Technical Report

CSC 468

Intro to Cloud Computing

Nest

Github Repository

Authors:
Brianna Ayala, Mike Collins,
David Rockwell, Zachary Smith,
and Sean Whoriskey

Submitted to:
Dr. Linh Ngo

May 8, 2023

Contents

1 Introduction

1.1 Vision and Purpose

Students at West Chester University are often seen absorbed in their personal devices, listening to music through headphones or earbuds. This behavior is indicative of a decrease in social interaction and creates barriers between individuals who might share common interests. Our vision for Nest is to create an innovative platform that encourages students to engage with one another, discover new music and help enrich the campus experience. The app's purpose is to break down barriers, promote social interactions, and foster a sense of community among students based on their shared love for music. As Nest and its features grow, we envision the app being used in a plethora of scenarios; from relaxed library study sessions to large scale campus events and parties.

1.2 What is Nest?

Nest is a dynamic song-sharing app designed to foster connections and social interactions among students through shared music interests. By integrating with Spotify, Nest allows users to view the currently playing songs of nearby users, facilitating conversations and connections among students with similar musical tastes. Currently, the target audience for Nest is university and college students seeking to engage with others on campus who share their music preferences. However, the app's potential for fostering connections extends beyond educational institutions and can be adapted for use in various social and professional settings. In the future, we plan to implement more location based music sharing features. These include location based collaborative playlists, a Nest map, proximity alerts, and much more.

The current state of Nest offers several key features designed to enhance the user experience and facilitate meaningful connections:

- *Spotify Integration:* Users can connect their Spotify accounts, allowing Nest to display their currently playing songs.
- *Real-time Location Updates:* Nest utilizes real-time location data to identify and display nearby users.
- *User-friendly Interface:* A visually appealing and intuitive UI allows users to easily navigate the app and see others in their Nest.
- *Music Discovery:* The app will provide a new way for users to discover new music by browsing other users' currently playing songs.

1.3 Technical Overview

In CSC 468 our group is tasked with developing an app and deploying it using cloud services. We employ Docker for containerization and Kubernetes for orchestrating the deployment, ensuring a smooth and efficient deployment process. Additionally, Jenkins and GitHub are utilized for our Continuous Integration and Continuous Deployment (CI/CD) process, allowing for seamless updates and improvements. Our prototype is deployed on CloudLab, an academic cloud service provider.

In this report, we will discuss the design, implementation, and deployment of Nest, as well as the challenges encountered and the potential for future enhancements.

2 System Design

2.1 Architecture Overview

The Nest app follows a modular architecture, which promotes scalability, maintainability, and ease of development. The system is composed of three main components: the web app, the backend API, and the database. The web app serves as the user interface, allowing users to interact with the platform. The backend API handles all the necessary processing and logic, while the database stores user information and song data.

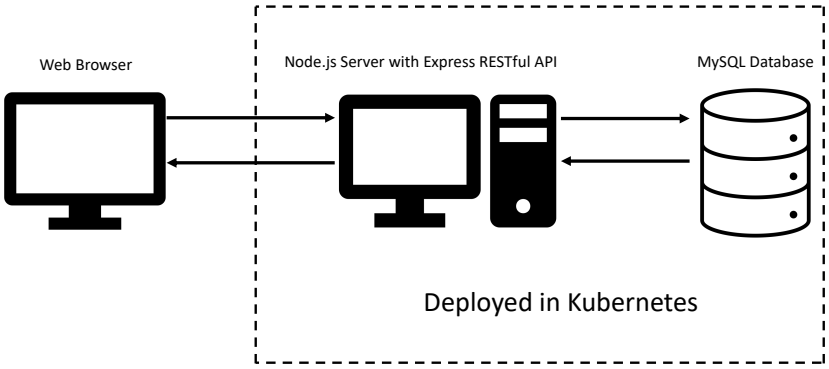

Figure 1: High-level architecture diagram of the Nest app

2.2 Web App

The web app is the primary interface for users to interact with Nest. It is currently built using Vanilla JavaScript. The web app offers the following key features:

- *User Authentication:* The application employs the OAuth 2.0 Authorization Code flow, enabling users to log in using their Spotify credentials. This approach ensures a secure and seamless experience while granting the application specific permissions to access the user's Spotify data. The resulting access and refresh tokens facilitate authorized API requests on behalf of the user and token refreshment as needed.
- *Song Display:* The web app displays the currently playing songs of nearby users in real-time, promoting engagement and discovery.
- *Responsive Design:* The web app's responsive design ensures a consistent and enjoyable user experience across various devices and screen sizes.

2.3 Database

The database is a crucial component of the Nest app, responsible for storing and managing user data, song information, and user interactions. We have chosen to use MySQL for the following reasons:

- *Structured Data:* MySQL allows us to store data in a structured format, making it easy to manage and query.
- *Scalability:* The MySQL can handle large amounts of data and scale as needed to accommodate the growing user base.
- *Data Integrity:* By enforcing constraints and relationships, MySQL ensures data integrity and consistency throughout the system.

The database schema currently consists of a users table and locations table. These tables store information such as user spotify information, song details, and geolocation data. The schema is designed to support efficient querying and data retrieval, enabling the app to provide a fast and responsive user experience.

```sql
CREATE TABLE location (
  locationID INT PRIMARY KEY AUTO_INCREMENT,
  latitude  FLOAT(10,6) NOT NULL,
  longitude FLOAT(10,6) NOT NULL
);

CREATE TABLE users (
  id INT NOT NULL AUTO_INCREMENT,
  Username VARCHAR(255) NOT NULL,
  SpotifyID VARCHAR(255) NOT NULL,
  Latitude FLOAT(10,6) NOT NULL,
  Longitude FLOAT(10,6) NOT NULL,
  CurrentSong VARCHAR(255),
  CurrentArtist VARCHAR(255),
  locationOld INT NOT NULL,
  locationNew INT NOT NULL,
  FOREIGN KEY (locationNew) REFERENCES location(locationID),
  PRIMARY KEY (id)
);
```

Figure 2: Database schema for the Nest app

3 Implementation

3.1 Development Environment

To ensure a consistent and efficient development process, we set up a flexible development environment for the Nest web app that allowed developers to work both locally and on CloudLab virtual machines. This environment includes the following key components:

- *Source Control:* We use Git for version control, allowing us to track changes, collaborate effectively, and maintain a clean and organized codebase. Our code is hosted on GitHub, which provides additional tools for project management and collaboration.
- *Pre CI/CD Pipeline Development:* Our development process was slow in the beginning of the semester as we had yet to configure a CI/CD pipeline. We would initially run the web app locally on our machines for testing and debugging purposes, connecting to the containerized MySQL database hosted on CloudLab virtual machines. We would then push changes to GitHub and test the application by building the web app Docker image on CloudLab virtual machines.
- *Post CI/CD Pipeline Development:* After configuring our CI/CD pipeline our development environment significantly improved. We were now able to push changes to the repository, have Jenkins test the code, build a new image and then deploy on Kubernetes.

3.2 Building Docker Images

Docker is used to create, deploy, and run applications in containers. Containers package an application with its dependencies, ensuring a consistent environment across different stages of the development pipeline. For the Nest web app, we use Docker to create images for both the Node.js server and the MySQL database:

- *Node.js Server:* We create a Dockerfile for the Node.js server, specifying the base Node.js image, installing required dependencies, and copying the application source code. The Dockerfile also defines the necessary environment variables and exposes the server's listening port.
- *MySQL Database:* For the database, we use the official mysql-server Docker image, which provides use our MySQL server. We configure the image with custom database name, user, and password, using environment variables.

With the Dockerfiles in place, we build the images using our ‘docker-compose.yaml‘ file and store them in Docker Hub for easy deployment.

```
FROM node:14-alpine

WORKDIR /webapp

COPY package*.json ./

RUN npm install

RUN npm install mysql2@latest --save

COPY . .

EXPOSE 3000

CMD ["npm", "start"]
```

Figure 3: Our Dockerfile for the web app

```
FROM mysql/mysql-server:latest

ENV MYSQL_ROOT_PASSWORD=$MYSQL_ROOT_PASSWORD

ENV MYSQL_DATABASE=$MYSQL_DATABASE
ENV MYSQL_USER=$MYSQL_USER
ENV MYSQL_PASSWORD=$MYSQL_PASSWORD

COPY ./init.sql /docker-entrypoint-initdb.d/
```

Figure 4: Our Dockerfile for the database

3.3 Kubernetes

Kubernetes is a container orchestration platform that automates the deployment, scaling, and management of containerized applications. We use Kubernetes to deploy the web with the help of CloudLab’s virtual machines, leveraging its features to ensure high availability, scalability, and efficient resource usage.

To deploy the Nest app with Kubernetes, we create configuration files (YAML files) that define the following resources:

- **Deployments:** A deployment specifies the desired state of a containerized application, such as the Docker image to use, the number of replicas, and update strategy. We create deployments for both the Node.js server and the MySQL database.
- **Services:** A service defines a set of rules for accessing a deployed application within the Kubernetes cluster or from the outside. We create a service for the Node.js server, exposing it to external traffic via a load balancer.

```
containers:
  - name: webapp
    image: REGISTRY:443/DOCKER_APP:BUILD_NUMBER
    ports:
      - name: http-port
        containerPort: 3000
    command:
      - /bin/sh
      - -c
    args:
      - export HEAD_NODE_HOSTNAME=$(cat /data/head_node_hostname.txt) &&
        exec npm start
    volumeMounts:
      - name: headhostname
        mountPath: /data
volumes:
  - name: headhostname
    emptyDir: {}
  - name: headhostname-script
    configMap:
      name: headhostname-script
```

Figure 5: Part of our deployment file for the web app

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  type: NodePort
  ports:
    - port: 3000
      targetPort: 3000
      nodePort: 30088
  selector:
    app: webapp
```

Figure 6: The service file for the web app

With the configuration files in place, we use the kubectl command-line tool to interact with the Kubernetes cluster and apply the defined resources:

- **Applying Configurations:** We use the kubectl apply command followed by the -f flag and the path to the configuration file to create or update the specified resources in the cluster. For example, to apply the deployment and service configurations for the Node.js server and the MySQL database, we execute the following commands:

```
kubectl apply -f webapp.yaml
kubectl apply -f webapp-service.yaml
kubectl apply -f database.yaml
kubectl apply -f database-service.yaml
```

- **Monitoring Cluster Status:** The kubectl command also allows us to monitor the status of the cluster and its resources. For example, we can use kubectl get pods to list all running pods, kubectl get deployments to display the status of deployments, and kubectl get services to view the available services and their associated IP addresses.

- **Scaling the Application:** Kubernetes enables us to easily scale the application to meet changing demand. We can update the number of replicas specified in the deployment configuration file and reapply the configuration using kubectl apply.

- **Kubernetes Dashboard:** In addition to the kubectl command-line tool, we deploy the Kubernetes Dashboard to provide developers with a user-friendly graphical interface for managing the cluster resources. The Kubernetes Dashboard is a web-based UI that allows users to interact with the Kubernetes cluster, offering an alternative to the command-line interface and improving overall usability.

  To deploy the Kubernetes Dashboard in our development environment, we use a script named launch_dashboard.sh. This script contains the following commands:

  - The first command applies the dashboard-insecure.yaml configuration file, which sets up the Kubernetes Dashboard without SSL/TLS encryption. This configuration is suitable for our development and testing purposes.

    ```
    kubectl apply -f /local/repository/cloud/k8s/dashboard/
        dashboard-insecure.yaml
    ```

  - The second command applies the socat.yaml configuration file, which deploys a socat container for forwarding network traffic. This is required for accessing the Kubernetes Dashboard from outside the cluster:

    ```
    kubectl apply -f /local/repository/cloud/k8s/dashboard/socat.yaml
    ```

  - The third command patches the kubernetes-dashboard service, specifying the desired node port (30082) to expose the Dashboard to external traffic:

    ```
    kubectl patch service kubernetes-dashboard -n kubernetes-
        dashboard --type='json' --patch='[{"op": "replace", "path
        ": "/spec/ports/0/nodePort", "value":30082}]'
    ```

  - The script then sleeps for 5 seconds to allow for the changes to take effect, before executing the final command to display the kubernetes-dashboard service information, including the external IP address and port required to access the Dashboard:

    ```
    sleep 5
    kubectl get svc --namespace=kubernetes-dashboard
    ```

  After running the launch_dashboard.sh script, developers can access the Kubernetes Dashboard via a web browser, providing a convenient and user-friendly way to interact with the Kubernetes cluster and manage the Nest web app deployment.

By leveraging Kubernetes and its powerful features, we're able to efficiently deploy, manage, and scale the Nest web app on CloudLab's virtual machines, ensuring a robust and responsive application.

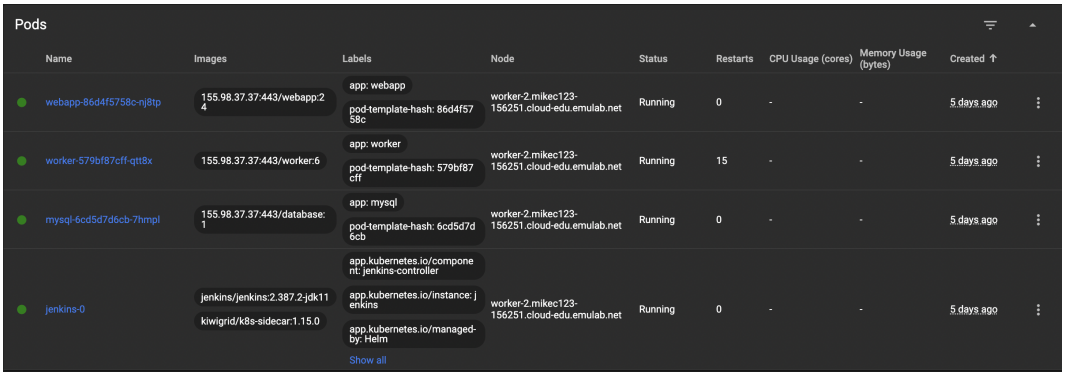

Figure 7: Pods displayed via Kubernetes Dashboard

4 Cloud Deployment

In this section, we discuss the deployment of the Nest web app on the cloud, our cloud provider, the setup of the infrastructure, and networking considerations.

4.1 Cloud Provider

For our deployment, we are using CloudLab, a research-oriented platform that caters to various experimental and educational projects. CloudLab provides virtual machines, storage, and networking solutions, enabling us to implement our desired architecture effectively. CloudLab also provides a flexible, and scalable infrastructure that allows researchers and educators to customize cloud-based systems and applications in a controlled environment. This enables us to implement our desired architecture effectively.

4.2 Infrastructure Setup

The infrastructure setup is automated using a combination of scripts and configuration files, which allows for easy provisioning and management of resources. Below is a portion of a python script used to request nodes on CloudLab and install Docker, Kubernetes, and Helm.

```
num_nodes = 3
for i in range(num_nodes):
    if i == 0:
        node = request.XenVM("head")
        bs_landing = node.Blockstore("bs_image", "/image")
        bs_landing.size = "500GB"
    else:
        node = request.XenVM("worker-" + str(i))
    node.cores = 4
    node.ram = 4096
    node.routable_control_ip = "true"
    node.disk_image = "urn:publicid:IDN+emulab.net+image+emulab-ops
       :UBUNTU20-64-STD"
    iface = node.addInterface("if" + str(i))
    iface.component_id = "eth1"
    iface.addAddress(pg.IPv4Address(prefixForIP + str(i + 1), "
       255.255.255.0"))
    link.addInterface(iface)

    # setup Docker
    node.addService(pg.Execute(shell="sh", command="sudo␣bash␣/
       local/repository/cloud/install_docker.sh"))
    # setup Kubernetes
```

```
node.addService(pg.Execute(shell="sh", command="sudo␣bash␣/
    local/repository/cloud/install_kubernetes.sh"))
node.addService(pg.Execute(shell="sh", command="sudo␣swapoff␣-a
    "))
node.addService(pg.Execute(shell="/bin/sh", command="sudo␣apt␣
    update"))
if i == 0:
  # install Kubernetes manager
    node.addService(pg.Execute(shell="sh", command="sudo␣bash␣/
        local/repository/cloud/kube_manager.sh␣" + params.userid
         + "␣" + str(num_nodes)))
  # install Helm
    node.addService(pg.Execute(shell="sh", command="sudo␣bash␣/
        local/repository/cloud/install_helm.sh"))
else:
    node.addService(pg.Execute(shell="sh", command="sudo␣bash␣/
        local/repository/cloud/kube_worker.sh"))
```

4.3 Networking

Networking is essential for ensuring the accessibility and performance of our web app. We have implemented various networking components and configurations for Docker, Kubernetes, and VMs to create a seamless and secure environment for our application.

- **Docker Networking:** Within our Docker containers, we use bridge networks to enable communication between the Node.js server and the MySQL database. This network isolation helps improve security and manageability while allowing the containers to exchange data efficiently.
- **Kubernetes Networking:** Kubernetes provides several networking components to manage communication between services and pods within the cluster. Our Kubernetes networking setup includes the following:
  - *Webapp Service:* A Kubernetes Service for our Node.js server, named 'webapp'. This service is of type 'NodePort', which exposes the application on each node's IP at a static port. The service listens on port 3000, targets port 3000 in the server container, and maps to a node port 30088, making it accessible both internally and externally.
  - *MySQL Service:* A Kubernetes Service for our MySQL database, named 'mysql'. This service is of type 'ClusterIP', making it reachable only within the Kubernetes cluster. It listens on port 3306 and targets port 3306 in the database container, allowing secure communication between the web app and the database within the cluster.
  - *Kubernetes Dashboard Networking:* The Kubernetes Dashboard is a service of type 'NodePort', which makes the dashboard accessible externally via each node's IP at a specific port. In our configuration, the Dashboard service is exposed at node port 30082.
  - *Jenkins Sercice:* To expose Jenkins externally, we create a service of type 'NodePort'. This service exposes Jenkins on each node's IP at port 30000, making it accessible for developers to manage and monitor the CI/CD pipeline.
- *VM Networking:* On the CloudLab platform, our virtual machines are connected through a virtual network. This network setup allows our VMs to communicate with each other and the internet, ensuring efficient and secure data exchange between the various components of our infrastructure.

Through these networking components and configurations, we ensure that our web app operates efficiently, securely, and remains accessible to users across different networking layers.

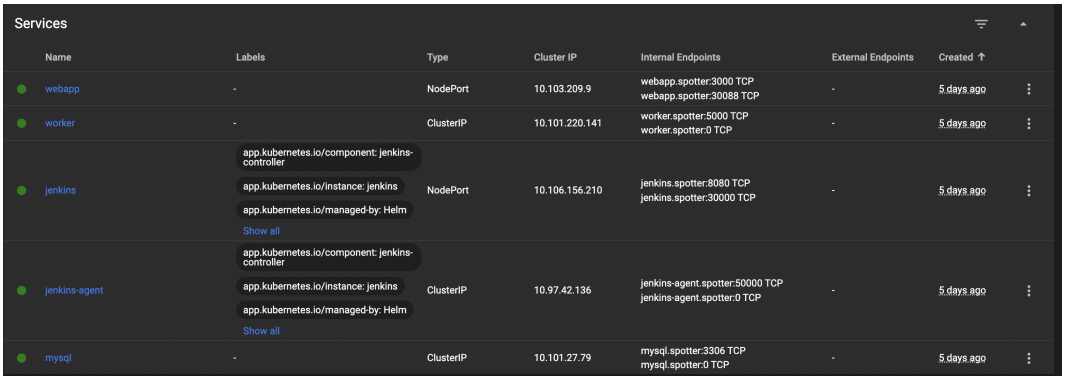

Figure 8: Services displayed via Kubernetes Dashboard

5 Continuous Integration and Continuous Deployment

We use Jenkins as our continuous Integration and continuous deployment (CI/CD) tool, which automates the building, testing, and deployment of our web application. Jenkins pipelines are configured using a Jenkinsfile that defines the stages and steps required for the entire process.

5.1 Pipeline Configuration

Our Jenkins pipeline consists of three main stages: Test, Publish, and Deploy. The pipeline is defined in a Jenkinsfile, which is a text file containing the pipeline's configuration written in Groovy.

In the *Test* stage, the pipeline runs our application's tests using a a specific container within the Kubernetes cluster. This ensures that any code changes are tested before being built and deployed.

In the *Publish* stage, the pipeline builds a Docker image of our web application and pushes it to our private Docker registry. This enables us to version our application images and deploy specific versions when needed.

In the *Deploy* stage, the pipeline updates the Kubernetes configuration files with the appropriate image and registry information, then applies these changes to our Kubernetes cluster. This ensures that our web application is deployed with the latest changes.

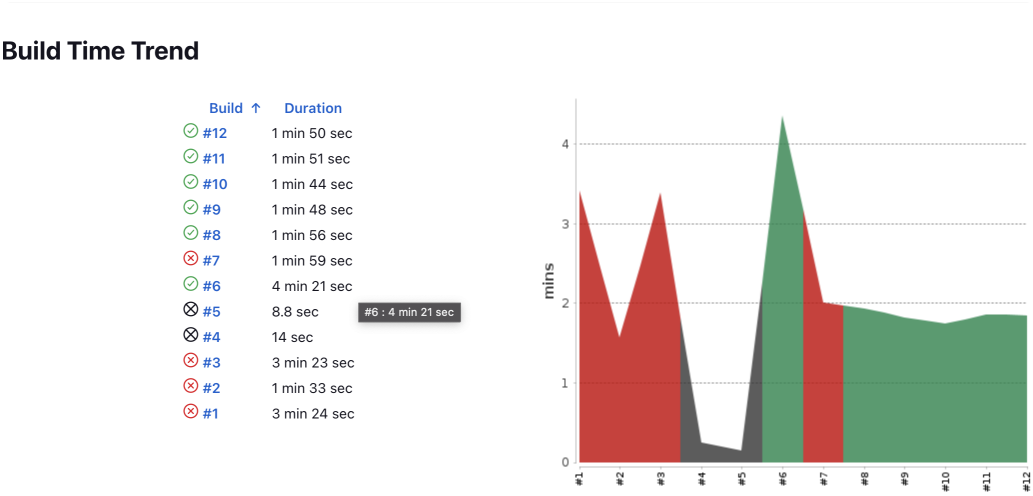

Figure 9: Display of multiple Jenkins builds

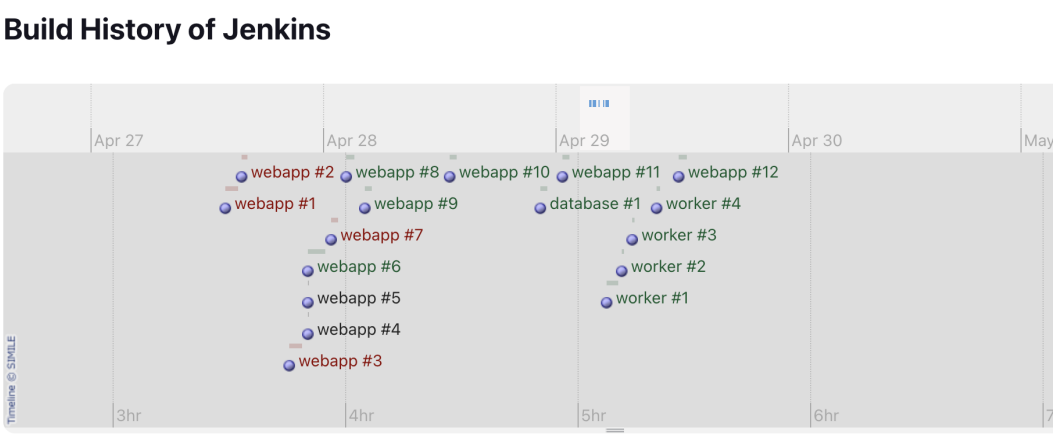

Figure 10: Multiple connected Jenkins components

5.2 Private Docker Registry Setup

We use a private Docker registry to store our web application's Docker images. The registry is set up using a script that generates the necessary SSL certificates, creates an authentication file, and configures the registry's IP address. By using a private registry, we can control access to our Docker images and ensure that only authorized users can push and pull images.

The registry's SSL certificates are generated using OpenSSL and stored in a shared directory. The authentication file is created using the 'htpasswd' tool and stored in the shared '/opt/keys/auth' directory. The registry's IP address is configured in the 'san.cnf' file, which is used to generate the SSL certificates.

```
#!/bin/bash

# create the certs in the shared /keys directory
mkdir -p /opt/keys/certs
cd /opt/keys/certs
rm -f domain.*
openssl genrsa 1024 > domain.key
chmod 400 domain.key
cp /local/repository/cloud/registry/san.cnf.template san.cnf

# In some cases we might have to switch to this to get ip address.
ip_address=$(nslookup $(hostname -f) | grep Server | awk -F ' ' '{printf $2}')

#ip_address=$(ip addr | grep eth0| awk -F ' ' '{print $2}' | awk -F '/' '{print $1'} | tail -n 1)

sed -i "s/IPADDR/${ip_address}/g" san.cnf
openssl req -new -x509 -nodes -sha1 -days 365 -key domain.key -out domain.crt -config san.cnf

# create login/password in the shared /keys directory
mkdir -p /opt/keys/auth
cd /opt/keys/auth
docker run --rm --entrypoint htpasswd registry:2.7.0 -Bbn admin registry > htpasswd

# create a template subdirectory to be mounted to pods
mkdir -p /opt/keys/certs.d/${ip_address}:443
cp /opt/keys/certs/domain.crt /opt/keys/certs.d/${ip_address}:443/ca.crt
```

Figure 11: This script handles SSL configuration for our registry

6 Challenges

During the development of our project, we faced several challenges that impacted our progress. In this section, we will discuss these challenges and the solutions we implemented.

6.1 API Integration

Integrating the Geolocation and Spotify APIs presented numerous difficulties, which acted as bottlenecks in our project's progress. These APIs required extensive research and understanding

in order to be utilized effectively.

6.1.1 Spotify API

The Spotify API enforces a 25-user limit for each app and requires a valid redirect link for authorization. Since the redirect link changes every time we launch a new experiment, we could not automate this process. Instead, we had to manually enter our redirect URL into the Spotify developer dashboard to ensure correct authorization. Once the app is fully developed we will be able to request full use of the Spotify API. We will also have migrated from CloudLab and will have a consisten redirect URI. We also faced challenges in developing the Authorization Code flow for the API. We made use of Spotify's developer GitHub that provided examples for us and eventually developed a suitable authorization flow. The flow includes redirecting users to Spotify for login, generating and validating a state parameter to prevent CSRF attacks, and exchanging a temporary authorization code for access and refresh tokens upon successful authorization. The server then retrieves the user's profile information using the access token, which allows it to make authorized API requests on the user's behalf.

Figure 12: A plethora of redirect URIs in our app's Spotify Developer account

6.1.2 Geolocation API

The Geolocation API requires a secure website to access location data on a non-locally hosted site. Due to the changing hostname for each new cloud project, we were concerned about automating the SSL certificate setup within the project's timeframe. To bypass this issue, we used Google Chrome Beta tools to whitelist the website and access geolocation data.

6.2 Database Connection

We encountered issues connecting our web app to the database, as the database container took longer to initialize than the web app. To resolve this problem, we:

- Added a "depends-on" statement for the "webapp" service in our docker-compose.yaml file to ensure the database initializes before the web app.
- Implemented a loop in the web app to continuously attempt a connection to the database until it is fully initialized.

6.3 Hostname Configuration

Setting up the hostname for our web app proved challenging. We resolved this issue by creating a script called "hostname.sh" that replaces any instance of the words "hostname" or "mysql" with the actual hostname of our machine. This streamlined our project instantiation on Docker. As the project grew and we deployed it on Kubernetes and integrated the Spotify API, the dynamic hostname management became more complex. To address this issue, we implemented a DaemonSet to modify the web app code, updating the redirect URI to include the current hostname while maintaining Kubernetes' automatic configurations.

6.4 CI/CD Pipeline

We encountered multiple challenges while setting up our Continuous Integration and Continuous Deployment (CI/CD) pipeline, primarily due to unfamiliarity with the technology. The key issues we faced included:

- Configuring the Jenkins podTemplates to properly launch and manage the required build agents for our CI/CD pipeline.
- Ensuring that our Node.js application tests were executed correctly and reliably within the CI/CD pipeline using Jest.
- Making appropriate changes to our Kubernetes deployment configurations to guarantee seamless and automated deployment of our application after successful testing and build processes.

Addressing these challenges demanded iterative improvements to our pipeline configuration and thorough testing to establish a smooth and automated deployment process that met our project's requirements.

7 Future Improvements

Our project has the potential for various improvements and enhancements in different areas, including the web app, infrastructure and deployment optimizations, and the addition of new features.

7.1 Web App Enhancements

Currently, our web app is built using vanilla JavaScript, which has limitations in performance and maintainability. To improve the web app, we plan to refactor it using React, a popular library with extensive resources for developers. Alternatively, we may opt for Angular, a comprehensive framework that can provide more clarity in terms of best practices and facilitate easier scaling of the application. Moreover, we aim to redesign the database structure to more efficiently store a broader range of user information.

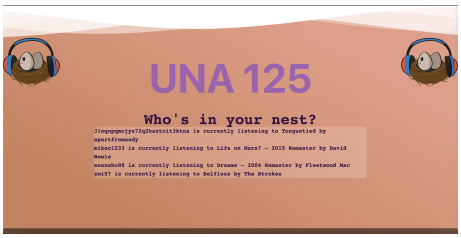

Figure 13: Nest's current UI

Figure 14: When a user leaves a Nest their information in the app ceases sharing

7.2 Infrastructure and Deployment Optimizations

While we are generally satisfied with our overall infrastructure, we would like to enhance it by using more environment variables to ensure a secure environment. Additionally, we plan to integrate ArgoCD into our CI/CD pipeline. This integration will allow us to adopt GitOps principles,

enabling us to manage infrastructure and deployments declaratively through version-controlled manifests. By adding a separate manifest repository for ArgoCD to monitor, we can streamline the deployment process, reduce human error, and maintain a clear understanding of changes.

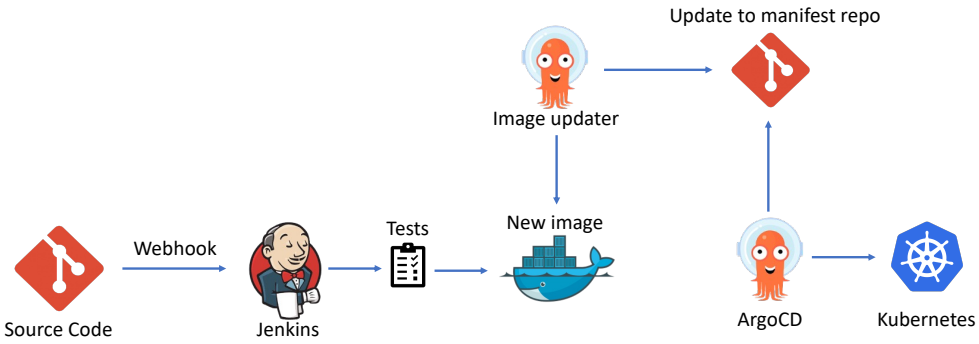

Figure 15: CI/CD Pipeline with ArgoCD and manifest repository.
Based off of work from Abhishek Veeramalla

7.3 Additional Features

As we progress, we plan to introduce new features to our application to enhance user experience and engagement. Some potential features include:

- A dynamic map, similar to Snapchat, displaying users and their associated songs based on their location.
- Location-based playlist curation, allowing users to discover and enjoy music relevant to their surroundings.
- Photo-based sharing, enabling users to share images associated with specific songs or locations.
- Social interaction features, such as friending, liking, commenting, and more, to foster a sense of community among users.

These improvements and additions will help ensure our application remains engaging, user-friendly, and scalable while meeting the evolving needs of our users.

8 Conclusion

Over the spring semester we have successfully created a proof of concept for our location-based music sharing application. We developed functionality to allow users to discover new music based on others' geolocation. The development process involved overcoming various challenges, such as working with multiple APIs, addressing cloud configuration related issues, and establishing a CI/CD pipeline that integrates Jenkins, Kubernetes, and Docker.

We have utilized Kubernetes for container orchestration and deployed the application using a private Docker registry and a public Dock Hub registry. Furthermore, we implemented a Jenkins based CI/CD pipeline, enabling smooth integration and deployment. We successfully implemented

the pipeline for our web app, MySQL database, and a separate worker node that will be used in the future for scaling purposes. Despite challenges in configuring Jenkins, we ultimately achieved a functional pipeline.

Moving forward, we plan to improve several aspects of the project, including a large refactoring of the web app, enhancing infrastructure and deployment processes, and adding new features to enrich user engagement and experience. These improvements will help ensure our application remains secure, maintainable, and meets user needs.

In conclusion, we have successfully achieved all technical requirements outlined for CSC468.