WEB-BASED MUSIC REVIEW APPLICATION WITH DEVOPS

CS 816 - Software Production Engineering - Major Project

Vignesh Bondugula
IMT2019092
vignesh.bondugula@iiitb.ac.in

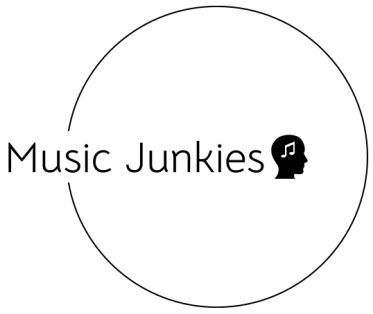
Abhinav H Kamath
IMT2019001
abhinav.kamath@iiitb.ac.in

Contents

1 Introduction

1.1 Project Idea

MusicJunkies is a responsive music review/ recommendation web application similar to letterboxd which is for movies. Our application allows users to review and keep track of their favorite music albums. It is a portal for music lovers to review and rate music.

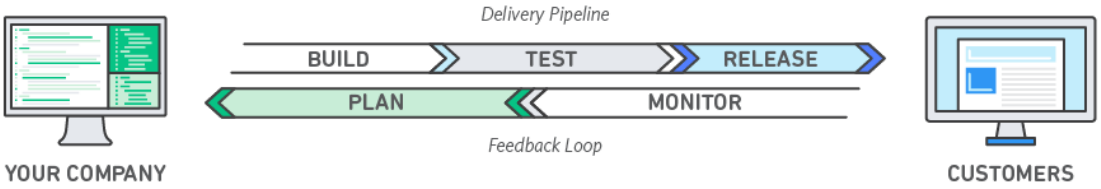


1.2 Project Context

The motivation for this project arrives from the fact that there are very few music review applications. There are a lot of Movie review applications. With this app we want to bring in ratings and review for music, with enough data we could later build a strong recommendation system. This recommendation system has became essential for many music application such as spotify users to find the music that suits them.

1.3 DevOps

This project uses a complete DevOps pipeline in all the steps mentioned above. But what exactly is DevOps. Under a DevOps model, development and operations teams are no longer "siloed." Sometimes, these two teams are merged into a single team where the engineers work across the entire application lifecycle, from development and test to deployment to operations, and develop a range of skills not limited to a single function. In our project too we are taking care of the code from the Development to Operations stage.

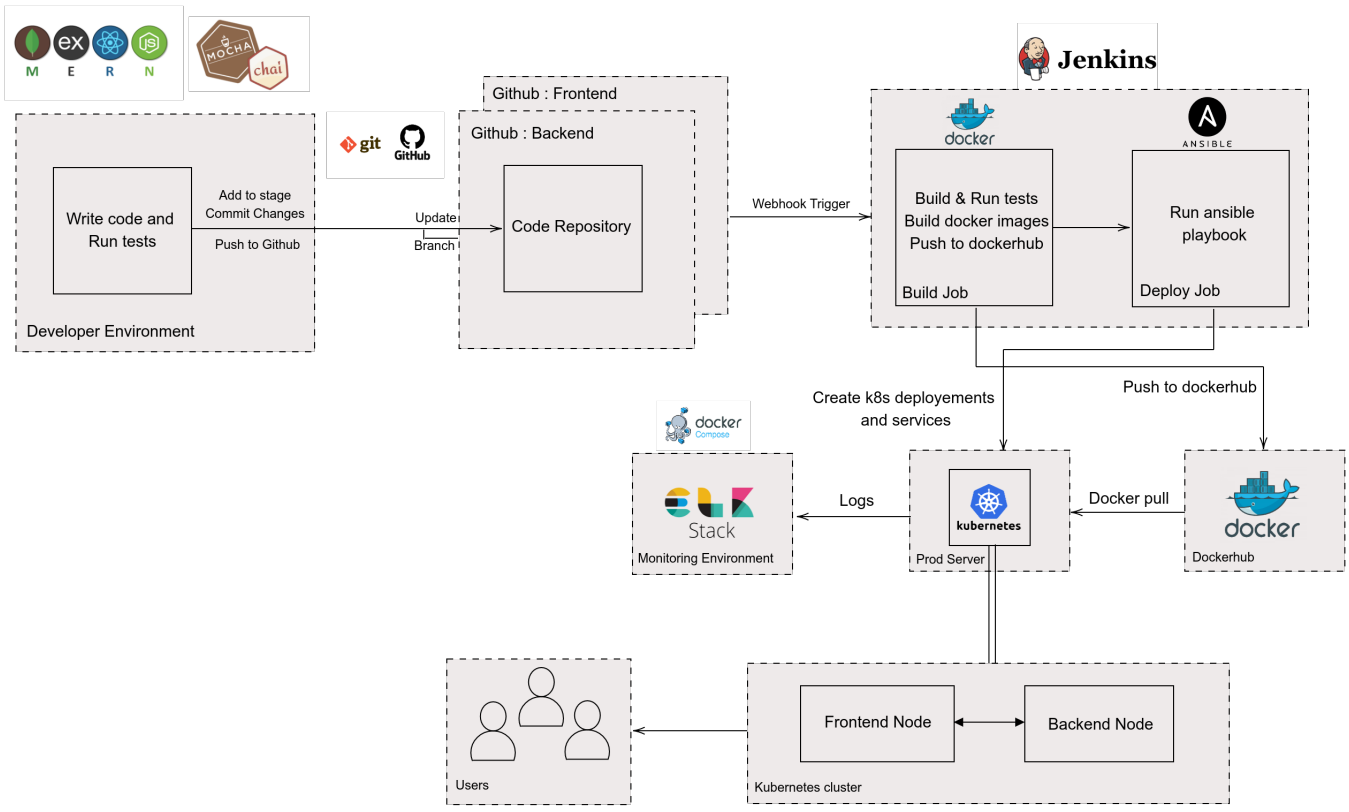


As we can infer from the image, the steps involved are Build, where we develop the code; Test, where we unit test the code; Release, where we publish the code and deploy it to production; Monitor, where we check the logs of the application continuously to fetch for any discrepancies.

1.4 TechStack

- Frontend - React
- Backend - Node, Express
- DB - MongoDB
- Containerisation - Docker
- CI/CD - Jenkins
- Automation - Ansible
- Orchestration - Kubernetes, Docker-compose
- Testing - Chai/Mocha
- Monitoring - Elastic Search, Logstash and Kibana
- VCS - Git

1.5 Pipeline



The above image shows the architecture for the entire application. There are 2 Github repositories, one for the Frontend and one for the Backend. CI is done independently on both the repositories whenever a push is done using Jenkins. The CI pipeline runs the tests and builds the docker image and pushes it to

Dockerhub. Once the build job is over, a deploy job is called which invokes Ansible. The playbook then creates deployements and services for the k8 nodes. Within our Cluster, there are 2 nodes, one for frontend service and another for the backend. Our database is already hosted on the cloud.

1.6 Links

Frontend Github Link - `https://github.com/Abhinav-Kamath/review`
Backend Github Link - `https://github.com/VigneshBondugula/MusicJunkies-Backend`
DockerHub Frontend Link - `https://hub.docker.com/repository/docker/vigneshbondugula/mj-frontend`
DockerHub Backend Link - `https://hub.docker.com/repository/docker/vigneshbondugula/mj-backend`

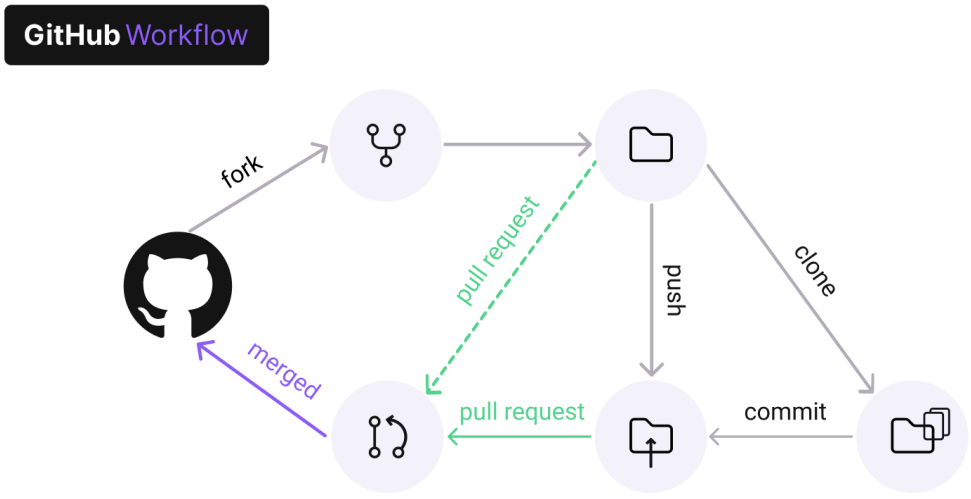
2 Source Control Management

The basic goal of Source Control Management is to keep the software in the current state which allows the developers to work on new version for further development or repair. In this project we used git and github for the version control.

2.1 Git & Github

Git is an open source distributed version control system, it is a tool to manage project source code history. It's built to manage everything from little to huge tasks quickly and effectively. Git is one of the most widely-used popular version control system in use today.

GitHub, Inc. is a provider of Internet hosting for software development and version control using Git. It offers the distributed version control and source code management (SCM) functionality of Git, plus its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, continuous integration and wikis for every project. Having a distributed architecture, Git is an example of a DVCS (hence Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.



Installation:

$ sudo pacman -Sy git

Verify the installation by using the below command,

$ git –version

git version 2.39.1

Creating the repository:

SignUp into github and create public repositories one for backend and one for frontend. Add a readme file and license if needed. **Create a local repository and add our remote repository**

$ mkdir SPE

$ cd SPE

$ git init

```
vikky@vigneshb~/Desktop/SPE$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/vikky/Desktop/SPE/.git/
```

$ git remote add origin https://github.com/VigneshBondugula/MusicJunkies-Backcend.git

Code and Commit

Start coding. The changes made are unstaged at start. We have to stage the changes first then commit them with an appropriate message. Then push the committed code to added remote repository. The below shows the image of unstaged changes. git add command adds the changes to staging area which can be commited.

$ git status

```
vikky@vigneshb~/Documents/8th Sem/SPE/Major Project/server$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Controllers/UserController.js
        modified:   logs/backend.log
        modified:   package.json
        modified:   server.js

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .mocharc.json
        test/

no changes added to commit (use "git add" and/or "git commit -a")
```

$ git add .

```
vikky@vigneshb~/Documents/8th Sem/SPE/Major Project/server$ git add .
vikky@vigneshb~/Documents/8th Sem/SPE/Major Project/server$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   .mocharc.json
        modified:   Controllers/UserController.js
        modified:   logs/backend.log
        modified:   package.json
        modified:   server.js
        new file:   test/users.test.js
```

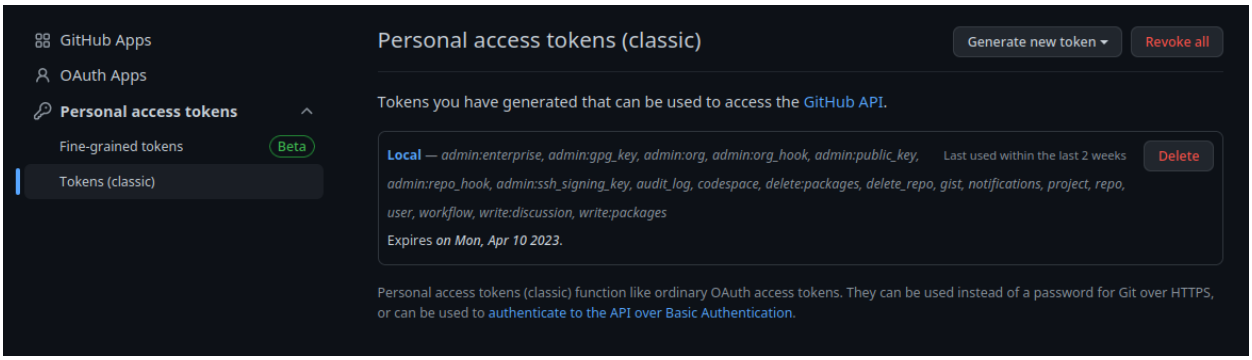
$ git commit -m 'message'

```
vikky@vigneshb~/Documents/8th Sem/SPE/Major Project/server$ git commit -m 'Added tests'
[master 7452a0a] Added tests
 6 files changed, 171 insertions(+), 7 deletions(-)
 create mode 100644 .mocharc.json
 create mode 100644 test/users.test.js
```

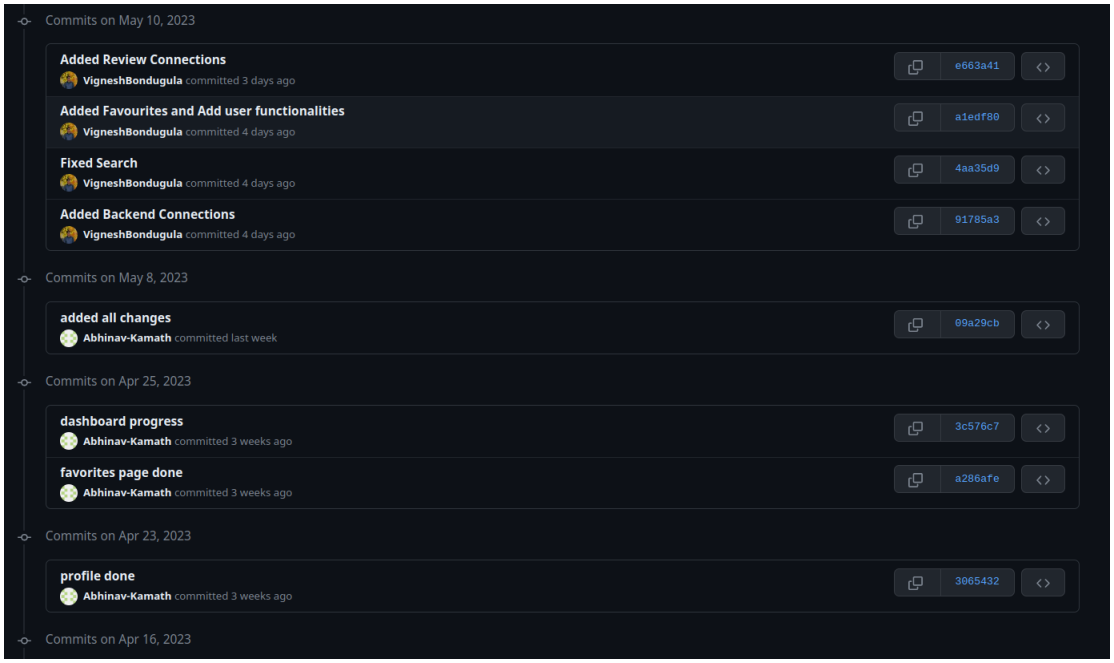
$ git push -u origin master'

```
vikky@vigneshb~/Documents/8th Sem/SPE/Major Project/server$ git push
Username for 'https://github.com': VigneshBondugula
Password for 'https://VigneshBondugula@github.com':
Enumerating objects: 18, done.
Counting objects: 100% (18/18), done.
Delta compression using up to 8 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (11/11), 2.49 KiB | 1.24 MiB/s, done.
Total 11 (delta 5), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (5/5), completed with 5 local objects.
To https://github.com/VigneshBondugula/MusicJunkies-Backend.git
   71ffcb6..7452a0a  master -> master
```

It will ask your github credentials for the authentication. Enter your github username and for password we need to create a personal access token giving all the necessary permissions.



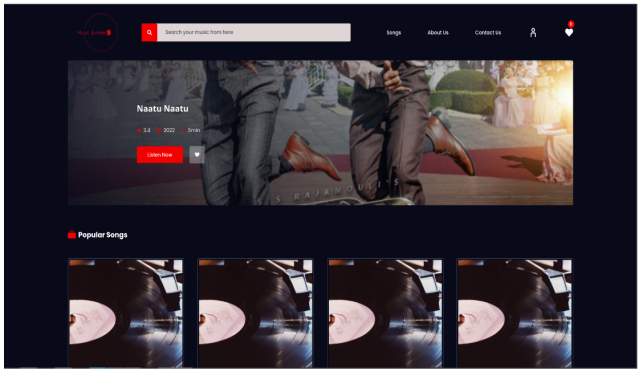
We can also see all the commits made till now in the github UI,



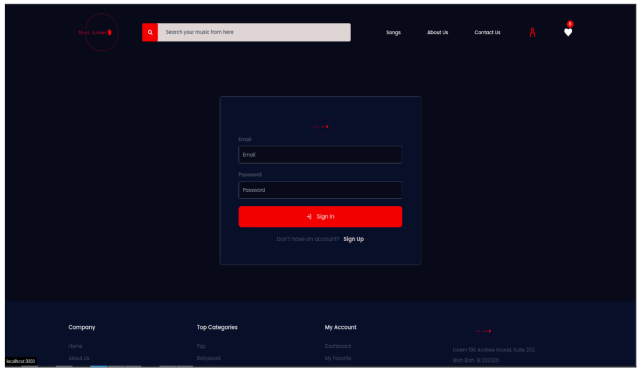
3 Building and Testing

3.1 Core Application: Frontend

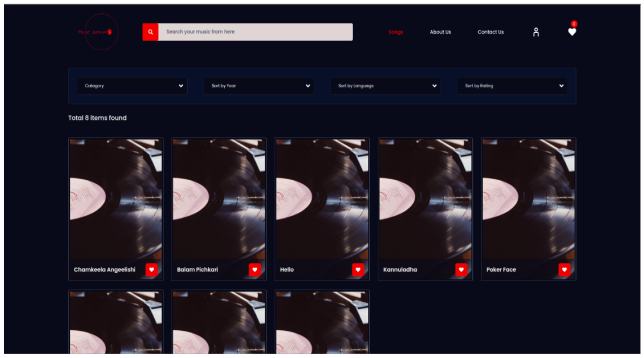
The frontend has been made using React. We had decided to go for the react due to the amount of community support and documentation present. We used Recoil package to store state data of frontend. The code connects to the backend using Axios by making API calls. A few screenshots of the UI are shared below,



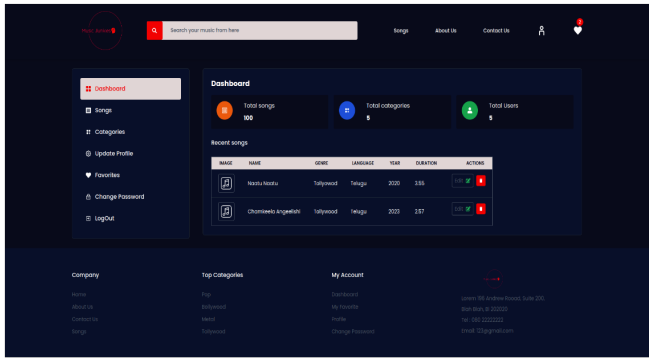
Landing Page



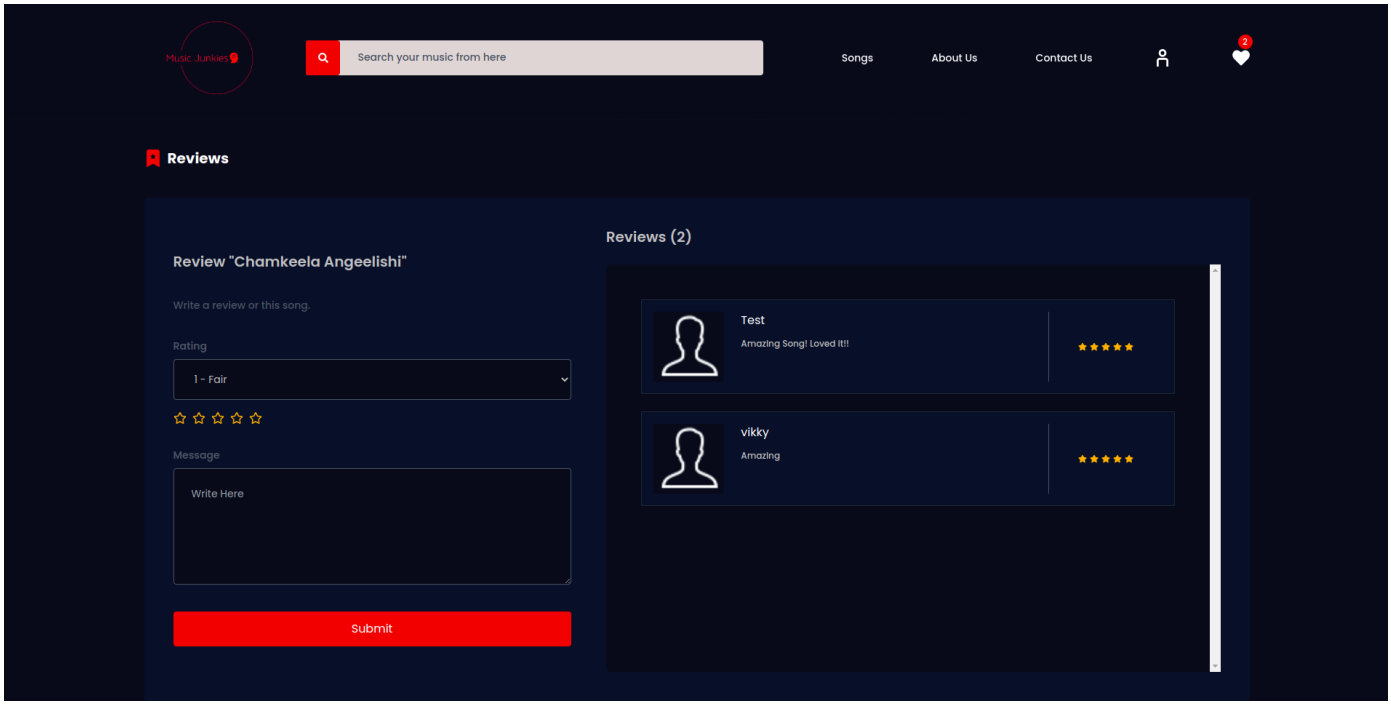
Login Page



Songs Page



User Dashboard Page



The above image is the Review page. We can see the previous reviews and ratings as well.

3.2 Core Application: Backend

In NodeJS, we utilised the power of ExpressJS and mongoose to write the APIs, connect to the database and perform operations. We created two major routes for APIs:

- users
- music

API

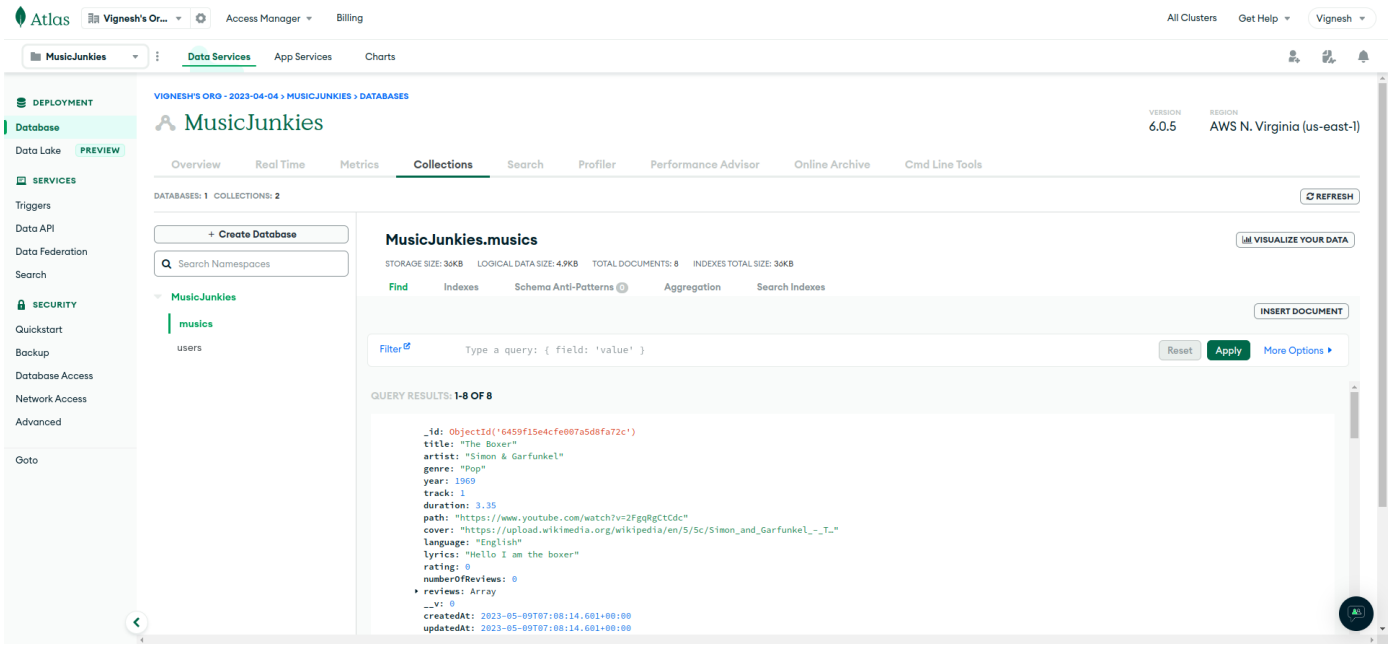
The purpose of the API is to enable connection with the front end and complete the application.

- **Users :** This API has the only purpose of creating users and login. We can get all the users that are present in the database. We are registering the users and logging them in using the APIs under this field. For this, we use JsonWebTokens to create tokens which are given a certain time limit after which they expire. Furthermore, we are hashing the passwords and storing them. We also handle User favourites. There are few APIs restricted for amin user only.
- **Music :** This API has purpose of extracting songs from database. We can also create reviews using these APIs. Admins can add songs to the database.

Link to API Documentation : `https://documenter.getpostman.com/view/16924735/2s93ecxW8X`

3.3 Database

We used MongoDB Atlas for handling our data. It provides an online database to store the data. We created 3 different models:



- User
- Music
- Review

We used mongoose to enable connection with MongoDB with NodeJS.

3.4 Building Environment

We have setup our environment using npm. The repositories have files package.json and package-lock.json which help in replicating the environment

- npm install - install all dependencies mentioned in package.json and package-lock.json
- npm run dev - runs server in developer mode using nodemon
- npm run test - runs the tests
- npm run start - starts the server and runs the application

3.5 Testing the code

This project runs tests on the backend part of the code. The tests have been written using Chai and Mocha. They are present in the /test folder. The test cases have been written keeping the coverage criterion in mind. The output on running the test cases is shown in the image below,

```
vikky@vigneshb~/Documents/8th Sem/SPE/Major Project/server$ npm run test

> server@1.0.0 test
> mocha --exit

Server running on port: 8000


  Admin Login API
    ✔ should return a token when admin logs in (6708ms)

  Admin Get all users API
    ✔ should return all users when requested by admin (522ms)

  users
    /POST /users/login
      ✔ it should NOT LOGIN using the wrong credentials (259ms)
      ✔ it should LOGIN using the correct credentials (338ms)


  4 passing (8s)
```

4 Containerization

4.1 Docker

Docker is a popular platform that enables developers to easily build, package, and deploy applications in containers. Containers are lightweight, standalone, and executable packages that contain all the necessary dependencies, libraries, and configuration files required to run an application. Docker provides a standardized way to create and manage containers, allowing developers to isolate their applications and easily move them between different environments such as development, testing, and production. Docker uses a client-server architecture and is built on top of several technologies, including the Linux kernel's cgroups and namespaces. By using Docker, developers can avoid the "works on my machine" problem, where an application runs fine on a developer's computer but fails in production due to differences in the environment. Docker enables developers to package an application along with its dependencies and configurations, ensuring that it runs consistently across different environments.

Installation:

$ sudo pacman -S docker

$ sudo systemctl start docker

Verify the installation by using the below command,

$ docker –version

Docker version 20.10.23, build 715524332f

4.2 Creating Docker Images

We create the docker image from the DockerFile. So first we need to create dockerfiles which looks like these,

Frontend Dockerfile

The docker file implements the following,

- **FROM:** It imports the base image node:alpine in-order to create a new image. We need node installed to run our application.

```
FROM node:alpine AS build
WORKDIR /app

COPY package*.json /app/
RUN npm install --silent
COPY . .

EXPOSE 8000
CMD [ "npm", "run", "start" ]
```

- **WORKDIR:** It changes the current working directory.
- **COPY:** It can copy a file into the image in working directory. We copy package.json to make installations.
- **RUN:** We run npm install to make necessary installations
- **COPY:** Copy all the files to run
- **EXPOSE:** Expose the port 8000 for the frontend to access backend
- **CMD:** Run command when the image is used

Backend Dockerfile

```
#get the latest alpine image from node registry
FROM node:alpine

#set the working directory
WORKDIR /app

#copy the package and package lock files
#from local to container work directory /app
COPY package*.json /app/

#Run command npm install to install packages
RUN npm install

#copy all the folder contents from local to container
COPY . .

# RUN npm test

EXPOSE 3000
CMD [ "node", "server.js" ]
```

The docker file implements the following,

- **FROM:** It imports the base image node:alpine in-order to create a new image. We need node installed to run our application.
- **WORKDIR:** It changes the current working directory.

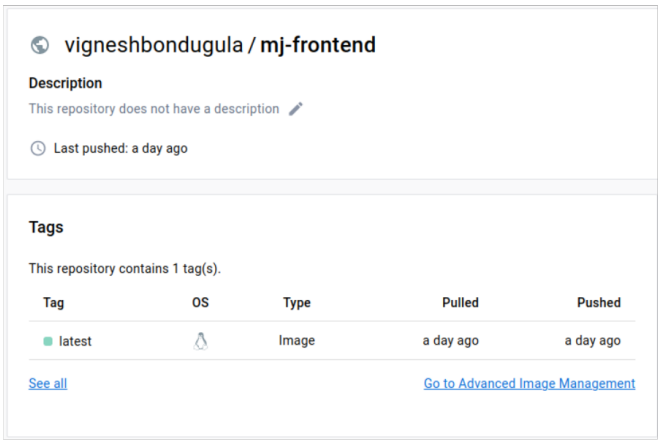
- **COPY:** It can copy a file into the image in working directory. We copy package.json to make installations.
- **RUN:** We run npm install to make necessary installations
- **COPY:** Copy all the files to run
- **EXPOSE:** Expose the port 3000 for the backend to access frontend
- **CMD:** Run command when the image is used

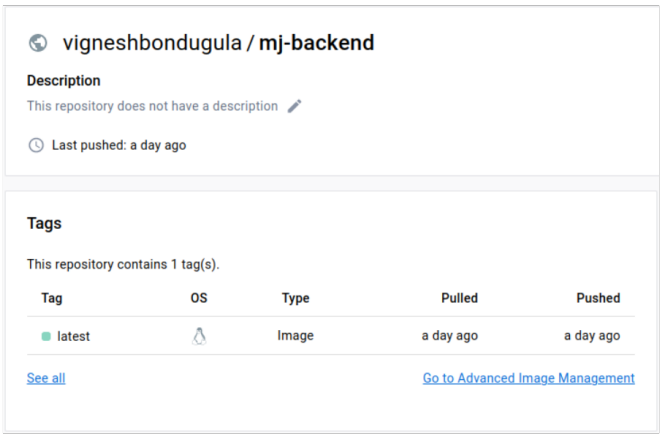
Run Dockerfile to create image:
$ docker build -t name:latest .
Run the above command to generate an image with a given tag using the steps given in Dockerfile. We can check if the image is created by running the following command,
$ docker images

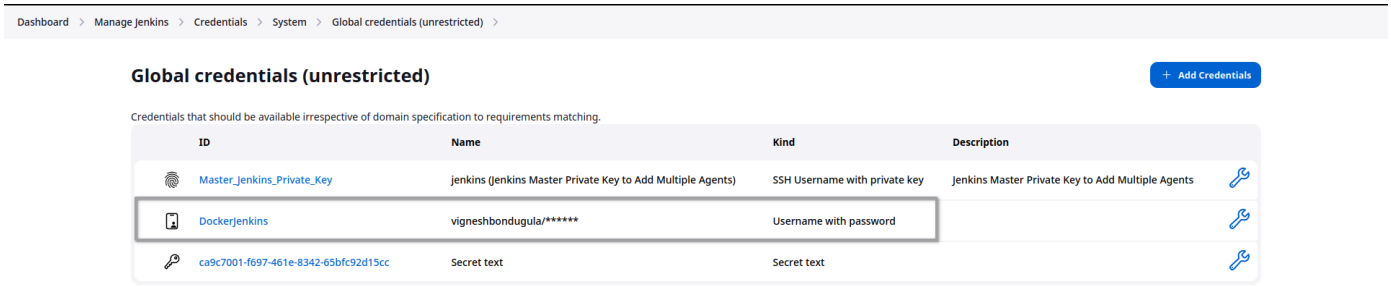
```
vikky@vigneshb~$ docker images
REPOSITORY                                TAG       IMAGE ID       CREATED         SIZE
vigneshbondugula/mj-backend               <none>    7710bdeaf927   2 days ago      204MB
vigneshbondugula/mj-frontend              <none>    fc76ffe01f83   2 days ago      571MB
vigneshbondugula/mj-frontend              <none>    666f29d7b99e   2 days ago      571MB
vigneshbondugula/mj-frontend              <none>    f143dcafc031   2 days ago      571MB
vigneshbondugula/mj-frontend              <none>    e5703bbe74b3   5 days ago      562MB
vigneshbondugula/mj-frontend              <none>    68d37f5d9d19   6 days ago      562MB
vigneshbondugula/mj-backend               <none>    8c40c9561476   6 days ago      204MB
```

4.3 Docker Hub

Docker Hub is a public cloud-based registry service that allows developers to store, manage, and share their Docker container images. It provides a central repository for Docker images that can be accessed by anyone with a Docker account. We now have to push the image we created into the docker hub.
Create a Dockerhub account:
We first create a dockerhub account like github. Then we have to create a repository in dockerhub which we later use to push and pull the images. We login into docker account from terminal by running the following command,
$ docker login
Enter the credentials and now we are ready to push the image we generated.
Push the image to Dockerhub:
Run the following command to push the image into dockerhub,
$ docker push username/projectname:latest
We can check if the image is pushed on to dockerhub by logging into the account on web and checking the repository.
Before adding this step to jenkins pipeline, we need to add the docker credentials to jenkins.
Add dockerhub credentials to jenkins:
Go to Dashboard → Manage Jenkins → Credentials → System → Global Credentials → Add Credentials
Enter the username and password for dockerhub and give a name to credentials.







5 Orchestration

5.1 Kubernetes

Installation:

Install kubectl

Download the latest release with the command:

$ curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

Install kubectl:

$ sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

Test to ensure the version you installed is up-to-date:

$ kubectl version –client

Install minikube

$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

$ sudo install minikube-linux-amd64 /usr/local/bin/minikube

5.2 Deployments and Services

Start the minikube by running the following command,

$ minikube start

Run the following yaml files which created deployements and services:

Frontend : `https://github.com/Abhinav-Kamath/review/blob/main/k8s/frontend-deployement-service.yaml`

Backend : `https://github.com/VigneshBondugula/MusicJunkies-Backend/blob/master/k8s/backend-deployem yaml`

```
vikky@vigneshb~/Documents/8th Sem/SPE/Major Project/server/k8s$ minikube start
😄  minikube v1.29.0 on Arch
✨  Using the docker driver based on existing profile
👍  Starting control plane node minikube in cluster minikube
    Pulling base image ...
    Restarting existing docker container for "minikube" ...

    Docker is nearly out of disk space, which may cause deployments to fail! (89% of capacity). You can pass '--force' to skip this check.
    Suggestion:

    Try one or more of the following to free up space on the device:

    1. Run "docker system prune" to remove unused Docker data (optionally with "-a")
    2. Increase the storage allocated to Docker for Desktop by clicking on:
    Docker icon > Preferences > Resources > Disk Image Size
    3. Run "minikube ssh -- docker system prune" if using the Docker container runtime
    Related issue: https://github.com/kubernetes/minikube/issues/9024

    minikube 1.30.1 is available! Download it: https://github.com/kubernetes/minikube/releases/tag/v1.30.1
    To disable this notice, run: 'minikube config set WantUpdateNotification false'

!   This container is having trouble accessing https://registry.k8s.io
    To pull new external images, you may need to configure a proxy: https://minikube.sigs.k8s.io/docs/reference/networking/proxy/
    Preparing Kubernetes v1.26.1 on Docker 20.10.23 ...
    Configuring bridge CNI (Container Networking Interface) ...
    Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
    Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

The entire application has been orchestrated using Kubernetes. A simple deployment is shown in the image below. The image shows details regarding the number of deployments and the service being used by Kubernetes Node to serve the application,

```
vikky@vigneshb~$ kubectl get all
NAME                                            READY   STATUS    RESTARTS   AGE
pod/musicjunkies-backend-depl-985dddddd-9rm5q   1/1     Running   0          84s
pod/musicjunkies-backend-depl-985dddddd-fmng9   1/1     Running   0          84s
pod/musicjunkies-frontend-depl-6696c66fd4-fphbq   1/1     Running   0          39s
pod/musicjunkies-frontend-depl-6696c66fd4-s4q5q   1/1     Running   0          39s

NAME                                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
service/kubernetes                  ClusterIP   10.96.0.1       <none>        443/TCP          51d
service/musicjunkies-backend-svc    ClusterIP   10.104.98.177   <none>        8000/TCP         84s
service/musicjunkies-frontend-svc   NodePort    10.109.69.236   <none>        3000:30000/TCP   39s

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/musicjunkies-backend-depl    2/2     2            2           84s
deployment.apps/musicjunkies-frontend-depl   2/2     2            2           39s

NAME                                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/musicjunkies-backend-depl-985dddddd     2         2         2       84s
replicaset.apps/musicjunkies-frontend-depl-6696c66fd4   2         2         2       39s
```

In the image, we can see that there are 2 pods running replicas of the frontend and 2 for backend. Frontend pods communicate using the NodePort service. The proxy mentioned in frontend sends requests to the backend ClusterIP service which is an internal service

6 Deployement

6.1 Ansible

Ansible is an open-source configuration management and automation tool that simplifies the deployment, configuration, and management of IT infrastructure. It allows users to automate repetitive tasks, manage multiple servers, and deploy applications in a fast and efficient manner. Ansible uses a simple, declarative language called YAML to define playbooks, which are sets of instructions for configuring and deploying systems. Playbooks can be written to perform a wide range of tasks, such as installing software, configuring network devices, managing users and permissions, and more.
In this project we use ansible to pull the image from the dockerhub and run it on a container in multiple servers using playbook.
Installation:
$ sudo pacman -S ansible
Make sure python is installed first. Verify the installation by using the below command,
$ ansible –version

```
vikky@vigneshb~/Documents/8th Sem/SPE/MiniProj/SPE_MiniProject$ ansible --version
ansible [core 2.14.2]
  config file = /etc/ansible/ansible.cfg
  configured module search path = ['/home/vikky/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3.10/site-packages/ansible
  ansible collection location = /home/vikky/.ansible/collections:/usr/share/ansible/collections
  executable location = /usr/bin/ansible
  python version = 3.10.9 (main, Dec 19 2022, 17:35:49) [GCC 12.2.0] (/usr/bin/python)
  jinja version = 3.1.2
  libyaml = True
```

6.2 Inventory

The inventory file in Ansible is used to specify the hosts and host groups that are targeted by tasks, modules, and commands in a playbook. Ansible can simultaneously manage multiple hosts within an infrastructure using an inventory list or a group of lists. After the inventory is defined, you can select specific hosts or groups by using patterns to run commands and tasks against them. For this project, we are try to deploy only on localhost. The following is the inventory file, To check if the hosts mentioned in the inventory file are active

```
[local]
localhost ansible_connection=local
```

we can run the following command,
$ ansible -i inventory -m ping all

```
vikky@vigneshb~/Documents/8th Sem/SPE/MiniProj/SPE_MiniProject$ ansible -i inventory -m ping all
[WARNING]: Platform linux on host localhost is using the discovered Python interpreter at /usr/bin/python3.10, but future installation of another Python
interpreter could change the meaning of that path. See https://docs.ansible.com/ansible-core/2.14/reference_appendices/interpreter_discovery.html for more
information.
localhost | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.10"
    },
    "changed": false,
    "ping": "pong"
}
```

6.3 Implementing and Running Playbooks

We create the following playbooks, which are a yaml files and mention all the steps that are needed for deployment,

```yaml
---
- name: "Deploying backend service"
  hosts: all
  connection: local
  tasks:
    - name: Start docker service
      service:
        name: docker
        state: started

    - name: "Delete deployment if already exists"
      kubernetes.core.k8s:
        state: absent
        definition: "{{ lookup('file', '../k8s/backend-deployement-service.yaml')}}"

    - name: "Create a backend deployement"
      kubernetes.core.k8s:
        state: present
        definition: "{{ lookup('file', '../k8s/backend-deployement-service.yaml')}}"
```

```yaml
---
- name: "Deploying frontend service"
  hosts: all
  connection: local
  tasks:
    - name: Start docker service
      service:
        name: docker
        state: started

    - name: "Delete deployment if already exists"
      kubernetes.core.k8s:
        state: absent
        definition: "{{ lookup('file', '../k8s/frontend-deployement-service.yaml')}}"

    - name: "Create a frontend deployement"
      kubernetes.core.k8s:
        state: present
        definition: "{{ lookup('file', '../k8s/frontend-deployement-service.yaml')}}"
```

Start docker service checks if the docker service is running in the host.
Delete deployements if exists checks if the kubernetes configuration files were already applied before and if so they delete them.
Create a deployement runs applies the kubernetes files and creates deployements and services .
To run the playbook we run the following command,
$ ansible -i inventory playbook.yml To check if the playbook we ran created pods run the following command,
$ kubectl get all

7 Continuous Integration

Continuous integration (CI) involves automating the process of integrating code changes from various contributors into a single software project. This is a fundamental best practice in DevOps, which enables developers to merge code changes into a central repository on a frequent basis, followed by the execution of builds and tests using automated tools to ensure the correctness of the new code before integration. The key component of the CI process is a source code version control system, which is augmented by other checks such as automated code quality tests and syntax style review tools. Regular integration through CI helps to rapidly detect and locate errors.

7.1 Jenkins

Jenkins is an automation server that is open-source and used to automate different stages of software development processes such as building, testing, and deploying software applications. It is customizable and can be configured to meet the requirements of various software development teams. Through a web-based interface, Jenkins allows users to create and set up jobs that automate different tasks in software development. These jobs can be triggered either manually or automatically based on a schedule or events such as changes in the code repository. Jenkins also supports third-party plugins that can be used to extend its functionality and integrate it with other software development tools. It is widely utilized in DevOps environments to simplify software development and deployment processes, enhance communication and collaboration between development and operations teams. The Jenkins pipeline was utilised in this project to handle until delivery, i.e. continuous delivery.

Installation:

$ sudo pacman -S jenkins

$ sudo systemctl start jenkins

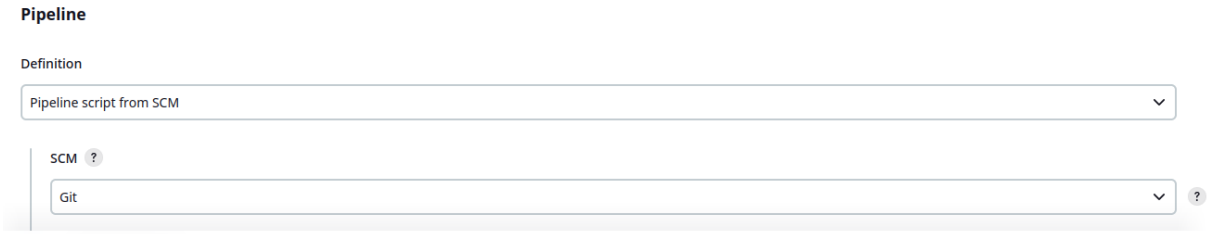
In Arch Linux jenkins is served on localhost:8090.

Prerequisites:

- Create a user in jenkins.
- Install all the suggested plugins.
- Additionally install jenkins-pipeline, git and ansible plugins.

7.2 Creating Jenkins Pipelines

Create two pipeline for frontend and backend by clicking create pipeline project in jenkins dashboard. This configures the github repository with the jenkins pipeline.



This configures to use the pipeline script from the source code. Hence, we create a Jenkinsfile in the repository

which contains the details of the pipeline. The pipeline script has various stages. The pipeline script for backend looks like below,

```
pipeline {
    // The "agent" section configures on which nodes the pipeline can be run.
    // Specifying "agent any" means that Jenkins will run the job on any of the
    // available nodes.
    agent any

    stages {
        stage('Git Pull') {
            steps {
                // Get code from a GitHub repository
                // Make sure to add your own git url and credentialsId
                git url: 'https://github.com/VigneshBondugula/MusicJunkies-Backend.git'
            }
        }
        stage('Build and Test') {
            steps {
                // Maven build, 'sh' specifies it is a shell command
                sh 'npm i && npm test'
            }
        }
        stage('Build Docker Images') {
             steps {
                 sh 'docker build -t vigneshbondugula/mj-backend:latest .'
             }
        }
        stage('Publish Docker Images') {
            steps {
                withDockerRegistry([ credentialsId: "DockerJenkins", url: "" ]) {
                    sh 'docker push vigneshbondugula/mj-backend:latest'
                }
            }
        }
        stage('Clean Docker Images') {
            steps {
               sh 'docker rmi -f vigneshbondugula/mj-backend:latest'
            }
        }
        stage('Deploy and Run Image') {
            steps {
                ansiblePlaybook becomeUser: null, colorized: true, disableHostKeyChecking: true, installation: 'Ansible', inventory: './ansible/inventory', playbook: './ansible/playbook.yml',
            }
        }
    }
      post {
            always {
                sh 'docker logout'
            }
        }
}
```

Stage View

	Declarative: Checkout SCM	Git Pull	Build and Test	Build Docker Images	Publish Docker Images	Clean Docker Images	Deploy and Run Image	Declarative: Post Actions
Average stage times: (Average full run time: ~3min 13s)	6s	6s	11s	30s	1min 51s	1s	16s	742ms
#30 May 12 22:30 2 commits	6s	6s	8s	28s	1min 49s	678ms	13s	1s
#29 May 12 16:17 2 commits	6s	5s	10s	25s	1min 56s	374ms	11s	492ms

There are multiple stages in the jenkins pipeline.

- We start with the git pull of the updated repository.
- Then we build and test using the npm commands.
- Next, we build docker images and publish them to docker hub.
- We clear the local docker images.
- Finally we deploy by running ansible playbook.

The stage view is shown above. The frontend pipeline also looks exactly similar with changes in the image names.

8 Monitoring Using ELK

8.1 Elastic Search and Kibana Visualization

To visualize our logs, we will utilize the online version of Kibana. Kibana is an exploration, analysis, and visualization platform that operates in a browser. It is a vital component of the Elastic stack, along with Elasticsearch and Logstash. Using Kibana's intuitive user interface, we can transform indexed Elasticsearch data into visual representations such as diagrams, graphs, charts, and maps. After we run the application in the kubernetes we can find a log file generated. We copy this log file from kubernetes container to minikube container by adding volumeMounts in kubernetes configuration file,

```
        volumeMounts:
          - name: data
            mountPath: /app/logs/
      volumes:
      - name: data
        hostPath:
          path: /home/docker/logs
```

Then run the following command to mount the minikube directory to local,
$ minikube mount /local/logs:/home/docker/logs
We now have updating log file locally available, hence can run ELK on that file.

8.2 Orchestration of ELK using docker-compose

I have used docker containers to run ELK. I made 3 containers and made orchestration using the below docker-compose file,

```
version: '1'
services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.13.1
    container_name: elasticsearch
    environment:
      - discovery.type=single-node
    ports:
      - "9200:9200"
      - "9300:9300"

  logstash:
    image: docker.elastic.co/logstash/logstash:7.13.1
    container_name: logstash
    command: logstash -f /config/logstash.conf
    links:
      - elasticsearch
    ports:
      - "5044:5044"
    volumes:
      - /home/vikky/logs/config:/config

  kibana:
    image: docker.elastic.co/kibana/kibana:7.13.1
    container_name: kibana
    ports:
      - "5601:5601"
```

We mention to create three containers with images of elasticsearch, logstash and kibana respectively. We mention the port number where they are running. We also mount volume to logstash container for it to access the config file.

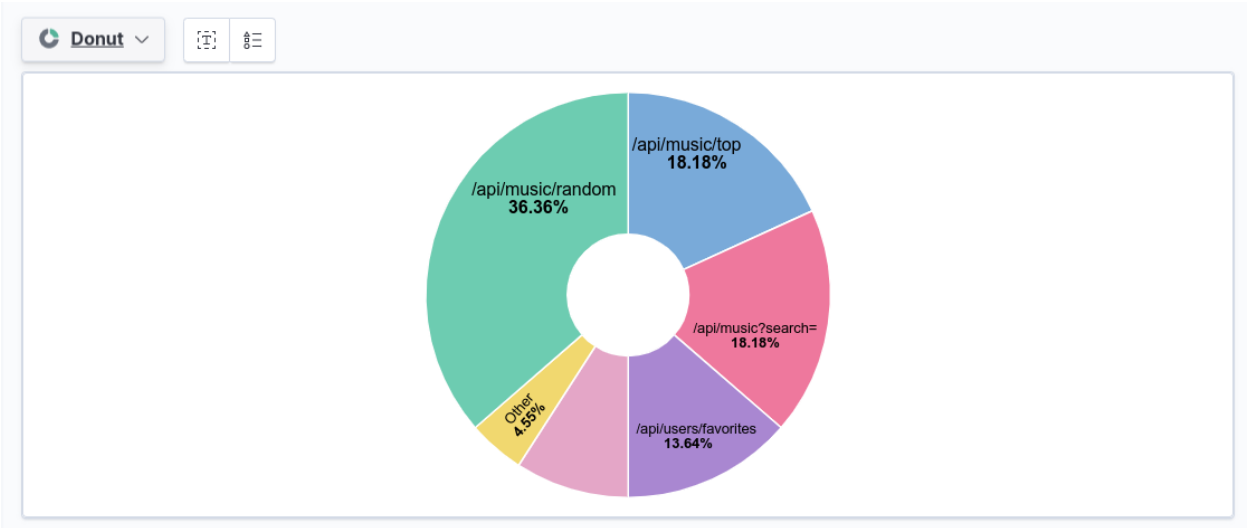
8.3 Logstash Script

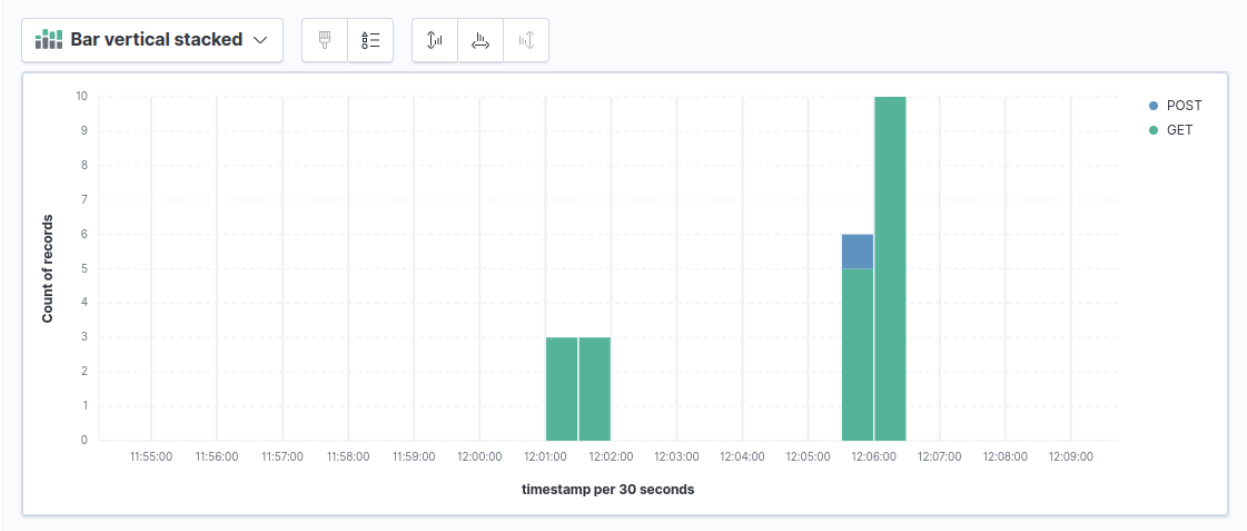
```
input {
  file {
    path => "/config/backend.log"
    start_position => "beginning"
  }
}

## Add your filters / logstash plugins configuration here

filter {
  grok {
    match => {
      "message"=>"%{TIMESTAMP_ISO8601:timestamp} %{WORD:method} %{URIPATHPARAM:request} %{DATA:userid} %{NUMBER:status}"
    }
  }

  date {
    match => [ "timestamp", ISO8601 ]
    target => "@timestamp"
  }
}

output {
  elasticsearch {
    action => "create"
    index => "musicjunkies_logs"
    hosts => "elasticsearch:9200"
  }
}
```

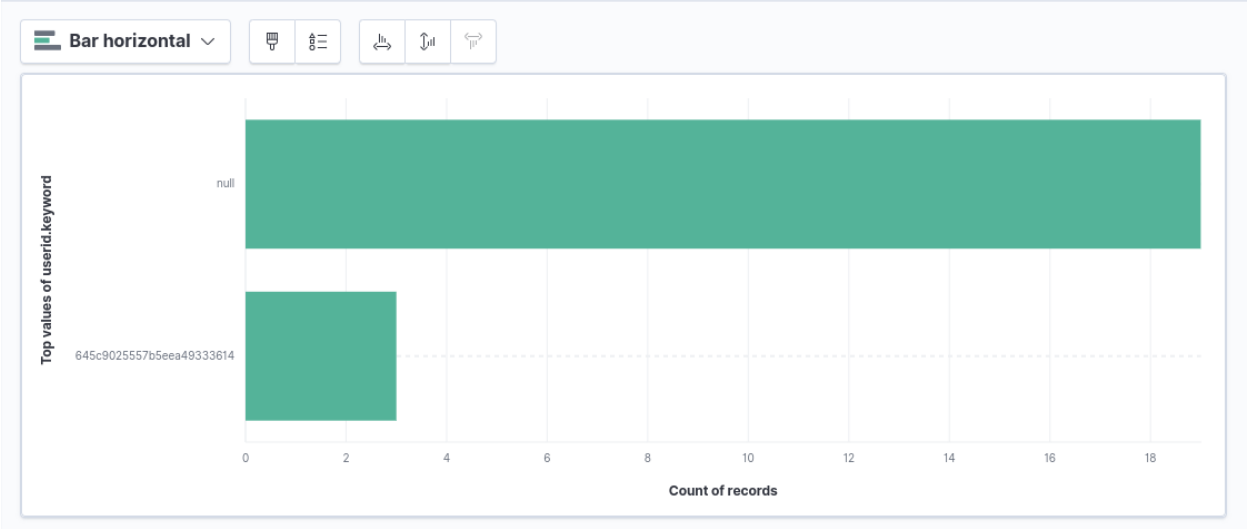
The above is the config file set for logstash. We set the input file path of the log file. Then we give the grok pattern to match with the logs. we have mentioned few fields like userid, status, request and method in the log so that we can make various visualizations using these. Then finally we set the index name and location of elasticsearch in hosts to forward the index.

8.4 Dashboards on Kibana

We created the following visualizations using the log files and created a dashboard, These visualizations tell







us which apis are ran more frequently and by which user. We can also see how many requests the server gets over timesteps.

9 Webhooks

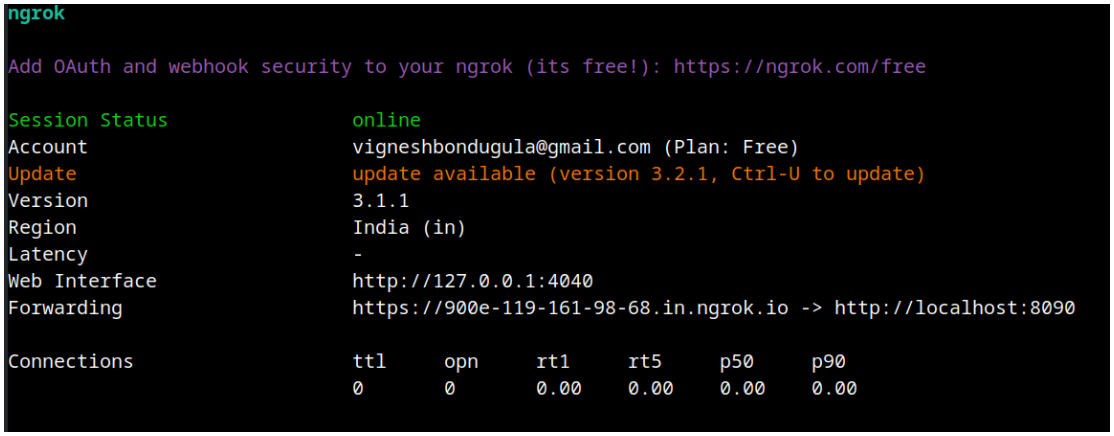
9.1 ngrok

Ngrok is a tool that allows you to create a secure tunnel to a local server or web application, enabling you to expose it to the internet for testing and development purposes. It provides a public URL that can be accessed from anywhere, even if your local server is behind a firewall or NAT. This is useful for developers who want to test their applications on multiple devices, share their work with others, or demo their application to clients or stakeholders. Ngrok supports HTTP, HTTPS, and TCP protocols, and can be used with any programming language or framework that can run on a local server. It is available as a command-line tool or a web interface, and offers both free and paid plans with varying levels of features and functionality. Since, we are running the jenkins server on the localhost we need to use ngrok in order to implement webhooks.

Installation:

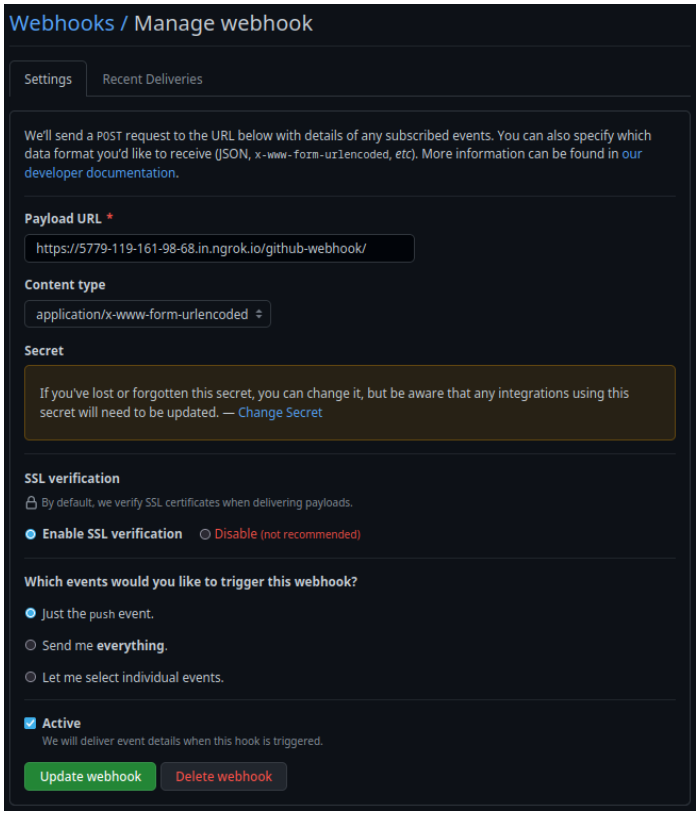
$ sudo pacman -S ngrok

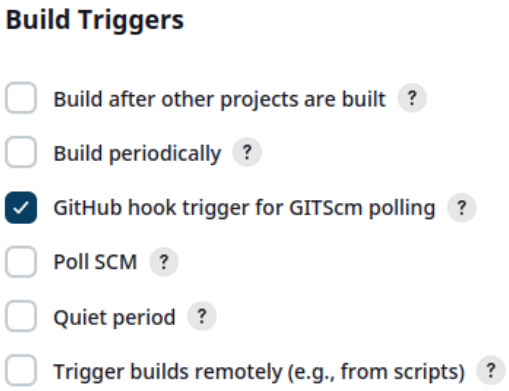
We need to signup in ngrok and give authorization through terminal using the command given in ngrok website. Now we run the following command to start ngrok forwarding,
$ ngrok http 8090



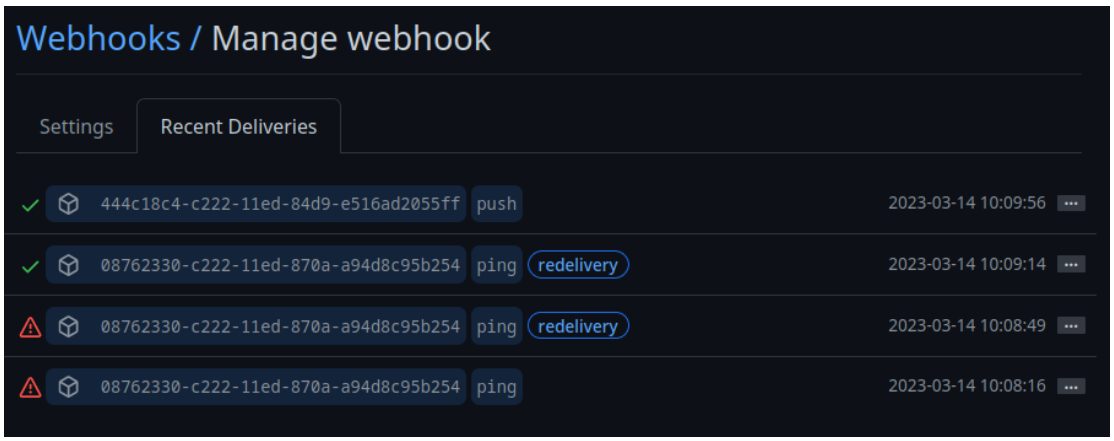
9.2 Github Webhook

We use the github webhook functionality to inform jenkins server that there is a change in the repository and trigger the pipeline build. We now need to copy the ngrok url into the payload url of github webhook settings as shown below,

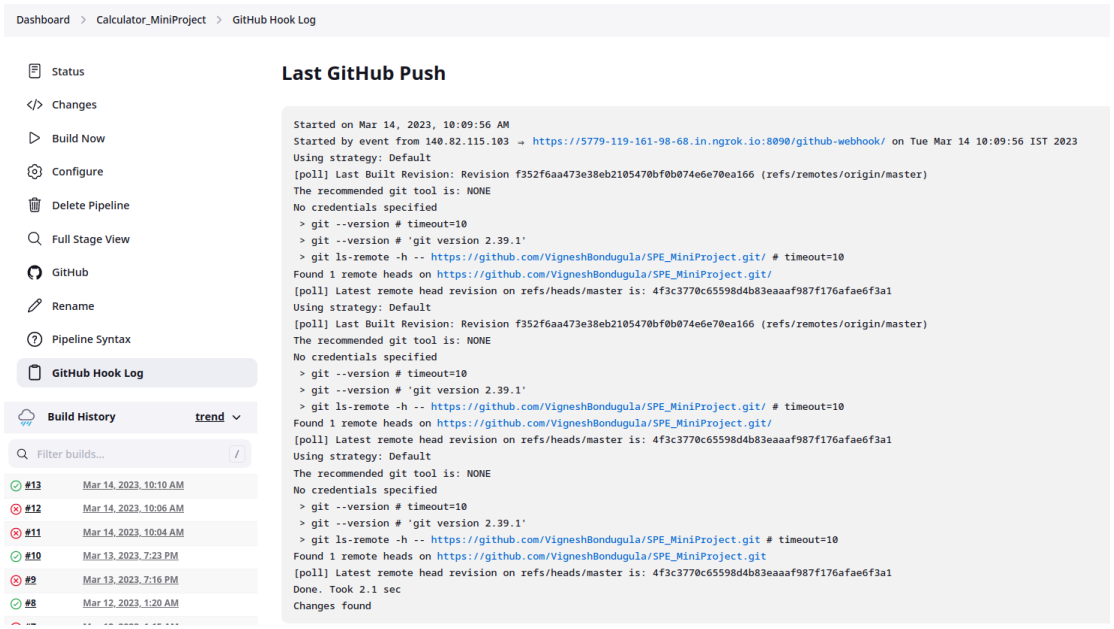




We also need to modify the configuration of build triggers in the jenkins pipeline as shown above. Now we can verify if the deliveries if we get a green mark in the deliveries as shown below,



Also the jenkins build should be triggered by itself. We can also see the github hook log in jenkins as shown below,



10 References

Docker

https://docs.docker.com/

https://www.google.com/url?sa=t&source=web&rct=j&url=https://docs.docker.com/engine/install/linux-postinstall/&ved=2ahUKEwjYxLPU0oL9AhWH-TgGHYaoDgwQFnoECAwQAQ&usg=AOvVaw3f3VUxOnrnwRaYG6V02pR0

https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/jenkins-docker-pipel

ELK Stack

https://logz.io/learn/complete-guide-elk-stack/

Jenkins Plugins

https://www.jenkins.io/doc/book/managing/plugins/