LinkUp

CS 816 - Software Production Engineering

Major Project - LinkUp - Social Media application

Divyanshi Rajput - IMT2019029

Nikhil Agarwal - IMT2019060

Project Description

LinkUp is a social media application that provides a platform for users to create a profile, make posts, share images, like and comment on posts, make friends, and engage with their community. The platform is designed to be user-friendly, with a simple and intuitive interface that allows users to easily navigate and interact with the platform.

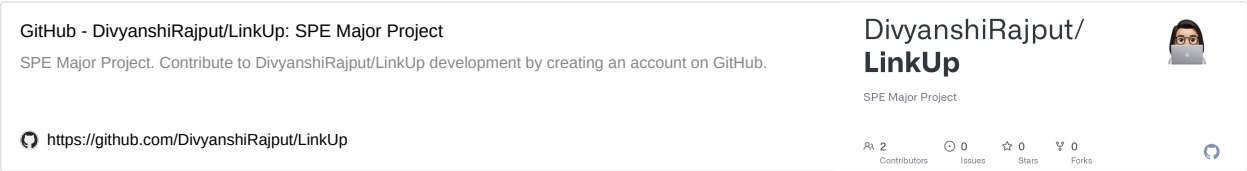
With LinkUp, users can create a personalized profile that showcases their interests, hobbies, and personality. They can then make posts and share images on their profile, which can be seen by their friends and followers. Users can also like and comment on posts made by others, creating a sense of community and engagement.

The architecture of our projects requires three layers:

- Frontend
- Backend
- Database

We have described the technology stack and functionalities of our application in more detail later in this report.

GitHub Repository Link



What is DevOps?

DevOps is a set of practices and principles that aim to improve communication and collaboration between development and operations teams, in order to improve the speed and quality of software delivery. DevOps involves automating processes, using agile methodologies, and leveraging tools to support continuous integration, delivery, and deployment. By breaking down silos and promoting collaboration, DevOps helps organizations to deliver software more quickly, with fewer errors, and with greater efficiency.

Few of the main practices and principles of DevOps are:

1. Continuous Integration (CI): developers integrate their code into a shared repository frequently, which triggers an automated build and test process.
2. Continuous Delivery (CD): code changes are automatically built, tested, and deployed to production.
3. Infrastructure as Code (IaC): infrastructure is defined in code and managed through version control, allowing for versioning, testing, and automated deployment.

4. Monitoring and Logging: real-time monitoring and logging enable teams to quickly identify and address issues.
5. Collaboration and Communication: DevOps emphasizes collaboration and communication between developers, operations, and other stakeholders.

Why DevOps?

DevOps is important because it helps organizations to deliver software quickly, with fewer errors, and with greater efficiency. Here are a few points of importance of DevOps:

- **Faster time-to-market**: DevOps enables organizations to release software more quickly, allowing them to respond to changing market conditions and customer needs more rapidly.
- **Improved quality**: By automating processes and using agile methodologies, DevOps helps to reduce the number of errors in software releases, resulting in higher quality software.
- **Increased collaboration**: DevOps emphasizes collaboration and communication between development and operations teams, which helps to break down silos and improve the overall efficiency of the organization.
- **Scalability**: DevOps practices enable organizations to scale their software delivery processes to meet the needs of their users, without sacrificing quality or speed.
- **Enhanced security**: By incorporating security into the development process, DevOps helps to reduce the risk of security vulnerabilities and data breaches.

Technology Stack used for Project Pipeline

Purpose	Tools	Description
Source Control Management	Git and GitHub	Version control system for software development and collaboration
Development	MERN Stack (MongoDB, Express, ReactJS and NodeJS)	MERN Stack is a full-stack web development framework that uses MongoDB as the database, Express as the web framework, ReactJS as the frontend library, and NodeJS as the backend runtime environment.
Build Trigger	Github webhook and ngrok	The Build Trigger uses a Github webhook to trigger a build and test process for the code changes. Ngrok is used to create a public URL to expose the local server to the internet.
Testing	Mocha	Testing framework for javascript
Continuous Integration	Jenkins	Automation server for continuous integration and continuous delivery
Containerize	Docker, Docker Desktop, and Docker Hub	Platform for developing, shipping, and running applications using containers
Deployment (on localhost)	Kubernetes	Open-source software for automating software provisioning, configuration management, and application deployment
Monitoring	ELK stack	Monitoring and logging platform for analyzing and visualizing data from multiple sources

API documentation

API route	API Endpoint	Method	Description
`/auth`	`/auth/register`	POST	`register`: This function is used to register a new user. It takes in the user information from the request body - including the first name, last name, email, password, picture path, friends, location, and occupation - and uses the `bcrypt` library to generate a salt and hash the user's password. It then creates a new `User` object using the `User` model and saves it to the database. The function also logs the registration event using a logger and returns the saved user object as a response.

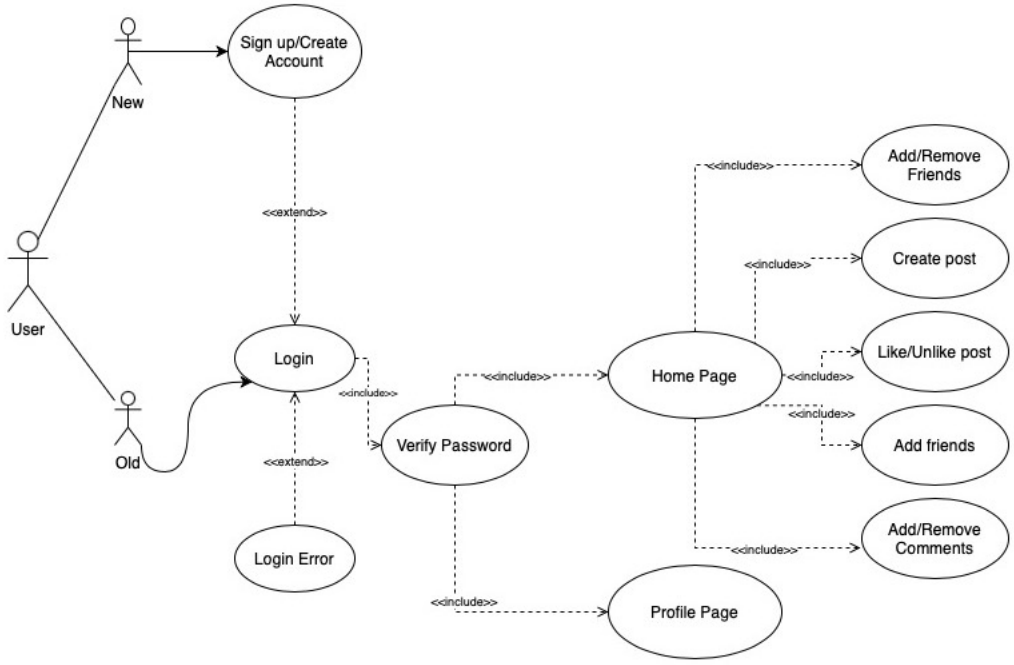
`/auth`	`/auth/login`	POST	`login`: This function is used to authenticate a user and generate a JWT token. It takes in the user's email and password from the request body and uses the `User` model to find the user with the given email. If the user is found, it uses the `bcrypt` library to check if the password matches the hashed password stored in the database. If the password is valid, the function generates a JWT token using the `jsonwebtoken` library and logs the login event using a logger. The function then returns the token and the user object (with the password removed) as a response.
`/users`	`/users/:id`	GET	`getUser`: This function is used to get a user's information. It takes in the user ID from the request parameters and uses the `User` model to find the user with the given ID. It then returns the user object as a response.
`/users`	`/users/:id/friends`	GET	`getUserFriends`: This function is used to get a user's friends. It takes in the user ID from the request parameters and uses the `User` model to find the user with the given ID. It then uses the `Promise.all` method to find all the friends of the user using the `User` model and the `id` property of each friend in the user's friends array. It formats the friends' data and returns it as a response.
`/users`	`/users/:id/:friendId`	PATCH	`addRemoveFriend`: This function is used to add or remove a friend for a user. It takes in the user ID and friend ID from the request parameters and uses the `User` model to find the user and friend with the given IDs. If the user already has the friend as a friend, the function removes the friend and vice versa. It then saves the changes to the database using the `save` method. The function also logs the event using a logger and returns the user's updated friend list as a response.
`/posts`	`/posts`	POST	`createPost`: This function is used to create a new post. It takes in the user ID, description, and picture path from the request body and uses the `User` model to find the user with the given ID. It then creates a new `Post` object using the `Post` model and saves it to the database. The function also logs the event using a logger and returns all posts as a response.
`/posts`	`/posts/`	GET	`getFeedPosts`: This function is used to get all posts in the database. It uses the `Post` model to find all posts and returns them as a response.
`/posts`	`/posts/:userId/posts`	GET	`getUserPosts`: This function is used to get all posts for a particular user. It takes in the user ID from the request parameters and uses the `Post` model to find all posts with the given user ID. It then returns the posts as a response.
`/posts`	`/posts/:id/like`	PATCH	`likePost`: This function is used to like or unlike a post. It takes in the post ID from the request parameters and the user ID from the request body. It uses the `Post` and `User` models to find the post and user with the given IDs, and checks if the user has already liked the post. If the user has liked the post, the function removes the like; otherwise, it adds a like. The function then updates the post in the database and returns the updated post as a response.

Features and Functionalities

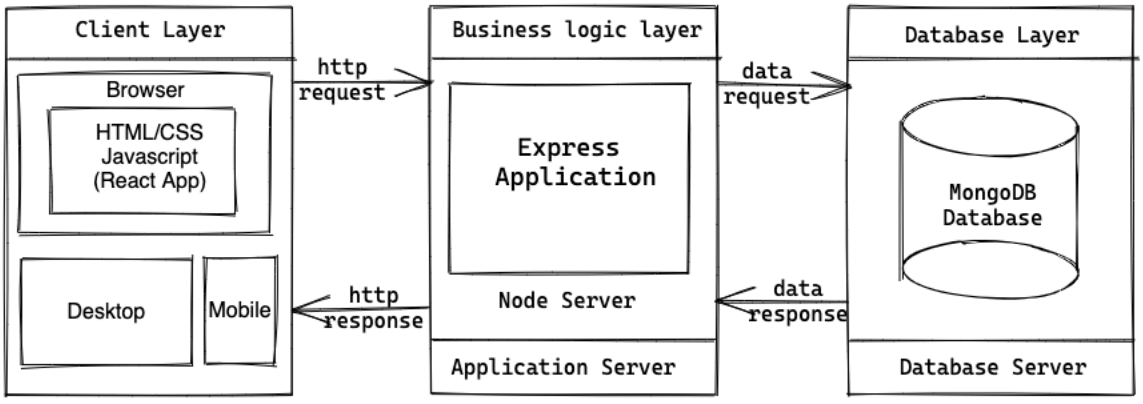
Feature	Description
User profile	Users can create a personalized profile that showcases their interests, hobbies, and personality.
Posts and sharing	Users can make posts and share images on their profile, which can be seen by their friends and followers.
Likes and comments	Users can like and comment on posts made by others, creating a sense of community and engagement.

Feature	Description
Friending	Users can make friends and engage with their community.
Dark Mode/ Light Mode	Change color theme according to user preference
User Authentication	User can login and logout of the web application
Add social handles of other social media	You can add social media handles of twitter and linkedin, which can enable others to connect you on other platforms
Post with images or text	You can make post with images, which will be stored securely in our database.

Use Case Diagram



High Level Architecture



Jenkins Pipeline

Building the CI/CD pipeline using Jenkins

Setting up Jenkins

- Install Java (this is required for Jenkins to run)

```
brew install java
java -version
```

You will get the following output if your java is installed properly

```
openjdk version "19.0.2" 2023-01-17
OpenJDK Runtime Environment Homebrew (build 19.0.2)
OpenJDK 64-Bit Server VM Homebrew (build 19.0.2, mixed mode, sharing)
```

- In mac, we can use homebrew to install Jenkins (homebrew is equivalent to apt in linux for macOS)

```
brew install jenkins-lts
```

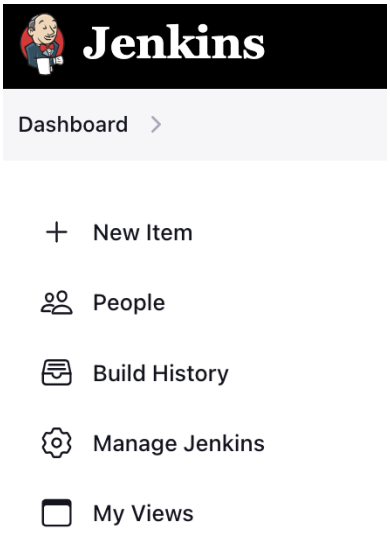
- Now to start jenkins, use the following command. This will start Jenkins service on localhost:8080

```
brew services start jenkins-lts
```

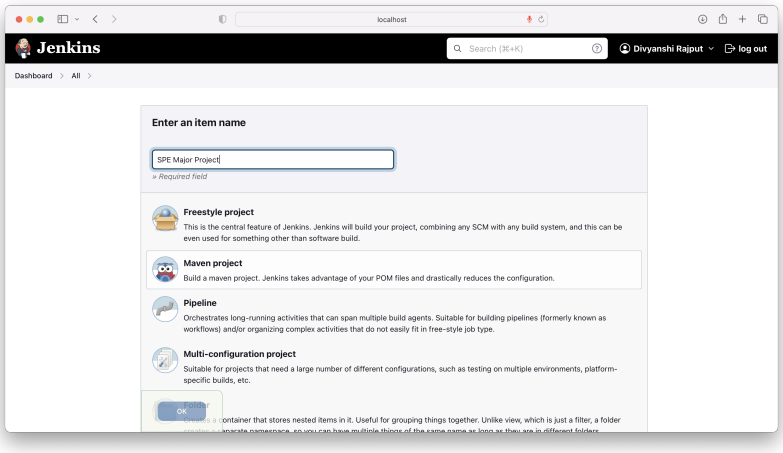
- Follow the setup wizard to create a adminstartion account using a username and password
- Once the setup is complete, login using your account details and now you can start creating a pipeline in Jenkins

Creating a pipeline in Jenkins

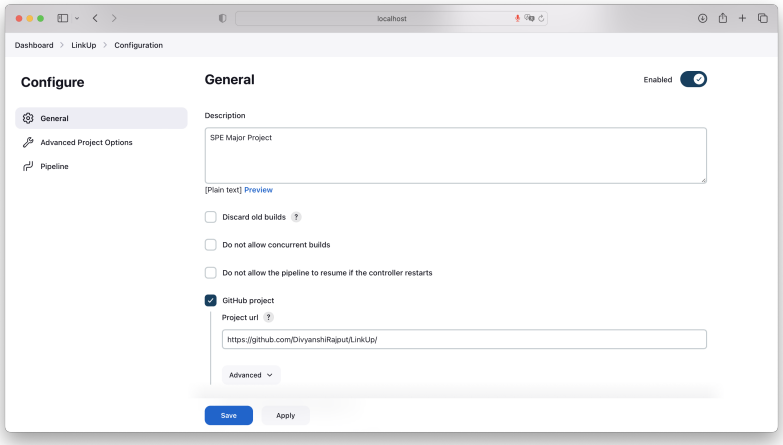
- create a new Item using the **+** option in Jenkins dashboard



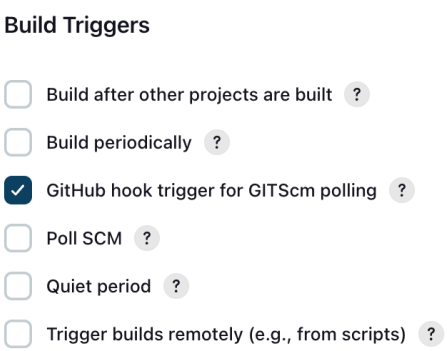
- Name your project and select the Pipeline option



- Add a description, and mention that its a github project by clicking the github project checkbox



- Add Github hook for build triggers, this will ensure that every new push to our github repository is build by Jenkins. (This is done using Github Webhook and ngrok which is describe in the following sections)

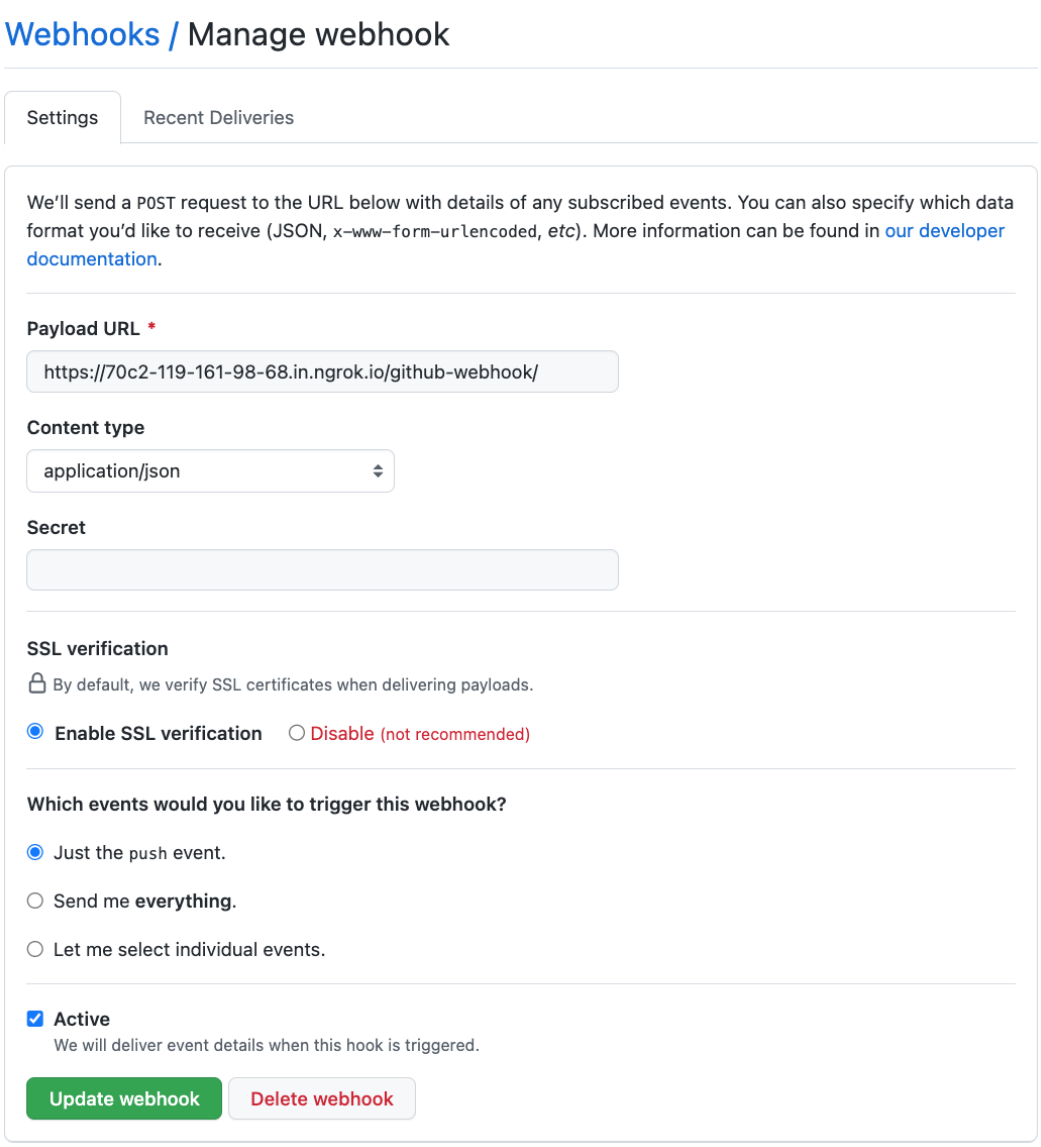


- Add the github repository URL which will be used to clone the repository by Jenkins on build trigger, also mention the branches which you want to build. Finally add the path to the script for the pipeline in github repository. In our case it is **Jenkinsfile** which

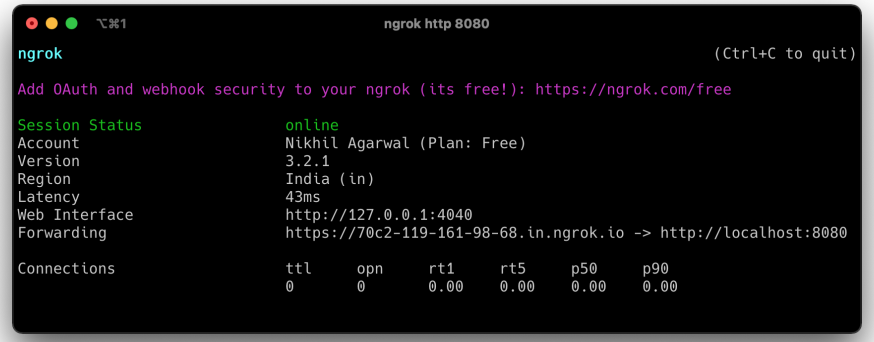
is at the root of the project repository. After adding this information click **Save**

Setting up build trigger - Github Webhook and Ngrok

- **Github Webhook**: Github provides a webhook feature which can be used to trigger a build on Jenkins on every code push to the repository. This webhook sends a POST request to the Jenkins server, which triggers the build. In order to set up a webhook, you need to do the following:
  1. Go to your repository on Github, and click on the **Settings** tab.
  2. Click on **Webhooks**, and then click on **Add webhook**.
  3. Enter the **Payload URL** for your Jenkins server. This will be the URL of your Jenkins server, followed by the path to your webhook URL. For example: `http://your-jenkins-server/github-webhook/`.
  4. Select the events that you want to trigger the webhook. In this case, we want to trigger the webhook on every code push, so we select **Just the push event**.
  5. Select the **Active** checkbox, and then click on **Add webhook**.



- **Ngrok**: Ngrok is a tool that allows you to create a secure tunnel from a public endpoint to a locally running web service. We will use Ngrok to create a public endpoint for our Jenkins server, so that Github can send webhook requests to it. To set up Ngrok, you need to do the following:
    1. Download and install Ngrok from the official website.
    2. Start Ngrok by running the following command in your terminal: `ngrok http 8080`. This will create a secure tunnel from a public endpoint to your locally running Jenkins server on port 8080.
    3. Copy the **Forwarding** URL from the Ngrok console. This is the public endpoint that you will use as your Payload URL in the Github webhook configuration.



After setting up Github webhook and Ngrok, you can test the webhook by pushing some code changes to your repository. If everything is set up correctly, you should see a new build trigger in your Jenkins dashboard.

Creating Jenkinsfile

To create a pipeline script for Jenkins, you need to create a file called `Jenkinsfile` in the root of your project repository. The `Jenkinsfile` should contain the steps that Jenkins will follow to build, test, and deploy your project.

The `Jenkinsfile` is written in a domain-specific language (DSL) called Groovy. Groovy is a Java-based scripting language that is designed for use with the Java Virtual Machine (JVM). You can use Groovy to define the steps that Jenkins will follow in your pipeline.

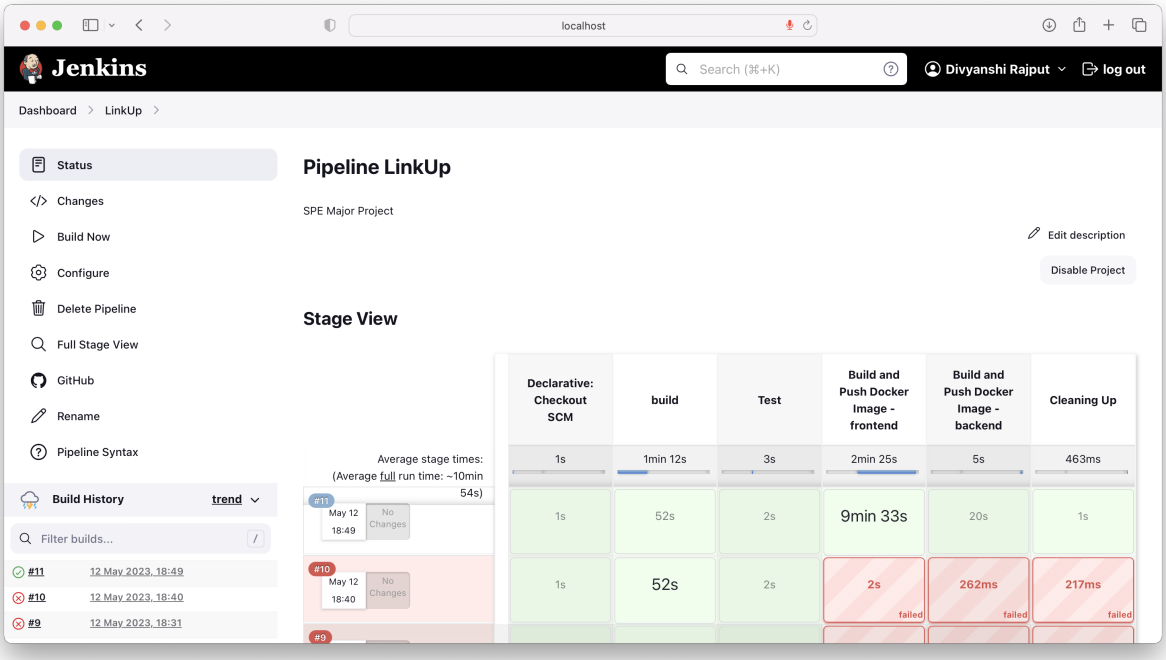
Here is an example of a simple `Jenkinsfile` that builds a Java project:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building...'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying...'
            }
        }
    }
}
```

This `Jenkinsfile` defines a pipeline with three stages: build, test, and deploy.

You can customize your `Jenkinsfile` to include any steps that you need for your specific project. There are many plugins and integrations available for Jenkins that can help you to automate your build, test, and deployment processes.

Once you have created your `Jenkinsfile`, you can configure your Jenkins pipeline to use it by specifying the path to the file in your Jenkins project configuration. Jenkins will then automatically run your pipeline when triggered by a build event, such as a code push to your repository.



Build

To build the application we need to install the required packages which are mentioned in `package.json` file of `server` and `client` directory.

Once the packages are install, we can build the frontend React app using `npm run build` command which makes a optimised version of the react app which is ready to deploy.

```
stage('build') {
    steps {
        checkout scm
        sh 'cd server && npm -f install'
        sh 'cd client && npm -f install'
        sh 'cd client && npm run build'
    }
}
```

Testing

Test cases are defined for the apis which are in the `controllers` folder using `chai` and `chai-http` which are testing framework for javascript.

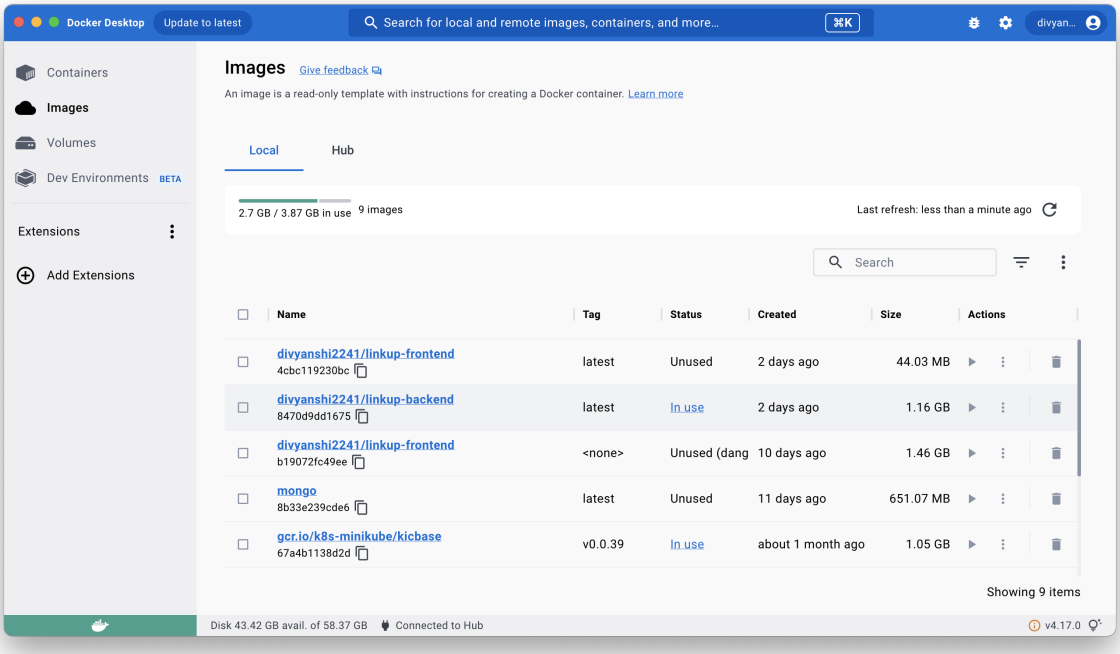
Now to run the test, we run the following command in the server directory: `npm run test`

In Jenkins pipeline, this can be done as follows:

```
stage('Testing') {
    steps {
        sh 'cd server && npm run test'
    }
}
```

Containerisation

1. To start with docker first we need to Install docker on mac, to do this just download the docker DMG from https://www.docker.com/products/docker-desktop/. On opening your docker deamon will start working and you can use all docker commands



2. Now to containerize our MERN app we need to make two Dockerfile which will basically give the steps to build an image.

- **Dockerfile for frontend**

```
FROM nginx:stable-alpine

RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d/nginx.conf

COPY build/ /usr/share/nginx/html
EXPOSE 3000
```

Overall, the Dockerfile for the frontend sets up a container that will host the compiled frontend code using the nginx web server, and exposes port 3000 to allow external access to the served content.

Finally, port 3000 is exposed using the `EXPOSE` command. This tells Docker that the container will listen on this port, and allows external processes to communicate with the container over this port.

The build directory, which contains the compiled frontend code, is then copied into the `/usr/share/nginx/html` directory of the container. This is the default directory where nginx looks for files to serve, and is where the frontend code will be hosted.

Next, a new configuration file called `nginx.conf` is copied into the container using the `COPY` command. This file contains the configuration settings for the nginx server, and is used to define how the server will handle incoming requests.

The default configuration file for nginx is removed using the `RUN` command, which executes a shell command during the build process. This is done to replace the default configuration file with a custom one that will be included later in the build process.

The Dockerfile for the frontend specifies that the base image should be `nginx:stable-alpine`, which is a lightweight and stable version of the nginx web server.

- **Dockerfile for backend**

```
FROM node:18.16
WORKDIR /app
COPY . .
RUN npm install --force
EXPOSE 3001
CMD ["npm", "start"]
```

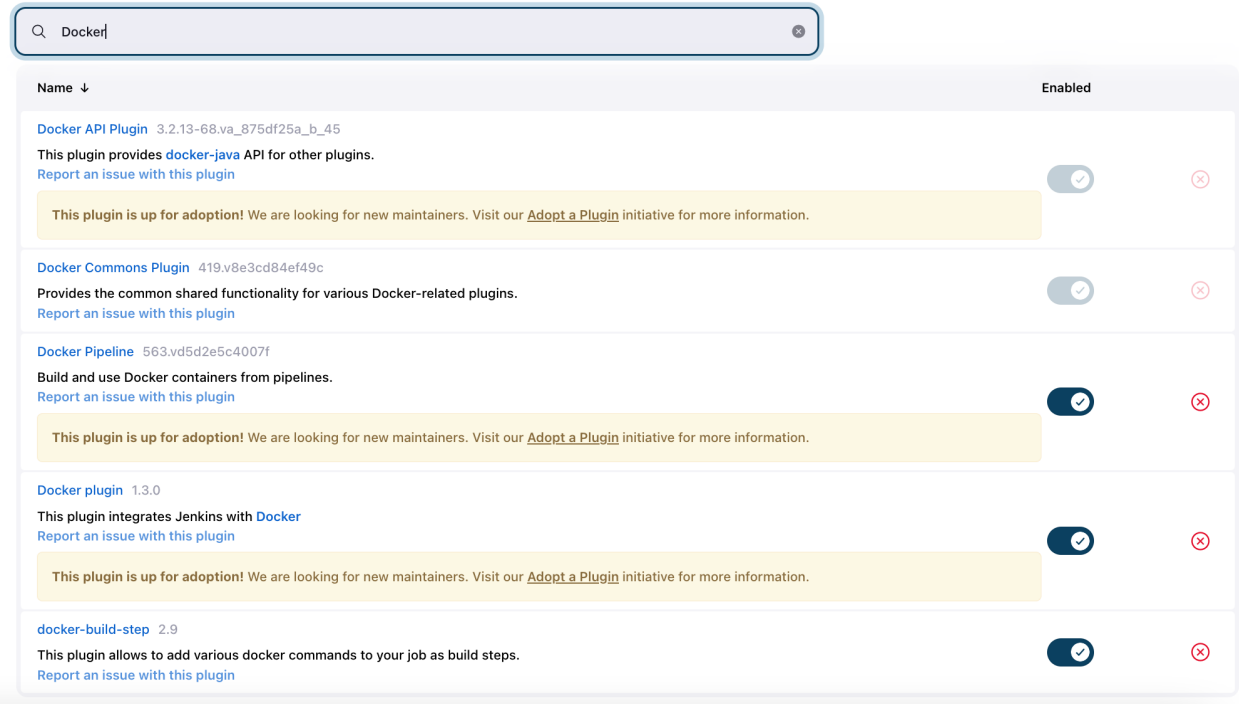
The Dockerfile for the backend specifies that the base image should be `node:18.16`, which is a pre-built image of the Node.js runtime environment. The working directory for the container is set to `/app`, and the contents of the current directory are copied to the container's `/app` directory.

The `RUN` command installs the Node.js dependencies listed in the `package.json` file using the `npm install` command. The `EXPOSE` command specifies that the container will listen on port 3001, which is the default port used by the Node.js server.

Finally, the `CMD` instruction specifies the command that should be run when the container starts - in this case, `npm start`. This command starts the Node.js server, which listens for incoming requests on port 3001.

Overall, the Dockerfile for the backend sets up a container that hosts the Node.js server, and exposes port 3001 to allow external access to the server.

- Now to build this images using Jenkins, we need download few plugins for Docker



- After adding this plugins, restart your Jenkins and you can use these functions in your pipeline scripts. Now the pipeline script for this stage looks as follow.

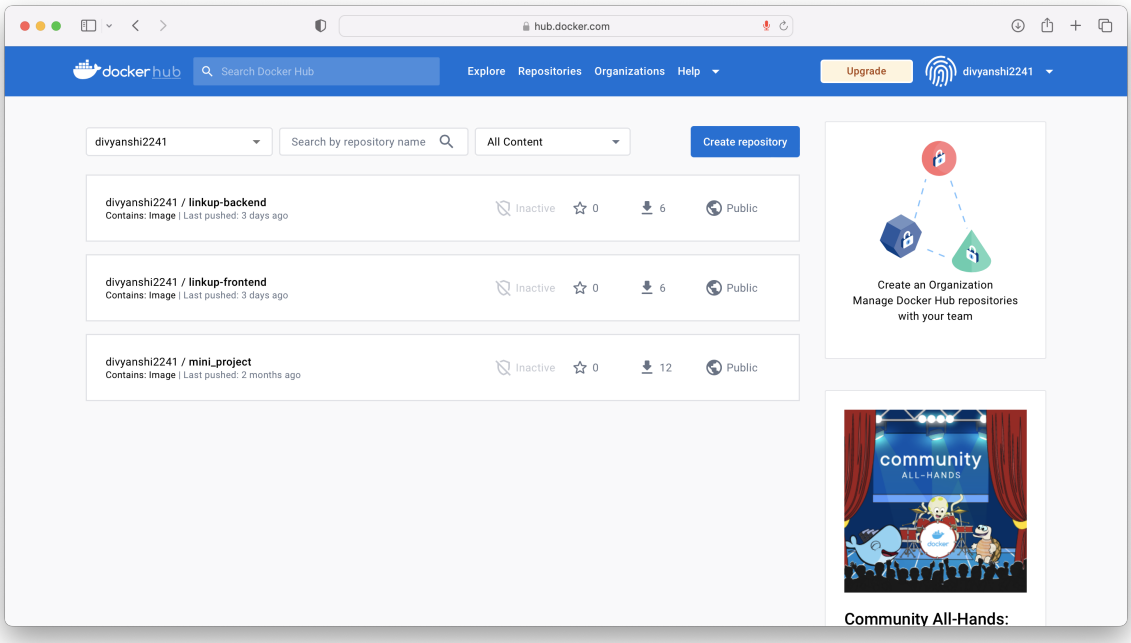
```
stage('Build and Push Docker Image - frontend') {
    steps {
        script{
            def dockerfileDir = "./client"
            dockerImage = docker.build(frontend + ":latest", "--file ${dockerfileDir}/Dockerfile ${dockerfileDir}")
            docker.withRegistry('', registryCredential) {
                dockerImage.push()
            }
        }
    }
}

stage('Build and Push Docker Image - backend') {
    steps {
        script{
            def dockerfileDir = "./server"
            dockerImage = docker.build(backend + ":latest", "--file ${dockerfileDir}/Dockerfile ${dockerfileDir}")
            docker.withRegistry('', registryCredential) {
                dockerImage.push()
            }
        }
    }
}
```

- We have added few these variable in `environment` at top of the script

```
environment {
        frontend = "divyanshi2241/linkup-frontend"
        backend = "divyanshi2241/linkup-backend"
        registryCredential = "4e7af052-eef1-43a5-8beb-5264998fc9c6"
        dockerImage = ""
 }
```

- This is how you can build a docker image locally, but using Jenkins plugin you can absract these commands. After these step in your pipeline, you should be able to see these images in your dockerhub repository.



Container Orchestration

Docker Compose

Docker Compose is a tool that allows you to define and run multi-container Docker applications. It is a YAML file that defines the services, networks, and volumes needed for a Docker application. Docker Compose makes it easier to manage Docker containers by allowing you to define and start them all at once.

Here are some basic Docker Compose commands:

- `docker-compose up` - starts the containers defined in the docker-compose.yml file.
- `docker-compose down` - stops and removes the containers defined in the docker-compose.yml file.
- `docker-compose ps` - lists all the containers in the Docker Compose project.

We started by creating a `docker-compose` file to understand how different images and network services work together.

```
version: "3.8"

services:
  frontend:
    image: divyanshi2241/linkup-frontend:latest
    ports:
      - "3000:3000"
    depends_on:
      - backend

  backend:
    image: divyanshi2241/linkup-backend:latest
    ports:
      - "3001:3001"
    environment:
      - MONGO_URL=mongodb://mongo:27017/myapp
      - PORT=3001
      - JWT_SECRET='socialMediaIsReallyCool'
    depends_on:
      - mongo

  mongo:
    image: mongo:latest
    ports:
      - "27017:27017"
    volumes:
      - ./data/db:/data/db:rw
      - ./data/configdb:/data/configdb:rw
```

This Docker Compose file describes three services: `frontend`, `backend`, and `mongo`.

The `frontend` service uses the image `divyanshi2241/linkup-frontend:latest` and maps port 3000 on the host to port 3000 in the container. It also depends on the `backend` service.

The `backend` service uses the image `divyanshi2241/linkup-backend:latest` and maps port 3001 on the host to port 3001 in the container. It sets several environment variables, including `MONGO_URL`, which points to the `mongo` service, and `JWT_SECRET`. It also depends on the `mongo` service.

The `mongo` service uses the official `mongo` image and maps port 27017 on the host to port 27017 in the container. It also mounts two volumes, `./data/db` and `./data/configdb`, to persist data outside of the container.

Together, these services create a multi-container application that includes a frontend, a backend, and a database, all of which can be started and stopped together using Docker Compose.

While Docker Compose is a good tool for defining and running multi-container Docker applications on a single host, it has it limitations when managing a slightly large application which has to be hosted on multiple servers and a scale on demand. So in the interest of learning and completing the pipeline, we decided to go ahead and use Kubernetes for actual deployment.

Kubernetes

Kubernetes is a more powerful and scalable container orchestration platform that can manage containerized applications across multiple hosts. Kubernetes is designed to handle large-scale production deployments with features like automatic scaling, rolling

updates, and self-healing capabilities. It provides a rich set of features for managing containerized applications, including load balancing, service discovery, and container storage management.

Here's a high-level overview of how Kubernetes works:

- You define your application and infrastructure requirements in a Kubernetes manifest file called a `resource file`, which describes the desired state of your application.
- You use the Kubernetes API to submit the manifest file to the Kubernetes control plane, which is a set of components that manage the overall state of the cluster.
- The Kubernetes control plane schedules the application components onto worker nodes, which are responsible for running the containers that make up your application.
- The Kubernetes control plane monitors the state of the worker nodes and the application components running on them, and takes action to maintain the desired state of the system.
- If a worker node fails or becomes unavailable, Kubernetes automatically reschedules the application components onto other available nodes to maintain availability.

So to make our application run on Kubernetes, we defined few resources file, which are YAML file that describes a Kubernetes object. A Kubernetes object represents a persistent entity in the Kubernetes system, such as a Pod, Service, Deployment, or ConfigMap.

Here are the resource file we created for backend, frontend and mongodb:

- Frontend deployment and service - `frontend-app.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
  labels:
    app: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: linkup-frontend
        image: divyanshi2241/frontend:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  type: LoadBalancer
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000
```

- Backend deployment and service - `backend-app.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
  labels:
    app: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: linkup-backend
        image: divyanshi2241/backend:latest
        imagePullPolicy: Never
        ports:
        - containerPort: 3001
        env:
        - name: MONGO_URL
          value: mongo-service
        - name: JWT_SECRET
          value: socialMediaIsReallyCool
        - name: PORT
          value: '3001'
        - name: ELASTIC_USERNAME
          value: elastic
        - name: ELASTIC_PASSWORD
          value: gA7Zr9ZNHazQ3uFrJeLBXxtD
        - name: ELASTIC_CLOUD_ID
          value: 4c9b00c9b6bb4f859d07abe312d0d965:dXMtY2VudHJhbDEuZ2NwLmNsb3VkLmVzLmlvOjQ0MyRmNWU1M2I4MTNlZjU0OGY5YmNhZTk5OTA5M2QzNmQ3ZSRmM
        volumeMounts:
             - name: varlog
               mountPath: /var/log
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 3001
      targetPort: 3001
```

- MongoDB deployment and service - `mongo-app.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo-deployment
  labels:
    app: mongo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
```

```
      - name: mongo
        image: mongo:latest
        ports:
        - containerPort: 27017
---
apiVersion: v1
kind: Service
metadata:
  name: mongo-service
spec:
  type: NodePort
  selector:
    app: mongo
  ports:
    - protocol: TCP
      port: 27017
      targetPort: 270
```

As describe in docker compose, these files creates network bridges between different containers which are running as pods inside the Kubernetes cluster, it also describe which images to pull, what are the ports which needs to be expose, number of replicas, etc.

Now to start with Kubernetes we have to download `kubectl` and `minikube`

- To install kubectl on mac: `brew install kubectl`
- To install minikube on mac refere this: https://minikube.sigs.k8s.io/docs/start/

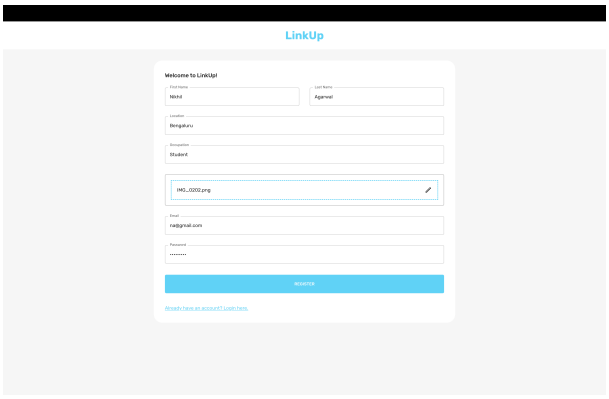
Following the next few steps, we were able to run our application using Kubernetes on minikube locally.

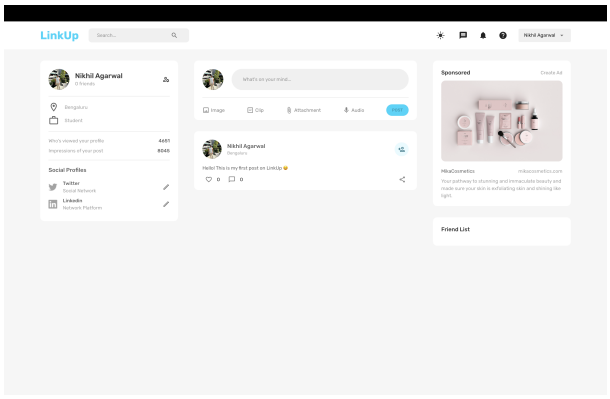
```
minikube start
kubectl apply -f mongo-app.yaml
kubectl apply -f backend-app.yaml
kubectl apply -f frontend-app.yaml

minikube tunnel
```

💡 tMinikube tunnel is a feature of Minikube that enables you to expose services running inside your Minikube cluster to the outside world. By default, services running inside Minikube are only accessible from within the cluster, but Minikube tunnel creates a secure tunnel between your local machine and the cluster, allowing you to access the services from outside the cluster.

Now if you go to `localhost:3000`, you will be able to see our frontend working





Configuration Management

For configuration management for deploying our application to host machines, we are using Ansible. It uses a combination of playbooks and inventory files to define and manage configurations.

A playbook in Ansible is a collection of tasks that are organized in a file and executed on a set of hosts defined in an inventory file. The tasks in a playbook are written in YAML format and can be used to perform a variety of operations, such as installing packages, managing services, copying files, and executing commands.

```
# playbook.yml
- name: pull updated image
  hosts: all
  pre_tasks:
    - name: Check Minikube's status.
      command: minikube status
      register: minikube_status
      changed_when: false
      ignore_errors: true
    - name: Start Minikube if it's not running.
      command: minikube start --vm-driver=docker --cpus 4 --memory 9152
      when: "not minikube_status.stdout or 'Running' not in minikube_status.stdout"
    - name: "Install kubernetes python package"
      pip:
        name: kubernetes
        state: present
  tasks:
    - name: Delete any existing mongo deployment
      kubernetes.core.k8s:
        src: ./k8/mongo-app.yaml
        state: absent
        namespace: default
    - name: Create a deployment for mongo
      kubernetes.core.k8s:
        src: ./k8/mongo-app.yaml
        state: present
        apply: true
        namespace: default
    - name: Delete any existing deployment of Backend Deployment
      kubernetes.core.k8s:
        src: ./k8/backend-app.yaml
        state: absent
        namespace: default
    - name: Create a deployment for Backend
      kubernetes.core.k8s:
        src: ./k8/backend-app.yaml
        state: present
        apply: true
        namespace: default
    - name: Delete any existing deployment of Frontend Deployment
      kubernetes.core.k8s:
        src: ./k8/frontend-app.yaml
        state: absent
        namespace: default
    - name: Create a deployment for Frontend
      kubernetes.core.k8s:
        src: ./k8/frontend-app.yaml
        state: present
        apply: true
        namespace: default
    - name: Delete any existing deployment of Filebeat
      kubernetes.core.k8s:
        src: ./k8/file-beat-config.yaml
        state: absent
        namespace: default
    - name: Create a deployment for Filebeat
      kubernetes.core.k8s:
        src: ./k8/file-beat-config.yaml
        state: present
        apply: true
        namespace: default
    - name: Delete any existing deployment of Logstash
      kubernetes.core.k8s:
        src: ./k8/logstash.yaml
        state: absent
        namespace: default
    - name: Create a deployment for Logstash
      kubernetes.core.k8s:
```

```
        src: ./k8/logstash.yaml
        state: present
        apply: true
        namespace: default
```

- The playbook starts by checking the status of Minikube, a tool for running Kubernetes locally, using the `minikube status` command. If Minikube is not running, the playbook starts it by running the `minikube start` command with some additional options.
- The playbook then installs the `kubernetes` Python package using the `pip` module, which is used to interact with the Kubernetes API.
- The playbook then proceeds to deploy several Kubernetes resources using the `kubernetes.core.k8s` module.
- For each resource, the playbook first deletes any existing deployments with the same name and in the same namespace, using the `state: absent` argument. It then creates a new deployment using the `state: present` argument and the `apply: true` argument.

Here is the inventory file for a host machine

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

Now to run ansible, use the following command

```
ansible-playbook -i inventory playbook.yml
```

```
● (base) ➜  LinkUp git:(main) ✗ ansible-playbook -i inventory playbook.yml

PLAY [pull updated image] ******************************************************************************************

TASK [Gathering Facts] *********************************************************************************************
[WARNING]: Platform darwin on host localhost is using the discovered Python interpreter at
/opt/homebrew/bin/python3.11, 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.
ok: [localhost]

TASK [Check Minikube's status.] ************************************************************************************
ok: [localhost]

TASK [Start Minikube if it's not running.] *************************************************************************
skipping: [localhost]

TASK [Install kubernetes python package] ***************************************************************************
ok: [localhost]

TASK [Delete any existing mongo deployment] ************************************************************************
ok: [localhost]

TASK [Create a deployment for mongo] *******************************************************************************
changed: [localhost]

TASK [Delete any existing deployment of Backend Deployment] ********************************************************
ok: [localhost]

TASK [Create a deployment for Blacklist Service] *******************************************************************
changed: [localhost]

TASK [Delete any existing deployment of Frontend Deployment] *******************************************************
ok: [localhost]

TASK [Create a deployment for Frontend Service] ********************************************************************
changed: [localhost]

TASK [Delete any existing deployment of Filebeat] ******************************************************************
changed: [localhost]

TASK [Create a deployment for Filebeat] ****************************************************************************
changed: [localhost]

TASK [Delete any existing deployment of Logstash] ******************************************************************
changed: [localhost]

TASK [Create a deployment for Logstash] ****************************************************************************
changed: [localhost]

PLAY RECAP *********************************************************************************************************
localhost                  : ok=13   changed=7    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
```

Now that we know it works, we can add it in our Jenkins pipeline like this:

```
stage('Deploy') {
    steps {
        echo 'Deploying locally..'
        sh 'ansible-playbook -i inventory playbook.yml'
        echo 'Done Deploying'
    }
}
```

Monitoring

Logging using Winston

Winston is a popular logging library for Node.js that provides a flexible and extensible way to log messages in a variety of formats and transports.

We define a logger function as follows:

```
import { createLogger, format, transports, config } from "winston";
const { combine, timestamp, json } = format;

const logger = createLogger({
  transports: [
    new transports.Console(),
    new transports.File({ filename: "../var/log/app.log", timestamp: true }),
  ],
  format: combine(
    timestamp({
      format: "YYYY-MM-DD HH:mm:ss",
    }),
    json()
  ),
});

export default logger;
```

This logger functions logs the message into the console and a file called `app.log`

Filebeat and Logstash configuration

Filebeat and Logstash are two tools commonly used together for logs and monitoring purposes in a distributed system architecture. Here's how they work:

- Filebeat reads log files or other sources and sends the log data to Logstash.
- Logstash processes the log data, such as parsing the log lines, adding metadata, and applying filters (**grok** pattern).
- Logstash sends the processed log data to Elasticsearch or another data store for indexing and analysis.
- Kibana can be used to visualize and analyze the log data in real-time or historical data.

To use Filebeat and Logstash with our kubernetes pod, We created `logstash.yaml` and `file-beat-config.yaml` in which we first mount the logs directory, now using the same mounted directory we use Filebeat (which is running as a pod) to ship log files to logstash (running as a pod). Now logstash process these logs and sends them to Cloud deployement of elastic search.

Grok pattern

`\{%{QUOTEDSTRING:level_label}:% {QUOTEDSTRING:loglevel},\"message\":\"User % {GREEDYDATA:user_email}: %{GREEDYDATA:action}\.\",% {QUOTEDSTRING:timestamp_label}:% {QUOTEDSTRING:timestamp}\}`

This grok pattern process the log level, user, user action, and timestamp from backend logs.

Here is an example:

```
{"level":"info","message":"User n@gmail.com: created a post.","timestamp":"2023-05-15 12:50:52"}

# after processing through grok pattern
{
            "timestamp_label": "timestamp",
            "action": "created a post",
            "user_email": "n@gmail.com",
            "level_label": "level",
            "loglevel": "info",
            "timestamp": "2023-05-15 12:50:52"
}
```

Kibana

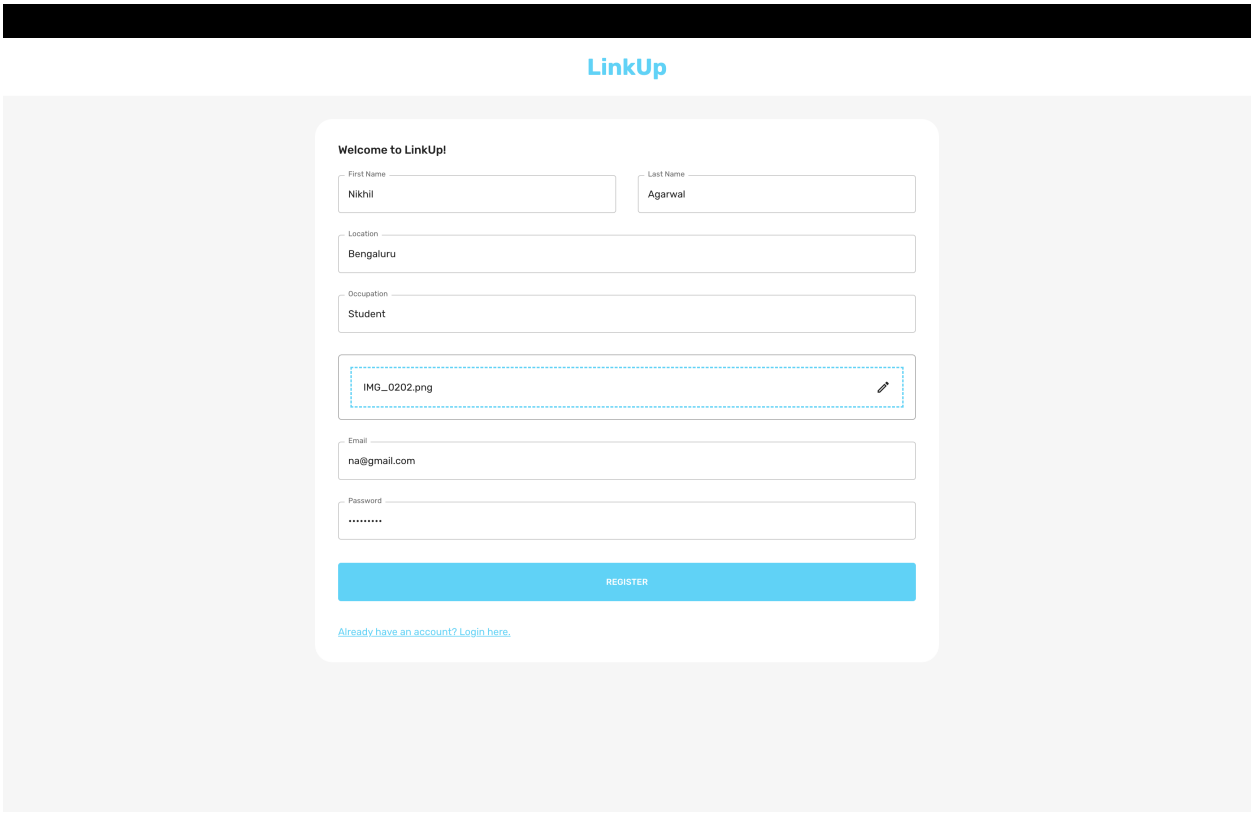
Using the above grok pattern helps us to see how the user is interacting with the social media, each of the actions like like/unlike, friend/unfriend, login, new user is included in the logs along with the user email ID. Also log level is show to tell whether the log is for information, error, warning, etc. Finally timestamp helps to know the exact time at which the action was performed by user.

Using these logs data in Elastic cloud deployment, we can use kibana to create visualisation dashboard to understand user interaction.

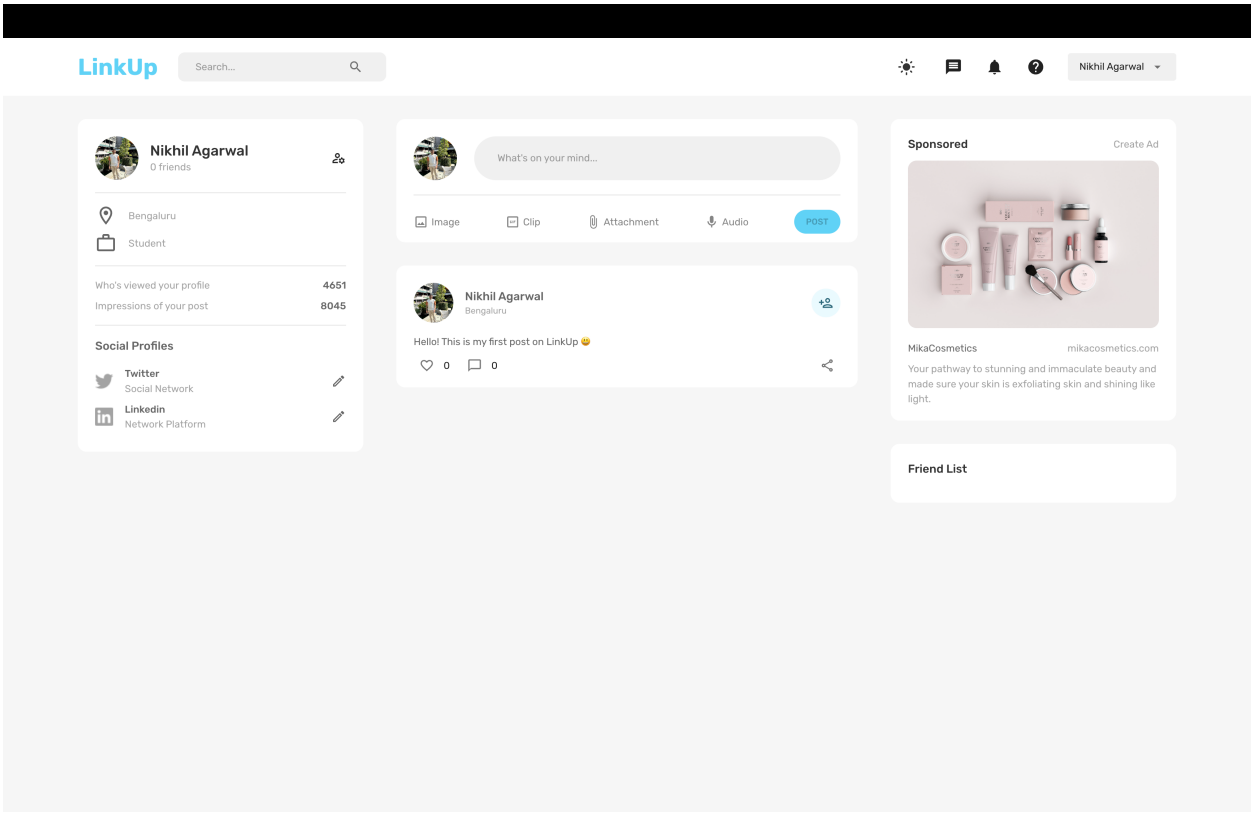
Example plot of user activity, which shows the percentage of activity the user did in the application.

Snippets of the Application

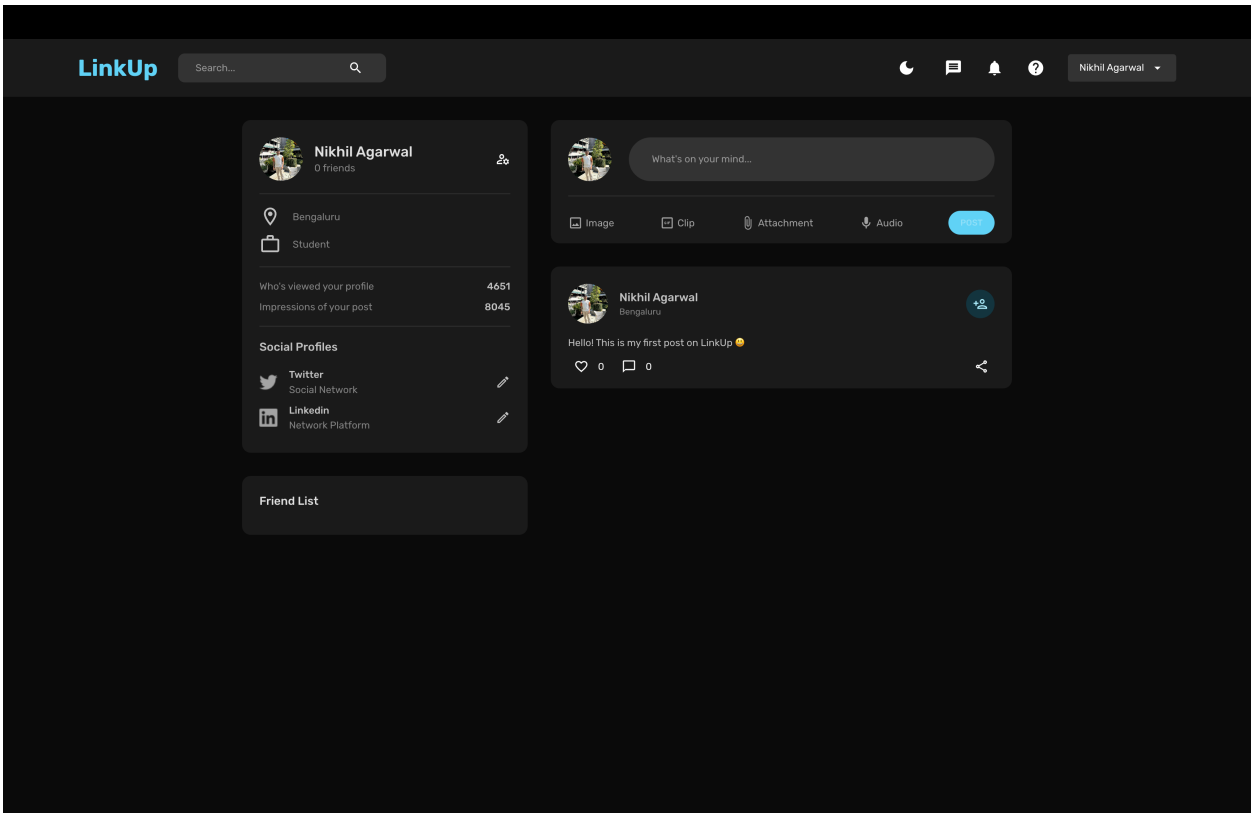
Sign Up Page



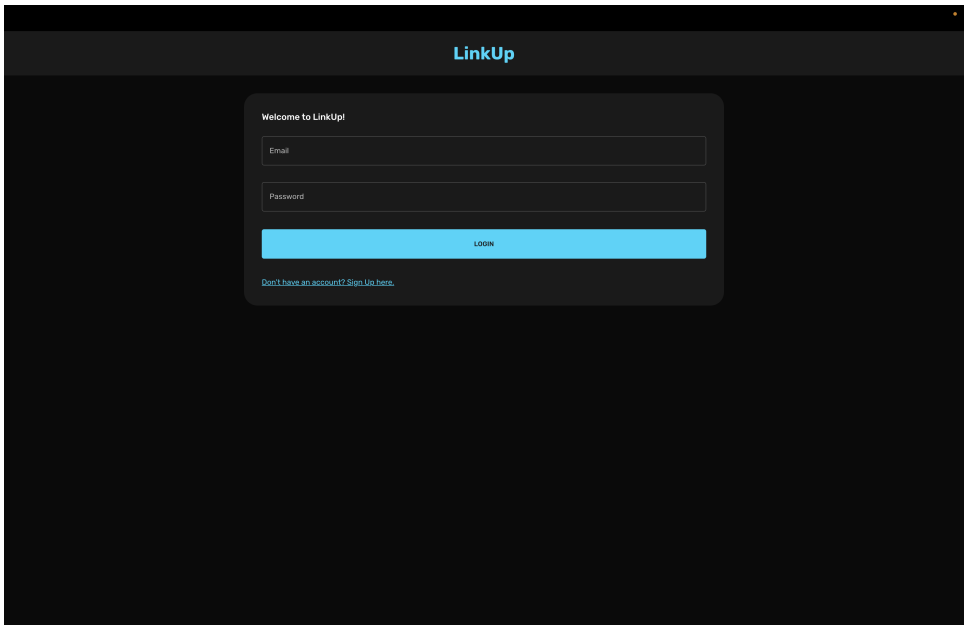
Home Page



Profile Page in Dark Mode



Login Page



Dark Mode, with Likes and Posts