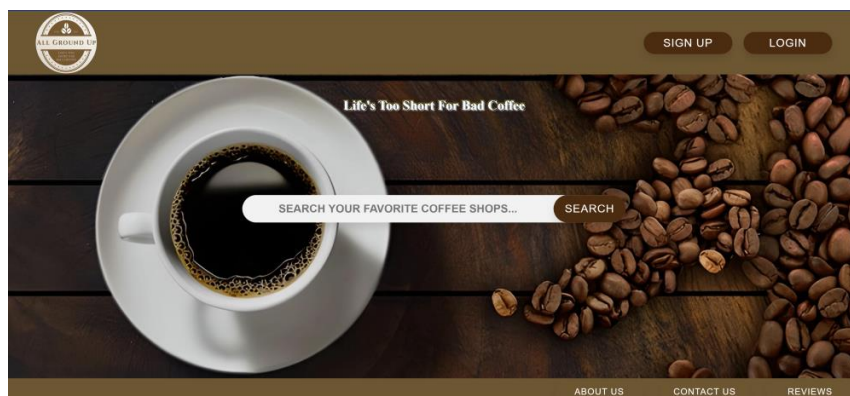
shop_id	zip_id	shop_name	shop_address
1	1	Turks Head Cafe	111 N Church St
2	1	Gryphon Cafe	111 W Gay St
3	2	Small World Coffee	14 Witherspoon St
4	2	The Coffee Club	5 Prospect Ave

Since we created our own database for the application, we had to fill the tables with initial base data so we could have information to test with. The most efficient way to fill the tables with data was to use CSV files to fill the tables. Above are two of the CSV files we created to fill the customer table and the coffee shop table.

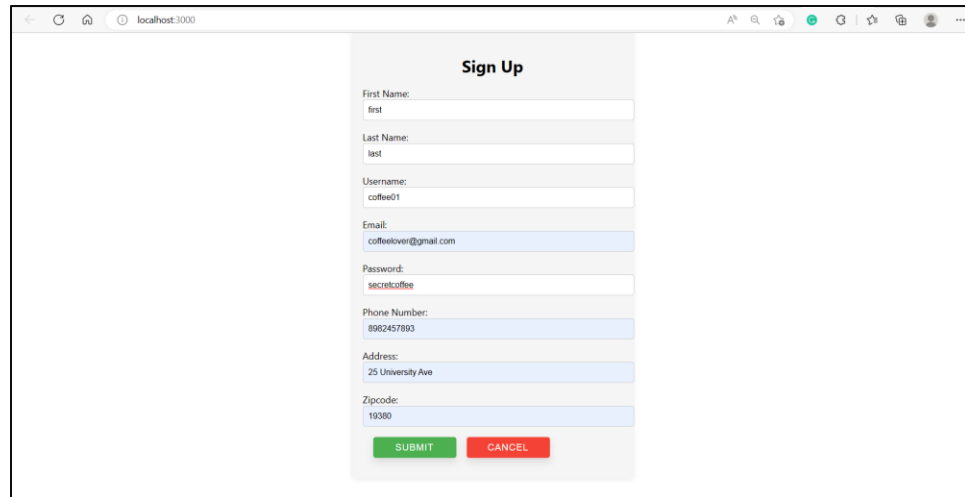
In order to build the database image, we had to navigate to the correct directory containing our Dockerfile for our database. We then utilized the Docker build command and named our image appropriately to reflect its usage. As with the other images we created, there was a need to map our image to the correct port in order for the Dockerfile to run properly. After trial and error, we were able to successfully interact with our database. We built our MySQL database image with the ability to insert, remove, and add tables as needed. This was a reassuring moment for our database image because we proved that we had properly implemented the base image and were able to add our All Ground Up database information to it. We were able to successfully create the database with our vision from chapter 1. The database was the first part of the project we completed since the other aspects of the project relied on the database.

2. The front-end of our application was successfully implemented using the React.js library architecture and JSON as an extension of JavaScript. We used JSON because of its ability to work well with the React libraries as well as the purpose of building an image for our web UI. After successfully completing a working website for the application locally, we needed to convert it into an image that could be managed in a cloud-based architecture. We used the base image of node:16-alpine in order to set the working directory and copy over the packages from the current web UI directory. This inherently copied over the package.json, packag-lock.json, and all other files in the current directory to build the image. Our web UI image still needed a base image of nginx:1.21.0-alpine which then changed the environment variable names to build out the rest of the image; the environment variables were changed so they could be turned into artifacts in Nginx. After successfully setting up the Nginx configuration for our web UI image we needed to properly run our Dockerfile build instructions.

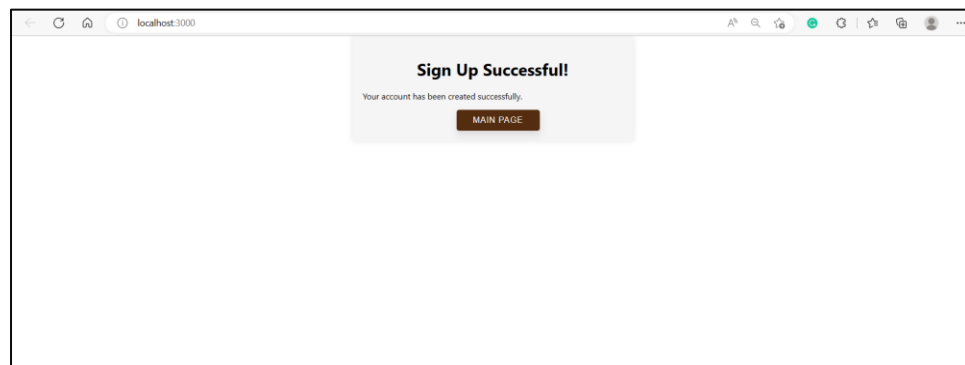
3.



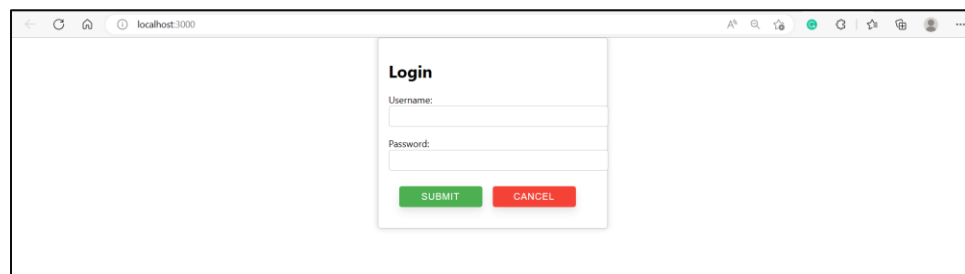
Main web application page.

A screenshot of a web browser window showing a 'Sign Up' form. The form is centered on a light gray background. It contains several input fields: 'First Name' (with 'first' entered), 'Last Name' (with 'last' entered), 'Username' (with 'coffee01' entered), 'Email' (with 'coffee01@gmail.com' entered), 'Password' (with 'secretcoffee' entered), 'Phone Number' (with '8962457893' entered), 'Address' (with '25 University Ave' entered), and 'Zipcode' (with '19380' entered). At the bottom of the form are two buttons: a green 'SUBMIT' button and a red 'CANCEL' button. The browser's address bar shows 'localhost:3000'.

Sign up page.



Sign up success page.

A screenshot of a web browser window showing a 'Login' form. The form is centered on a light gray background. It contains two input fields: 'Username' and 'Password'. At the bottom of the form are two buttons: a green 'SUBMIT' button and a red 'CANCEL' button. The browser's address bar shows 'localhost:3000'.

Login page.

Above are screenshots of the different web pages from the React.js application. The sign-up success page shows that the React.js application was able to successfully connect with the Python worker and MySQL database on the back end.

We were successful in containerizing our web UI image and still needed to add our Dockerfile to Kubernetes. After managing the container id and navigating our image to the right port we were able to successfully add our web UI Dockerfile to Kubernetes. This process took several attempts to achieve our desired goal. After we successfully containerized our web UI image, we knew we could map our web UI image. Our web UI image was successfully added to Kubernetes.

The hardest part of building the front-end portion of our web-based application was not building the site locally, that turned out to be the easier part. The harder part came with the task of figuring out the proper way of containerizing our website into our web UI container. This was the part of the project that took the most work because it was new to every one of our group members. After a substantial amount of research and guidance we were ultimately able to carry out the mapping of our web UI image to Kubernetes. Any website changes were easy after we had an understanding of the build process and what went into the containerization, implementation, and how Kubernetes was going to orchestrate our web UI image.

4. The hardest working part of our cloud-based web application is the Python worker. This is the component of our application that interacts with all of the other parts of the application. Without the Python worker, we would have no automation that makes the application run properly. This component interacts with the web UI and then makes necessary calls to the database for customer information, vendor information, menus, reviews, orders, order statuses, and payments. The Python worker is an integral part of the functionality of our application.

We started by writing a basic local script for the Python worker and then focused on figuring out how to get it working as a Dockerfile image that could communicate with the web UI and database images. The first problem we ran into was getting the Python script to connect with the database and pull/push information into the database as we wanted. We knew that we needed to use the mysql.connector commands to connect, but we needed to find the correct syntax.

```
connection = sqlconnector.connect(user='user', password='123', host='mysql-container', database='maindb')
```

Above is the screenshot of the line in our Python code that connected the script to the MySQL database. The username, password, and database are all defined in the MySQL Docker image.

The host was the part of the line that we ran into the most errors with. On our computers, the host was localhost since the MySQL database was running locally on our computers. With Docker, the MySQL database is running within a container. This means the hostname must come from the container, which can be defined when running it. Our hostname in our code is mysql-container.

```
cursor = connection.cursor()
cursor.execute('SELECT ZIP_ID FROM ZIP WHERE ZIP_CODE IN (%s)', (zipcode, ))
result = cursor.fetchone()
zip_id = result[0]
```

Once we connected to the MySQL database container, we were able to add and change data from the maindb database using SQL commands. In the screenshot above, we are showing one of the commands we are using to search the ZIP database. First, we connect to the database and then we execute our command and finally, we save the result to a local variable. We used SQL commands throughout our Python code to seamlessly make changes and calls to the database.

The next part we had issues with was connecting to our front-end user interface. This was a little more difficult to figure out as we decided to use Flask.

```
from flask import Flask, render_template, request
from flask_cors import CORS

app = Flask(__name__)
cors = CORS(app)
```

The first step we took was importing the necessary tools and creating our Flask application as shown above. We did not originally have the “flask_cors” library imported, but we ran into errors when trying to run the web user interface with Flask. CORS stands for cross-origin resource sharing which we needed to add so we could connect with the React.js library.

```
@app.route('/api/signup', methods=['POST'])
def signup():
    data = request.get_json()

    firstname = data['firstName']
    lastname = data['lastName']
    username = data['username']
    password = data['password']
    email = data['email']
    phone = data['phoneNumber']
    address = data['address']
    zipcode = data['zipcode']

    cursor = connection.cursor()
    cursor.execute('SELECT ZIP_ID FROM ZIP WHERE ZIP_CODE IN (%s)', (zipcode, ))
    result = cursor.fetchone()
    zip_id = result[0]

    cursor.execute('INSERT INTO CUSTOMER (FIRST_NAME, LAST_NAME, USERNAME, PASS, EMAIL, PHONE, CUSTOMER_ADDRESS, ZIP_ID) VALUES')
    connection.commit()
    cursor.close()

    return {'status': 'success'}
```

Above is a screenshot of our signup() function. Before the function, we use the @app.route command to route the URL to the specific function. The URL can then be called from the web user interface’s JavaScript files to perform the function. The Flask app needs to be running for the web user interface to be able to call the functions. The signup() function gets the data from the React.js in JSON format and then uses the data to search the database and insert data into the database. Finally, the function returns the status of the function to the web user interface file.


```

const handleSubmit = (e) => {
  e.preventDefault();
  const { username, password } = formData;
  const data = {
    username,
    password
  };

  fetch('http://127.0.0.1:5000/api/login', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(data)
  })
  .then(response => response.json())
  .then(data => {
    if (data.status === 'success') {
      console.log('Form submission successful!');
      setIsFormSubmitted(true);
    } else {
      console.error('Error submitting form:', data.error);
    }
  })
  .catch(error => {
    console.error('Error submitting form:', error);
  });
};

```

Above is the corresponding JavaScript file in React.js that fetches the signup() function from the Python worker through the Flask URL. The submit part of the JavaScript file can then receive the success status from the Python function and change the web page from the sign-up page to the sign-up success page. The Python worker successfully interacts with both the MySQL database and the web UI.

We were not able to create a Python worker to fit the magnitude of our vision from chapter 1. The main problem we ran into was the time restriction. By the time we got everything working together in the cloud, there was not a lot of time left to develop the Python worker fully. We were able to get the login and sign in pages fully functional with the database. Even though the Python worker did not reach all the parts of our vision from chapter 1, we were able to learn a lot about how Python can interact with different programming languages and applications.

5. To ensure the different components of the project were able to deploy together cohesively, we used Kubernetes to create the containers ran from images uploaded to a shared DockerHub repository. From there, each part needed a yaml file to create a cluster for the project to function within, as well as handle orchestration and communication between each container. The first yaml file we needed to create handled hosting for most of the processes and was necessary to properly deploy was the agu-deployment.yaml file as shown below:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: webui
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webui
  template:
    metadata:
      labels:
        app: webui
    spec:
      containers:
        - name: webui
          image: jettcanavarro/allgroundup:webui
          ports:
            - containerPort: 80

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - image: jettcanavarro/allgroundup:database
          name: mysql

```

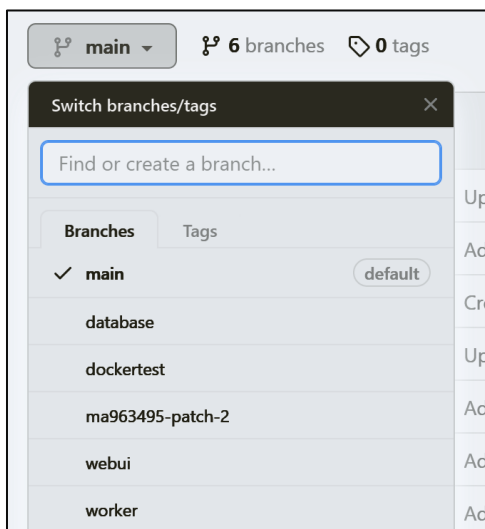
As seen from the code snippet, each part of the project is implemented by pulling the image from the DockerHub repository. The “kind” label shows whether a deployment, service, secret, etc. is created when the command “kubectl create -f agu-deployment.yaml” is run. The app label under template shows the name of the application being deployed. It was important to expose the internal port 80 for the web UI, as well as the internal port 3306 for the MySQL database. This port mapping was important for the containers to be able to communicate within the cluster.

```

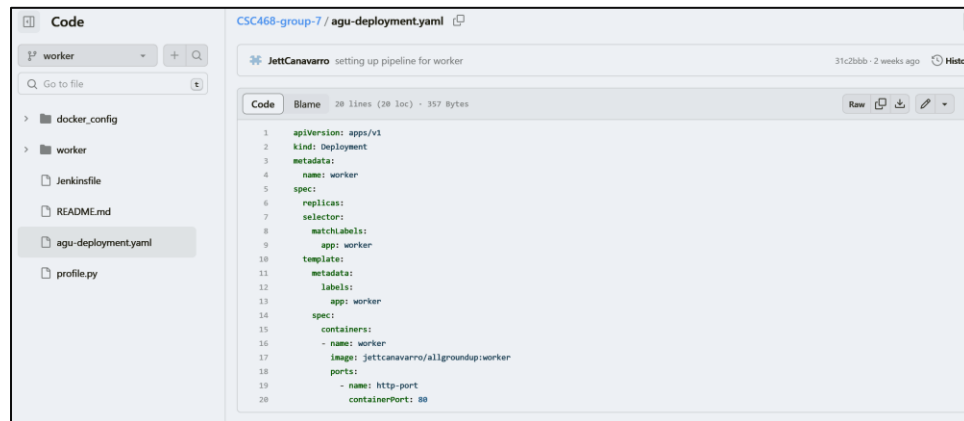
apiVersion: v1
kind: Service
metadata:
  name: webui-service
  labels:
    app: webui
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
      nodePort: 31000 ##might need to change
  selector:
    app: webui
---
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
    - port: 3306
  selector:
    app: mysql

```

The web UI and mysql applications required another type of yaml file which was labeled “ag-service.yaml”. The service file was necessary for the web UI to help balance the load of potential network traffic, likewise the file was necessary for the database to handle the number of queries processed. Another two yaml files were created in the form of a “mysql-secret.yaml” file and a “mysql-storage.yaml” file; the former’s purpose being to store secret data like usernames and passwords, and the latter to keep track of data in the case of a pod or container shutting down. From the command line, each pod was able to be created and the different yaml files were run in the default namespace. However, to get the project to run, we needed to have the ability to edit and update the worker, web UI, and database files while deployment required the help of Jenkins to set up a CI/CD pipeline.



The first step for setting the pipeline up was to create a separate branch for each part of the project within the GitHub repository, while maintaining all the current data within the main branch. The MySQL branch held a copy of the database folder from the main branch, as well as the yaml files, however each yaml file only held data related to the MySQL database. The same was done for the web UI branch and worker branch with their respective data. The figure below shows the worker branch with its yaml file.

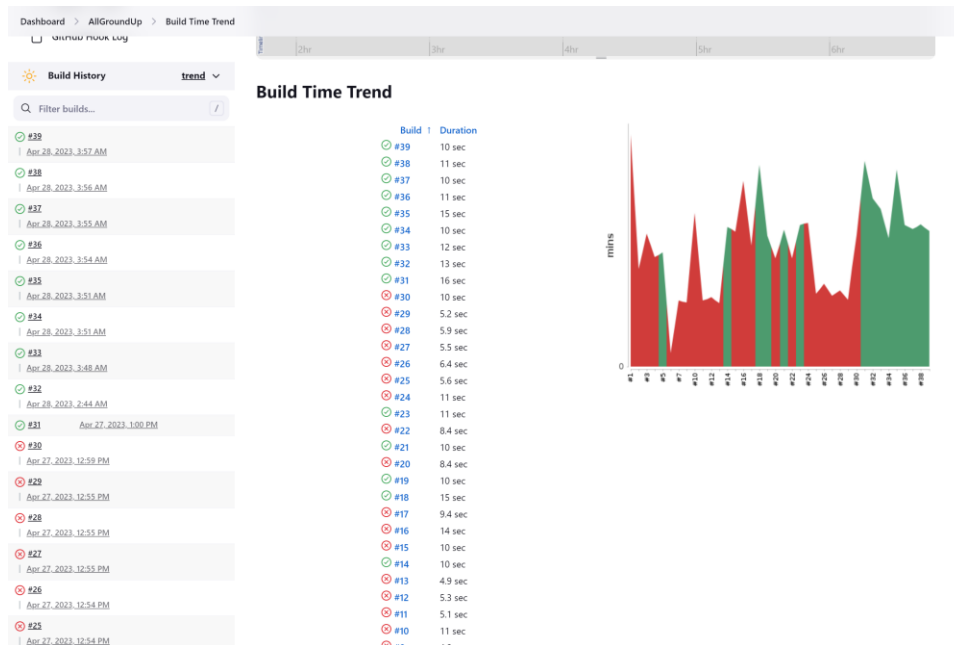


To set up the pipeline with Jenkins, each of these three new branches also needed a file called Jenkinsfile. These files contain instructions for Jenkins to interpret when the pipeline begins building and deploying each separate part of the project.

```
pipeline {
  agent none
  environment {
    docker_app = "worker"
    registry = "155.98.37.91"
    userid = "jc997541"
  }
  stages {
    stage ('Deploy') {
      agent {
        node {
          label 'deploy'
        }
      }
      steps {
        sshagent(credentials: ['cloudlab']) {
          sh "sed -i 's/REGISTRY/${registry}/g' agu-deployment.yaml"
          sh "sed -i 's/DOCKER_APP/${docker_app}/g' agu-deployment.yaml"
          sh "sed -i 's/BUILD_NUMBER/${BUILD_NUMBER}/g' agu-deployment.yaml"
          sh "scp -r -v -o StrictHostKeyChecking=no *.yaml ${userid}@${registry}:~/"
          sh "ssh -o StrictHostKeyChecking=no ${userid}@${registry} kubectl apply -f /users/${userid}/agu-deployment.yaml"
        }
      }
    }
  }
}
```

The code snippet above is the Jenkinsfile from the web UI branch of the project. The data environment section of the file shows the name of the Docker application, the Jenkins registry which controls and orchestrates the pipeline configuration, and the user id for the GitHub account that has the necessary webhook for the repository that holds the source code. The deploy stages finds the necessary yaml files and applies the Kubernetes command to deploy those files. Each time the pipeline is built, the Jenkinsfiles in the three GitHub branches should be called and that branch's yaml files will be built. The Kubernetes dashboard also relays the information showing

the state of each container, pod, deployment, service, secret, and storage. These are displayed in a figure in Chapter 4.



The current state of the group's Jenkins CI/CD pipeline does include a running web UI and database however the group was not able to successfully get the pipeline to build the worker application, even after getting the worker to run successfully with Docker. The main problem we had with Jenkins was not implementing the build and test stages within each Jenkinsfile. Without those stages, after so many builds, Jenkins did not reliably restart and redeploy each container after changes were made at the repository level. After making changes to the design and slogan for the web UI, the changes were not reflected in the running web UI container, even after removing the relevant deployments and data in the Kubernetes dashboard. In fact, at one point the web UI reverted to an earlier, once thought deleted, version of the main page. Each of these failures with the pipeline at the end only show what areas are important to study further and more closely to fully implement All Ground Up and continue to create a more robust cloud-based application.

Jessica Arriaga

(732)-853-6031 | www.linkedin.com/in/jessicaarriaga | JA958931@wcupa.edu | South Brunswick, NJ

Education

West Chester University of Pennsylvania, West Chester, PA
West Chester University Honors College, Class XXII

Bachelor of Science in Computer Science
Minors - Communication Studies, Criminal Justice, Spanish

Cumulative GPA: 4.0
Anticipated Graduation: May 2024

Experience

- | | |
|--|-------------|
| Intern at Essentia Advisory Partners | Summer 2022 |
| <ul style="list-style-type: none">• Worked with a lead developer to learn about projects assigned to different teams• Worked side by side with another intern to complete various coding projects• Developed introductory slide decks about commodities like Natural Gas and Power | |
| Intern at Essentia Advisory Partners | Summer 2021 |
| <ul style="list-style-type: none">• Completed research on different applications that the company utilizes• Created initial training material for these applications• Worked with developers to learn more about tools and services utilized | |

Involvement

- | | |
|---|-------------------------|
| Upsilon Pi Epsilon | Spring 2023 – current |
| Computer Science Honors Society | |
| Women in Computer Science Club | |
| Secretary | Fall 2021 – Spring 2022 |
| Vice President | Spring 2022 – current |
| Activity Clubs | |
| Art Club, Honors Student Association, Spanish Club, Stitched Together | |
| Sigma Delta Pi | Spring 2022 – current |
| Spanish Honors Society | |
| Institute for Cultural Competence & Inclusive Excellence – West Chester University | |
| Working towards certification through training and education workshops in the areas of diversity, equity, and inclusion | |
| Python Bootcamp | March 2021 |
| Women in Computer Science Club sponsored workshop; learned foundational concepts of the programming language, Python | |

Skills

Programming Languages: Java, Python, C
Relevant Courses: Data Structures and Algorithms, Software Security, Computer Systems, Intro to Cloud Computing

Jett Alexander Canavarro

Contact Info:

Email: jettcanavarro@gmail.com

Phone: (610) 420-3532

EDUCATION:

- West Chester University of Pennsylvania
 - Bachelor's of Science in Computer Science, Fall 2023
 - Master's of Science in Computer Science, Fall (accelerated program) 2024
- Delaware County Community College
 - Associate's Degree in Computer Science, December 2021

SKILLS:

- Adept knowledge of troubleshooting software and hardware issues
- Well-versed with JAVA, as well as Intermediate knowledge of C and Python.
- Capable knowledge of Mathematics, including Algebra, Calculus, & Differential Equations
- Proficient in Microsoft office applications (Word, Excel, PowerPoint, Outlook, etc.)
- Friendly and inclusive attitude & exceptional interpersonal skills, with great attention to detail
- Fast and decisive learner, which allows me to perform tasks autonomously

WORK EXPERIENCE:

- Wawa Inc. - Lead Customer Service Associate - 10/2018 to Present - West Chester
- Trained to work in all areas of the store including deli, coffee station, making beverages, point of sales, facilities, fuel court, and receiving & stocking orders
- Put in the role of talking to customers, training new employees, and taking on solo tasks, to ensure the store operates smoothly

RELEVANT COURSES:

- CSC 583 Topics in Computer Security
- CSC 545 Database Systems Concepts
- CSC 402 Software Engineering
- CSC 345 Programming Language Concept/Paradigms
- CSC 302 Computer Security

Mateen Ali

mateenali9001@gmail.com | (267) 575-9261

EDUCATION: West Chester University of Pennsylvania

Bachelor of Science in Computer Science

Expected Graduation: May 2023

Delaware County Community College

Associate of Science in Computer Science

Graduated 2022

CODING SKILLS:

- Intermediate Java
- Intermediate Python
- JavaScript
- Intermediate C
- MySQL

TECHNICAL SKILLS:

- GitHub
- MS-Office
- Linux
- Frameworks

PROFESSIONAL EXPERIENCE:

Samsung

Position: Samsung Experience Consultant (SEC)

August 2021- Present

- Help troubleshoot Samsung devices and fix any issues.
- Train employees and familiarize them with Samsung products.
- Introduce customers to the newest Samsung devices and their features.

Delaware County Community College

Position: Art Studios Assistant

Mar 2019 – Mar 2020

- Sorted and managed the products required by the students and instructors.
- Helped the students with certain uncomfortable products at the studios.
- Developed and executed plans to monitor standard process adherence.

Delaware County Community College

Position: Java Student Tutor

Aug 2018 – Mar2019

- Tutored struggling students individually and in small groups to reinforce java programming concepts.
- Created lesson materials, visual and digital presentations to supplement lesson plans.
- Used specialized teaching techniques to teach beginning java material in an understanding way.

Steven Coffey

(484)-653-9804 | sc986055@wcupa.edu | West Chester, Pa

Education

West Chester University of Pennsylvania

Bachelor of Science, Computer Science

Graduated: May 2023

Certification: Software Security

Concentration: Software Engineering

GPA: 3.4

Delaware County Community College

Associate of Science, Department of Computer Science

Graduated: May 2021

Certification: Mobile Web Programming

Completed: December 2020

GPA: 3.3, National Technical Honors Society, Dean's List

Relevant Skills

- Proficient in Java, Python, C/C++, SQL, HTML/CSS, JavaScript/jQuery/Json, Kubernetes
- Experience with the Agile and Scrum ideologies
- Database management
- SOLID Software design, Software Development Lifecycle Management

Experience

Waresoft Solutions Inc, Royersford, PA

July 2022-Present

Solutions Delivery Associate (Intern-fulltime)

- Used the Spring Boot framework to develop a website for Pfizer with multiple teams.
- Maintained updates for webpages using Java, SQL and various front-end/mark-up languages.
- Created webpage designs, flowcharts and layouts using Visio.
- Worked closely with the Solutions Architect to Design website functionality and logic.
- Fixed webpage errors and code logic using coding languages such as Java and JavaScript.

Charlestown Landscaping, Malvern, PA

June 2018-October 2021

Operations Manager

- Developed team members and procedures as well as coordinating crews.
- Created work schedules, customer relations, and customer price quotes.
- Logged employee hours, maintained equipment, and customer conflict resolution.
- Operated heavy machinery to design landscapes to customer specifications.

Dana Bellafiore Landscaping, West Chester, PA

March-November 2017

Landscape Foreman

- Installed landscape lighting, walkways and maintained customer's' properties.
- Provided quotes for future work and became a point of contact for customers.

CTDI, West Chester, PA

September 2014-April 2015

Warehouse Associate/ Product Development

- Developed new testing procedures with software engineers to create machines for telecommunication companies.
- Loaded software onto cable boxes, uploading TV guides, quality control and Maintained production timelines to ensure timelines were met.

Alejandro Almaraz

Coatesville · 610-457-5760

alejo9973@gmail.com

Responsible and motivated student ready to apply education in the workplace. Offers excellent technical abilities with software and applications, ability to handle challenging work, working as a team, and an outside the box thinker.

Education

Expected graduation date May 2024

Bachelor of Science, Computer Science, West Chester University of Pennsylvania

- Concentration in Cybersecurity
- Currently a member of the Cybersecurity Club
 - Computer Security Certificate from the NSA expected at the same time as graduation.

WORK Experience

Dates From May 2022 – August 2022

CYBERSECURITY CONSULTANT, SAP AMERICA, INC.

- Successfully designed and implemented a malware analysis software.
- Successfully mitigated a cyberattack which is calculated to have an impact of 100K USD
- Identified an unauthorized access of information via Bluetooth which is often called "bluesnarfing".

Skills

<ul style="list-style-type: none">• Managing Security Breaches• Telecommunications Systems• Monitoring Computer Viruses• Reporting and Documentation• Microsoft Office	<ul style="list-style-type: none">• Critical Thinking• Computer Skills• Problem Solution• Fluent in Spanish• Working as a Team
--	--

Certificates

- CompTIA Security+
- Computer Security Certificate from NSA (Expected May 2024)

Additional Information

- Java
- JavaScript
- Linux (Red Hat, Fedora)
- GitHub
- Unix
- OllyDbg
- IDA