CS 816 Software Production Engineering

Final Project - Chat Application

Yukta Rajapur (IMT2021066)
Nilay Kamat(IMT2021096)

December 2024

Contents

Chapter 1

Introduction and Setup

1.1 Introduction

Our project focuses on deploying and managing a real-time chat application built with Spring Boot for backend and ReactJS for frontend. The application enables users to send and receive messages through a WebSocket-based back-end for real-time communication, paired with a user-friendly front-end interface. The project integrates key DevOps tools such as Docker for containerization, Jenkins for automated CI/CD pipelines, Kubernetes for scalable deployment, and Ansible for configuration management. The goal of the project is to emphasize the DevOps tools and workflows used to develop, deploy, and monitor applications effectively.

1.2 Objective

The specific objectives of this project are:

1. Develop a full-stack application with a robust back-end and interactive front-end.
2. Set up version control with **Git** and integrate with a remote repository on **GitHub**.
3. Automate the build, test, and deployment process using **Jenkins** pipelines.
4. Containerize the application using **Docker** and manage images with a remote container registry.
5. Deploy the containerized application to a **Kubernetes** cluster for orchestration and scalability.

6. Use **Ansible** for configuration management and deployment automation.
7. Implement **testing** of backend to ensure code reliability.
8. Set up centralized **logging** and monitoring using the **Kibana for visualizations** and **Grafana, Loki and Prometheus** for enhanced observability.

1.3 Tools Used

1. **IntelliJ IDEA Ultimate:** A *powerful IDE and Code Editor* for Java and other languages, offering advanced development, debugging and integration features for seamless coding and deployment.
2. **Git & GitHub:** Git is a *version control system and source code management* tool that allows developers to collaborate on code and track changes. GitHub is a web-based platform for *version control and collaboration*, allowing developers to host, review, and manage code repositories using Git.
3. **Apache Maven:** A *build automation* tool that helps manage dependencies and build Java-based projects.
4. **Jenkins:** a continuous integration and continuous delivery (CI/CD) tool that automates the build, test, and deployment processes.
5. **Docker:** a *containerization platform* that enables developers to package applications and dependencies into portable, lightweight containers.
6. **Ansible:** a *configuration management tool* that automates the deployment and management of infrastructure and applications.
7. **GitHub Webhooks:** a tool that *triggers automated actions* when specific events occur in a GitHub repository.
8. **JUnit:** an open-source *testing framework* for Java that enables developers to write and run unit tests.
9. **Monitoring Stack**: A set of tools *(Prometheus, Loki, Grafana, and Kibana)* for collecting, aggregating, and visualizing metrics and logs, providing insights into application performance and health.

10. **Kubernetes**: A *container orchestration* platform that automates the deployment, scaling, and management of containerized applications.

1.4 Setup

We need to install and setup some tools and frameworks before getting started on the workflow.

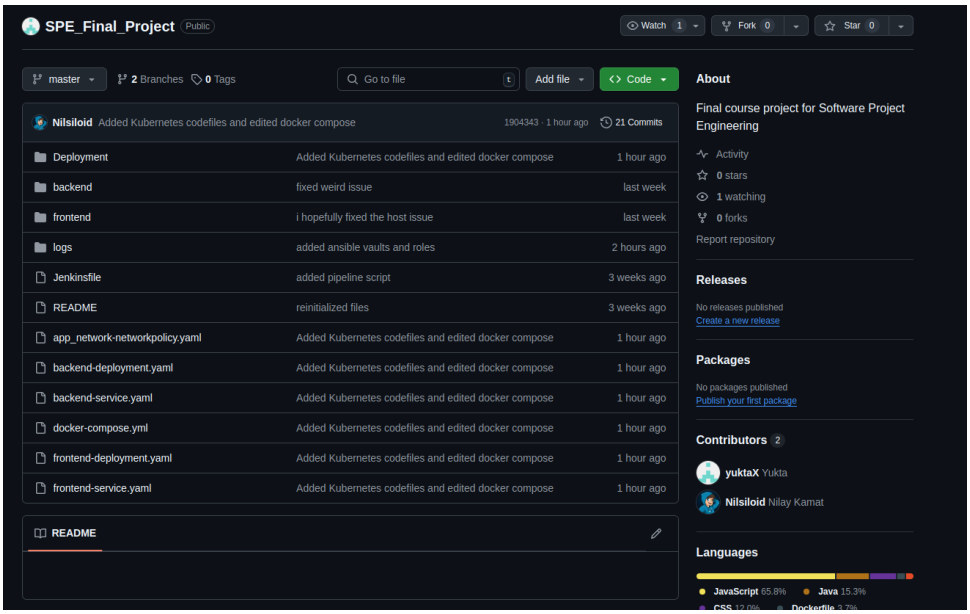
1.4.1 Git & GitHub

To install Git and check the version, we use the following commands:

```
1 sudo apt install git
2
3 git --version
```

```
nilay@Nilay-Lenovo-V14:~/IIITB/SEM7/SPE/Calculator$ git --version
git version 2.34.1
nilay@Nilay-Lenovo-V14:~/IIITB/SEM7/SPE/Calculator$
```

Visit GitHub and create an account if you don't have one. Then create a new repository named "SPE_Final_Project".



1.4.2 Maven

To install Maven and check the version, we use the following commands:

```
nilay@Nilay-Lenovo-V14:~/IIITB/SEM7/SPE/Calculator$ mvn --version
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfcdc97d260186937)
Maven home: /home/nilay/IIITB/SEM7/SPE/apache-maven-3.9.9
Java version: 17.0.12, vendor: Ubuntu, runtime: /usr/lib/jvm/java-17-openjdk-amd64
Default locale: en_IN, platform encoding: UTF-8
OS name: "linux", version: "6.8.0-40-generic", arch: "amd64", family: "unix"
nilay@Nilay-Lenovo-V14:~/IIITB/SEM7/SPE/Calculator$
```

1.4.3 Jenkins

To install Jenkins and check the version, we use the following commands:

```
sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \
  https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key

echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc]" \
  https://pkg.jenkins.io/debian-stable binary/ | sudo tee \
  /etc/apt/sources.list.d/jenkins.list > /dev/null

sudo apt-get update

sudo apt-get install jenkins
```

```
sudo systemctl start jenkins

sudo systemctl status jenkins
```

1.4.4 Docker

To install Docker, check the version, and ensure that both Jenkins and Docker are in the same user group, we use the following commands:

```
sudo apt install curl
curl -fsSL https://get.docker.com -o get-docker.sh
sh get-docker.sh
sudo docker --version
```

```
nilay@Nilay-Lenovo-V14:~/IIITB/SEM7/SPE/Calculator$ sudo docker --version
[sudo] password for nilay:
Docker version 27.3.1, build ce12230
nilay@Nilay-Lenovo-V14:~/IIITB/SEM7/SPE/Calculator$
```

```
sudo tail /etc/gshadow
sudo usermod -aG docker jenkins
sudo systemctl restart jenkins
```

1.4.5 Ansible

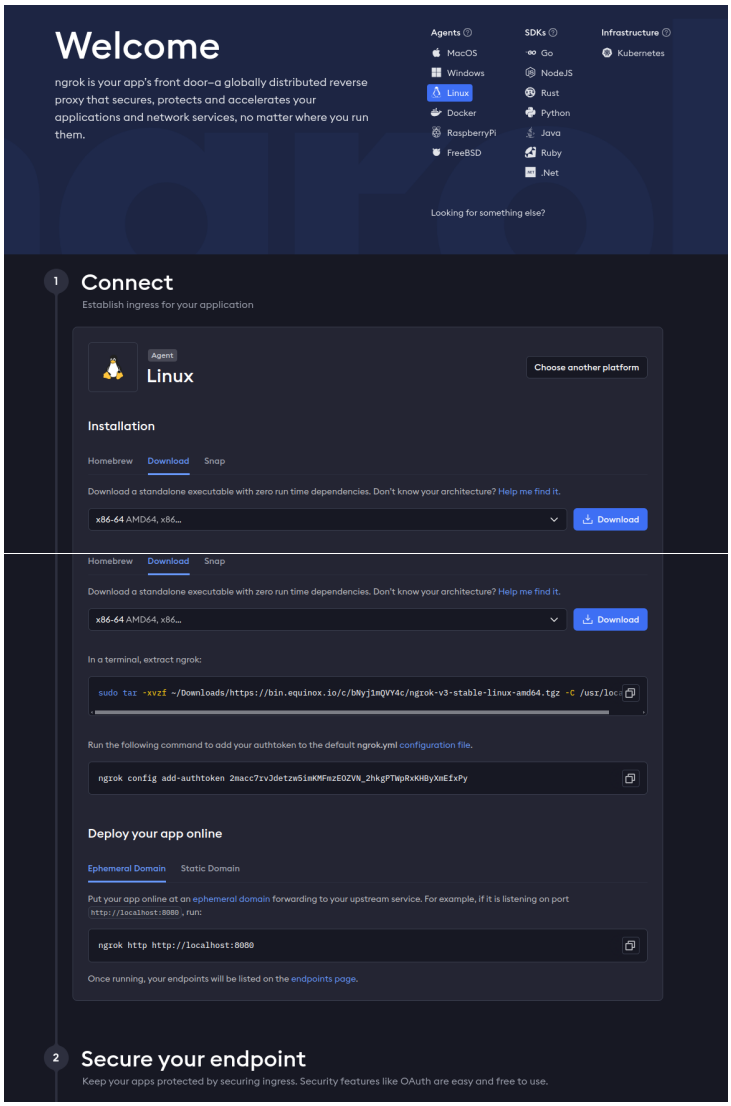
To install Ansible and check the version, we use the following commands:

```
sudo apt-add-repository ppa:ansible/ansible
sudo apt update
sudo apt install ansible
```

```
nilay@Nilay-Lenovo-V14:~/IIITB/SEM7/SPE/Calculator$ sudo ansible --version
ansible 2.10.8
  config file = None
  configured module search path = ['/root/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3/dist-packages/ansible
  executable location = /usr/bin/ansible
  python version = 3.10.12 (main, Sep 11 2024, 15:47:36) [GCC 11.4.0]
nilay@Nilay-Lenovo-V14:~/IIITB/SEM7/SPE/Calculator$
```

1.4.6 Ngrok

Visit the Ngrok official website and sign up for a free account. Then download the Ngrok binary executable for your OS.



Once you have downloaded ngrok, you run the following command in the same directory that you have downloaded ngrok, to extract it to the destination:

```
sudo tar xvzf ~/Downloads/ngrok-v3-stable-linux-amd64.tgz -C /usr/local/bin
```

Ensure you replace ngrok-v3-stable-linux-amd64.tgz with the appropriate version of the zip you downloaded. Once you do that, you run the following command:

```
ngrok config add-authtoken 2macc7rvJdetzw5imKMFmzEOZVN_2hkgPTWpRxKHByXmEfxPy
```

Replace the auth token with what is shown in your account dashboard.

1.4.7 Kubernetes

To use Kubernetes, we will install *kubectl*, the command-line tool for interacting with Kubernetes clusters, *minikube* - a lightweight tool that creates a local Kubernetes cluster on your machine.

```
sudo snap install kubectl --classic

curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
chmod +x minikube
sudo mv minikube /usr/local/bin/

minikube start --driver=docker
minikube status
```

Chapter 2

Project Implementation

2.1 Workflow

1. **Code Development:** Utilised IntelliJ IDEA Ultimate and VSCode to develop the frontend and backend for the real-time chat application. The frontend was built using React.js, while the backend was developed with Spring Boot, implementing WebSocket-based communication for real-time messaging.

2. **Version Control/Source Code Management:** Stored the code in a Git repository hosted on GitHub, enabling collaborative development and version control. GitHub also facilitated continuous integration and deployment (CI/CD) workflows.

3. **Building:** Employed Apache Maven as the build tool to automate the compilation and packaging of the Spring Boot backend. For the React.js frontend, npm was used for building the production-ready application.

4. **Continuous Integration and Continuous Delivery (CI/CD):** Used Jenkins as the CI/CD tool to automatically pull code from GitHub, run unit tests, build the frontend and backend, and deploy the application to different environments. Integration with GitHub webhooks ensured that updates in the codebase triggered automated builds and deployments.

5. **Containerisation:** Docker was used to containerize both the frontend and backend applications, ensuring consistent deployment across various environments. Each service was packaged into a separate Docker image for easy scalability and deployment.

6. **Kubernetes Deployment:** Deployed the containerized applications on a Kubernetes cluster to achieve high availability and scalability. Kubernetes managed the deployment, scaling, and load balancing of the application services.

7. **Horizontal Pod Autoscaling (HPA):** Configured HPA within Kubernetes to automatically scale the backend services based on CPU usage or other metrics, ensuring optimal performance and resource allocation under varying loads.

8. **Configuration Management:** Utilised Ansible to automate the configuration of infrastructure resources. Ansible roles were used to configure Kubernetes clusters, deploy applications, and manage configurations across multiple environments, ensuring consistency in infrastructure management.

9. **Git SCM Polling and Build Automation:** Employed Ngrok to create a secure tunnel for GitHub webhooks, enabling automatic triggering of Jenkins builds whenever code is updated in the GitHub repository.

2.2 Code Development and Source Code Management

We have built a chat application project that allows users to communicate with each other in real-time. It is built using the Spring Boot framework for the backend, SockJS for WebSocket communication, and React for the frontend. The project utilizes Apache Maven as the build tool for the backend.

Source Code Management tools allow developers to collaborate on code, track changes and maintain version control of their codebase. They provide features such as branching and merging, which allow developers to work on different versions of the codebase and merge changes together.

Tech Stack used:

1. Spring Boot: A Java-based framework used for building the backend server and handling business logic.

2. SockJS: A WebSocket emulation library that enables real-time communication between the server and clients.

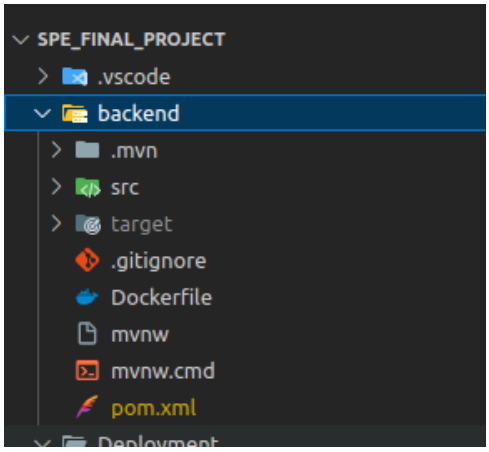
3. React: A JavaScript library used for building the user interface and handling frontend

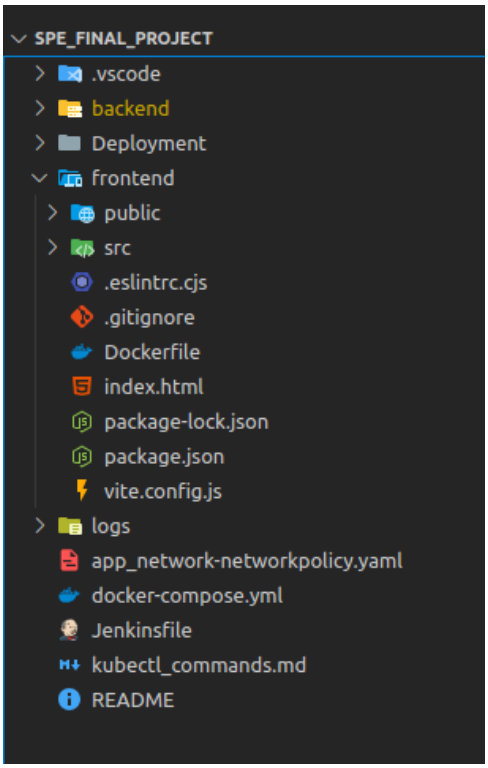
functionality.

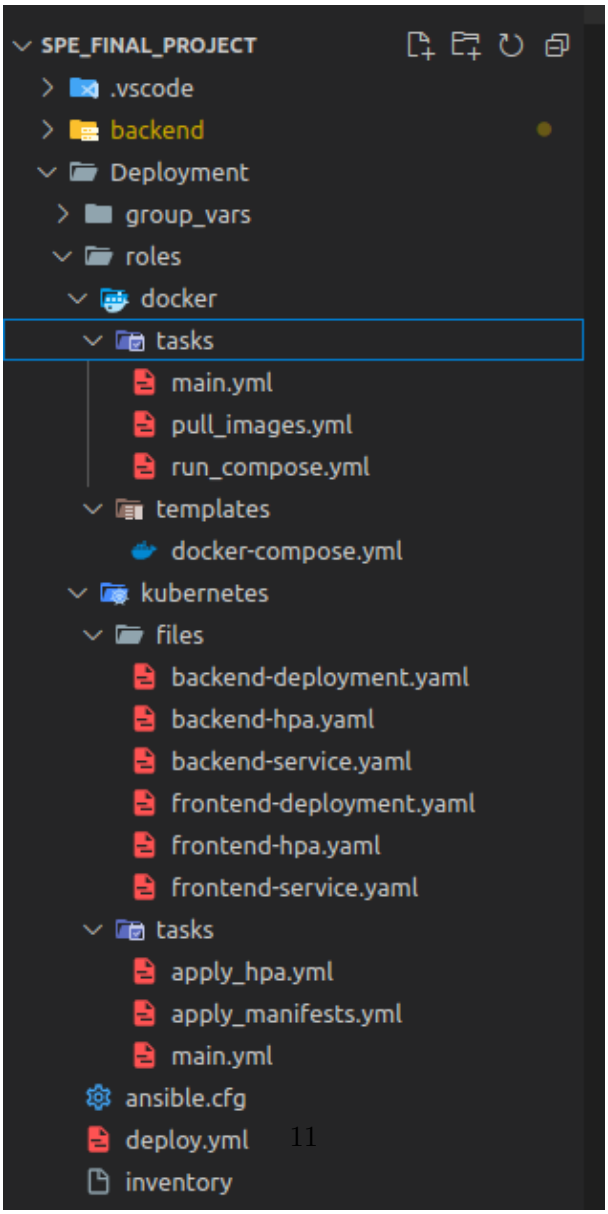
4. Kibana, Grafana - Tool used for Log Visualisation
5. Loki and Prometheus - Loki is a log aggregation tool, Prometheus is a metrics monitoring tool
6. Testing - JUnit, Mockito
7. Build - Maven
8. SCM - Git and GitHub

Git Workflow and Directory Structure: The follwing are the commands used to initialise a new Git repository, add files to the staging area, commit changes, set up a remote repository on GitHub, and push the changes to the remote repository.

```
git init
git add .
git commit -m "<message>"
git status
git remote add origin <github_repository_url>
git push -u origin main
<enter username>
<enter personal access token>
```

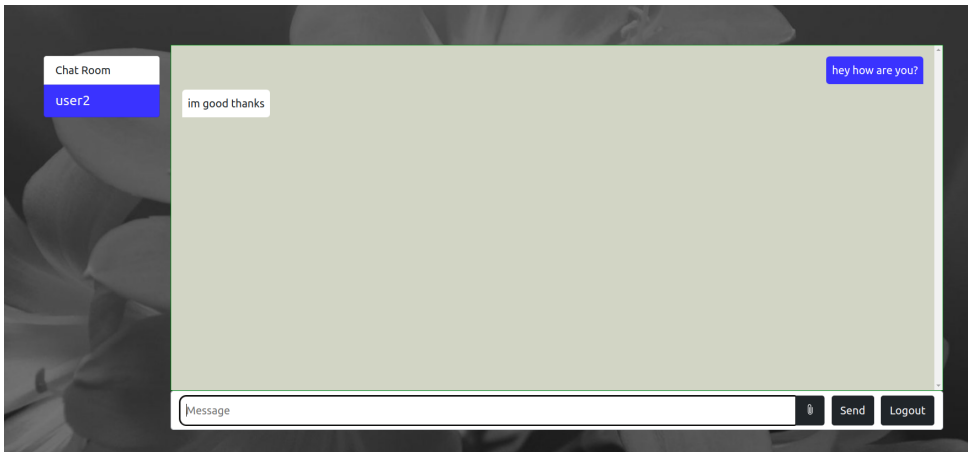

SPE_FINAL_PROJECT
.vscode
backend
.mvn
src
target
.gitignore
Dockerfile
mvnw
mvnw.cmd
pom.xml

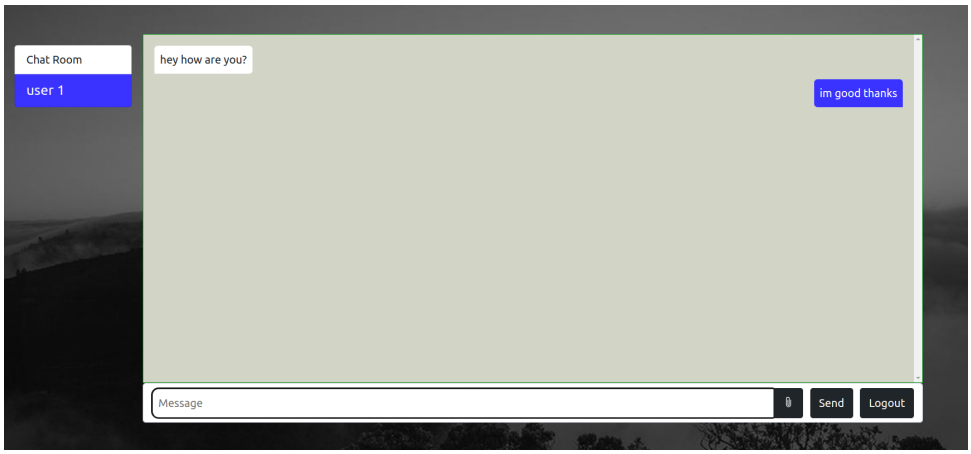

SPE_FINAL_PROJECT
.vscode
backend
Deployment
frontend
public
src
.eslintrc.cjs
.gitignore
Dockerfile
index.html
package-lock.json
package.json
vite.config.js
logs
app_network-networkpolicy.yaml
docker-compose.yml
Jenkinsfile
kubectl_commands.md
README


SPE_FINAL_PROJECT
.vscode
backend
Deployment
group_vars
roles
docker
tasks
main.yml
pull_images.yml
run_compose.yml
templates
docker-compose.yml
kubernetes
files
backend-deployment.yaml
backend-hpa.yaml
backend-service.yaml
frontend-deployment.yaml
frontend-hpa.yaml
frontend-service.yaml
tasks
apply_hpa.yml
apply_manifests.yml
main.yml
ansible.cfg
deploy.yml
inventory

Project Overview:

1. **Folders:**
   (a) **Deployment:** Contains files for Ansible.
   (b) **backend:** Contains source code for back-end implementation done using Spring Boot.
   (c) **frontend:** Contains source code for front-end implementation done using ReactJS.
2. **Features:**
   (a) **User Authentication**: Users are required to log in with a username before they can access the chat application.
   (b) **Real-time Chat**: Once logged in, users are directed to the chat page where they can send messages in the chatroom.
   (c) **User Presence**: Users are notified when new users join the chatroom, allowing them to be aware of other participants.
   (d) **Private Messages**: Users have the ability to send private messages to specific individuals.
   (e) **Multimedia Transfer**: The application supports the transfer of multimedia files, such as photos and videos.
   (f) **Logout**: Users can log out from the application, and their username will be removed from the user list displayed to other participants





2.3 Steps to Build and Run the Project

To generate a JAR file with dependencies, we run the following commands:

1. **mvn clean:** Removes the "target" folder, ensuring a fresh start for the subsequent compilation. This step eliminates any previous build artifacts.
2. **mvn compile:** Compiles the project and its associated test cases. This phase ensures the code is error-free and ready for the subsequent steps.
3. **mvn install:** Generates the JAR file. This final step packages the project, creating the desired output artifact (JAR file) once the compilation is successful.

Maven will build the project and check all test cases. Once completed, a "target" directory will be created in the current directory, which will contain the JAR file.

Now, we navigate to the "target" folder using the command: ***cd target***.

Then, we run the JAR file using the command: ***java -jar filename.jar***

Within the Maven configuration file(pom.xml), the *mainClass* tag specifies the path to the main Java file following the package structure. Additionally, the *descriptorRef* tag is employed to modify the default output JAR file name. To include project dependencies, the *dependencies* tag is utilised, enabling the addition of external libraries. In our scenario, we've incorporated dependencies such as log4j, utilised for logging functionality, and JUnit, employed for testing purposes.

Log42j.xml file:

```
%d{yyy-MM-dd HH:mm:ss.SSS}[%t] %-5level %logger{36} - %msg%n
```

The specified logging format shown above includes the following components:

1. **%dyyy-MM-dd HH:mm:ss.SSS:** Represents the timestamp with millisecond precision.
2. **%t:** Indicates the running thread's name.
3. **%-5level:** Denotes the log level with left alignment and a maximum width of 5 characters.
4. **%logger36:** Refers to the logger's name within a maximum of 36 characters.
5. **%msg:** Represents the user-written message contained in the source code.

The necessary imports for Log4j is shown in below image along with creation of an instance of the Logger in Log4j which involves using a statement to initialise a Logger object within the code.

```
import org.apache.logging.log4j.LogManager
import org.apache.logging.log4j.Logger

private static final Logger logger = LogManager.getLogger(Main.class)
```

2.4 Testing

The backend testing focused on validating the functionality, reliability, and performance of the chat application's core components. Using JUnit and Mockito, we ensured that each module behaves as expected in isolation and under different scenarios.

1. **Configuration Testing (WebSocketConfigTest.java)**
   - **Objective**: To verify the WebSocket configuration for real-time communication.
   - **Approach**:
     - Tested the `registerStompEndpoints` method to ensure the `/ws` endpoint is registered correctly with `SockJS` support.

  - Verified that allowed origin patterns are set correctly to permit cross-origin requests.
  - Validated the `configureMessageBroker` method to ensure correct prefixes for application and user destinations (`/app`, `/user`).
- **Key Tools**: Mockito to mock the `StompEndpointRegistry`.

2. **Controller Testing (ChatControllerTest.java)**
   - **Objective**: To validate the behavior of public and private messaging endpoints.
   - **Approach**:
     - Mocked the `SimpMessagingTemplate` to test private message delivery without a real broker.
     - For `receiveMessage`, ensured the controller correctly processes and returns public messages.
     - For `privateMessage`, verified that the correct user and destination are used in the `convertAndSendToUser` method.
   - **Key Tools**: Mockito for mocking dependencies and `verify` for validating method calls.
3. **Model Testing (MessageTest.java)**
   - **Objective**: To confirm the integrity of the `Message` class, which represents the data structure for messages.
   - **Approach**:
     - Conducted unit tests for getter and setter methods to ensure proper field initialization and retrieval.
     - Verified `equals`, `hashCode`, and `toString` methods for accurate comparisons and debugging outputs.
   - **Key Tools**: JUnit assertions to validate object behavior.

Testing Strategy

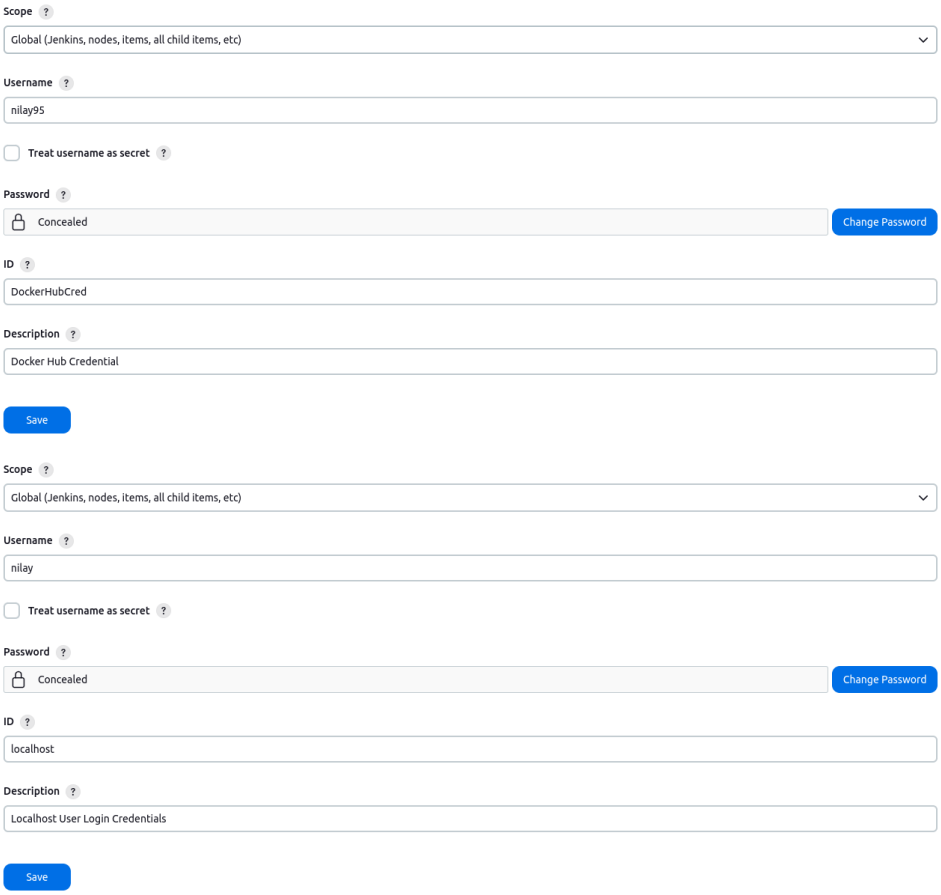
- **Unit Testing**: Focused on individual methods and classes in isolation to ensure correctness.
- **Mocking Dependencies**: Used Mockito to simulate real dependencies, such as the messaging template and WebSocket configuration registry.
- **Behavioral Verification**: Ensured expected interactions between components, especially in the controller.

This structured testing approach ensured that all critical backend functionalities of the chat application were thoroughly validated before deployment.

2.5 CI/CD using Jenkins

- Jenkins is an open-source automation tool written in Java with plugins built for continuous integration.
- Jenkins is utilized to continuously build and test software projects, simplifying the process for developers to integrate changes, and allowing users to obtain up-to-date builds with ease.
- After installing Jenkins using the procedure mentioned in the previous chapter, go to https://localhost:8080, then browse to ***Manage Jenkins*** $\longrightarrow$ ***Plugin Manager*** $\longrightarrow$ ***Available Plugins*** and install the following necessary plugins.
  - Git plugins & GitHub plugins
  - Maven Integration
  - Docker plugin & Docker pipeline
  - Ansible plugin
  - Kubernetes plugin
  - JUnit plugin

Following this, browse to ***Manage Jenkins*** $\longrightarrow$ ***Credentials*** $\longrightarrow$ ***System*** $\longrightarrow$ ***Global Credentials*** and create 2 credentials as shown in image below:



We will then establish a Jenkins pipeline comprising of 6 distinct stages:

1. **Stage 1 : Git Clone -** This stage clones the repository from the main branch of the provided GitHub URL.

```
stage('Stage 1: Git Clone'){
    steps{
        git branch: 'master',
        url:'https://github.com/yuktaX/SPE_Final_Project'
    }
}
```

2. **Stage 2 : Maven Build -** Executes the mvn clean install command to build the project and resolve dependencies using Maven.

```
stage('Stage 2: Setup Backend'){
    steps{
        sh '''
            cd backend
            mvn clean install
        '''
    }
}
```

3. **Stage 3 : Test Backend -** Executes mvn test command to test backend of application.

```
stage('Stage 3: Test Backend'){
    steps{
        sh '''
            cd backend
            mvn test
        '''
    }
}
```

4. **Stage 4 : Build and Push Backend Docker image -** Uses the Docker build process to create a Backend Docker image from the project. Authenticates using DockerHub credentials (DockerHubCred) and pushes the created image to Docker Hub.

```
stage('Stage 4: Build and Push Backend Docker Image') {
    steps {
        script {
            def backendImage = docker.build(env.BACKEND_IMAGE_NAME, './backend')
            docker.withRegistry('', 'DockerHubCred') {
                backendImage.push('latest')
            }
        }
    }
}
```

5. **Stage 5 : Build and Push Frontend Docker image -** Uses the Docker build process to create a Frontend Docker image from the project. Authenticates using DockerHub credentials (DockerHubCred) and pushes the created image to Docker Hub.

```
stage('Stage 5: Build and Push Frontend Docker Image') {
    steps {
        script {
            def frontendImage = docker.build(env.FRONTEND_IMAGE_NAME, './frontend')
            docker.withRegistry('', 'DockerHubCred') {
                frontendImage.push('latest')
            }
        }
    }
}
```

6. **Stage 6 : Clean Docker images -** Removes any stopped Docker containers and unused Docker images to free up space.

```
stage('Stage 6: Clean Docker Images') {
    steps {
        script {
            sh 'docker container prune -f'
            sh 'docker image prune -f'
        }
    }
}
```

7. **Stage 7 : Ansible Deployment -** Executes an Ansible playbook (Deployment/de-

ploy.yml) to automate the deployment of the application using the specified inventory file and settings.

```
stage('Stage 7: Ansible Deployment'){
    steps
    {
        ansiblePlaybook becomeUser: null,
        colorized: true,
        credentialsId: 'localhost',
        disableHostKeyChecking: true,
        installation: 'Ansible',
        inventory: 'Deployment/inventory',
        playbook: 'Deployment/deploy.yml',
        sudoUser: null
    }
}
```

2.6 Containerisation

- Containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure.
- Containers are lightweight, portable, and self-contained environments that enable developers to package an application with all its dependencies, libraries, and configuration files, ensuring that it runs consistently across different environments.
- We will use Docker to create containers. Docker is an open-source tool that enables developers to build, package, and deploy applications in a containerized environment.

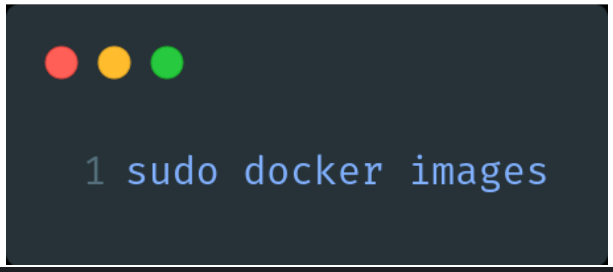
We will now create a Docker container for our project by creating a Dockerfile.

- **FROM:** We use the OpenJDK 11 base image to build our image.
- **COPY:** Copy jar file from source on the host machine into the container's file system.

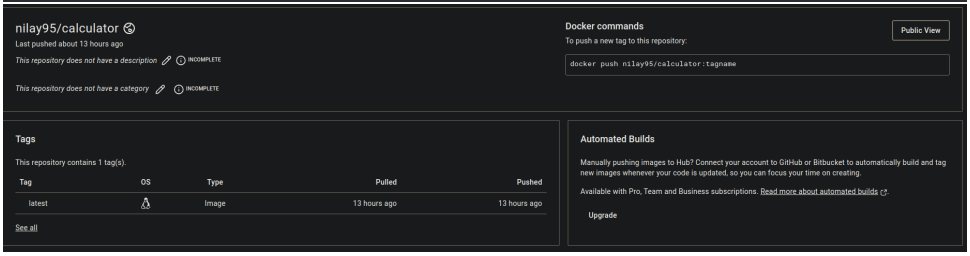
- **WORKDIR:** Changes the current working directory.
- **ENTRYPOINT:** Specify the command that should be run when a container based on the image is started.

```
FROM openjdk:11
COPY ./target/Calculator-1.0-SNAPSHOT-jar-with-dependencies.jar ./
WORKDIR ./
CMD ["java","-cp","Calculator-1.0-SNAPSHOT-jar-with-dependencies.jar","org.example.Main"]
```

The purpose of the above code is to generate a Docker container with OpenJDK version 11 as the base image which acts like a JVM. The final container can be run independently without any dependencies. The Jenkins Pipeline script has been configured to include the Dockerfile and Docker commands, automating the process of building and pushing the Docker Image to DockerHub. In the pipeline script, docker_image = docker.build "nilay95/calculator" builds the docker image. The following command allows verification of the successful creation of a Docker image.




```
nilay@Nilay-Lenovo-V14:~/IIITB/SEM7/SPE/Calculator$ sudo docker images
[sudo] password for nilay:
REPOSITORY              TAG       IMAGE ID       CREATED         SIZE
nilay95/calculator      latest    b47ed0429f8e   14 hours ago    654MB
openquantumsafe/curl    latest    99e4a58e76ed   14 months ago   47.1MB
hello-world             latest    9c7a54a9a43c   16 months ago   13.3kB
nilay@Nilay-Lenovo-V14:~/IIITB/SEM7/SPE/Calculator$
```



2.7 Configuration Management/Deployment

- We will use Ansible for local deployment. Ansible is a suite of software tools that enables infrastructure as code.

- In Ansible, managed hosts or servers which are controlled by the Ansible control node are defined in a host inventory file. The Ansible inventory file defines the hosts and groups of hosts upon which commands, modules, and tasks in a playbook operate.
- We are going to pull the images from the DockerHub and create containers using Ansible. We will create a Deployment folder and create two files named inventory and deploy.yml.

```
1   localhost ansible_user = local ansible_user = nilay
```

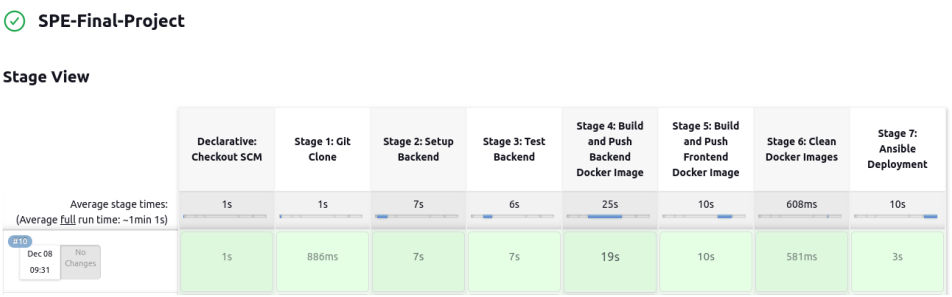
- Ansible Playbooks offer a repeatable, re-usable, simple configuration management.
- Playbooks consist of one or more plays run in a particular order. A play is an ordered set of tasks run against hosts chosen from your inventory. Plays define the work to be done. Each play contains a set of hosts to configure, and a list of tasks to be executed.

deploy.yml M

Deployment > deploy.yml

```
1  ---
2  - name: Deploy Full Stack Application
3    hosts: localhost
4    vars_files:
5      - group_vars/all/vault.yml # Includes sensitive variables
6    roles:
7      # - docker
8      - kubernetes
9
```

- The Jenkins Pipeline script has been configured to execute the Ansible Playbook automatically.
- At the end of these 7 steps of the workflow, the Jenkins Pipeline is completed and the stage view should be as follows:



2.8 Kubernetes

Kubernetes, commonly abbreviated as K8s, is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. In this project, Kubernetes was used to manage the backend and frontend services of the application, ensuring high availability, scalability, and efficient resource usage.

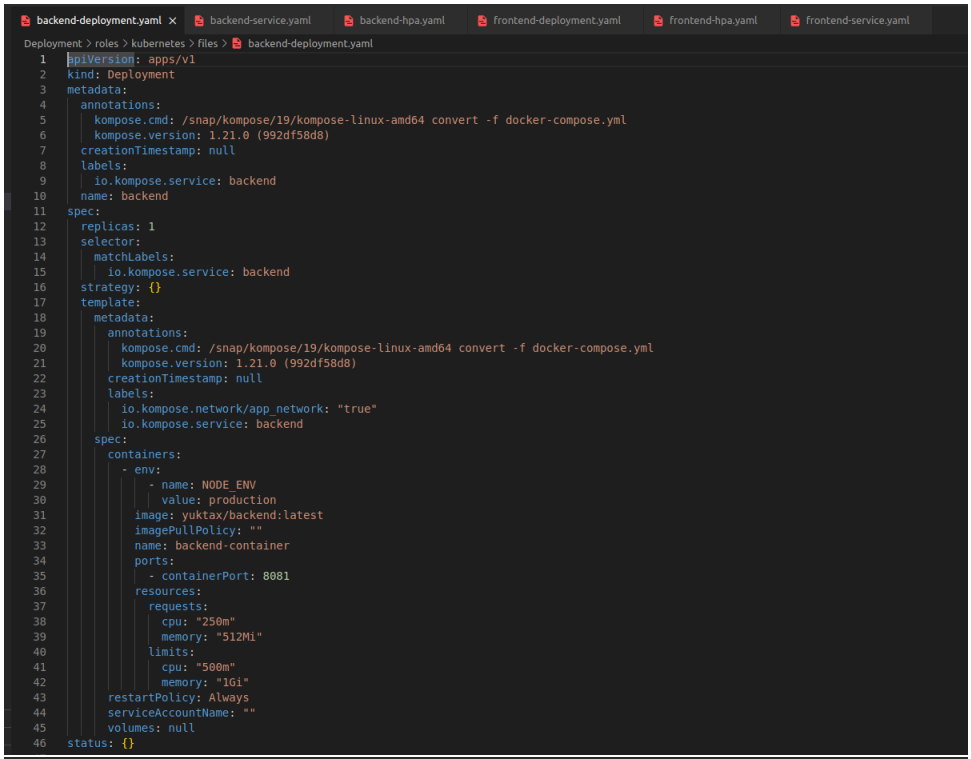
- **Containerization of Application**: Before deploying to Kubernetes, the application was containerized using Docker:
  1. **Backend**: A Spring Boot application was containerized as yuktax/backend:latest.
  2. **Frontend**: A React application was containerized as yuktax/frontend:latest. These images were built using Docker and pushed to Docker Hub to make them accessible to Kubernetes.
- **Setup**: Kubernetes was set up on the system to orchestrate the containerized application:
  1. **kubectl**: The Kubernetes command-line tool was installed and configured to interact with the cluster.
  2. **Cluster**: A local Kubernetes cluster was used (Minikube) to deploy the application.
- **Resources**: The following Kubernetes resources were created and applied to manage the application:
  1. **Deployments:** The backend-deployment.yaml and frontend-deployment.yaml files defined the desired state of the backend and frontend applications. Each deployment specified:
     (a) Container image: The Docker image for the respective service.
     (b) Replicas: The number of instances of each pod to run.
     (c) Environment variables: Configurations for inter-service communication.
     (d) Resource Requests and Limits: CPU and memory allocations to optimize resource usage.
  2. **Services:** Kubernetes services exposed the deployments to enable communication:
     (a) ClusterIP for backend: Used for internal communication within the cluster.

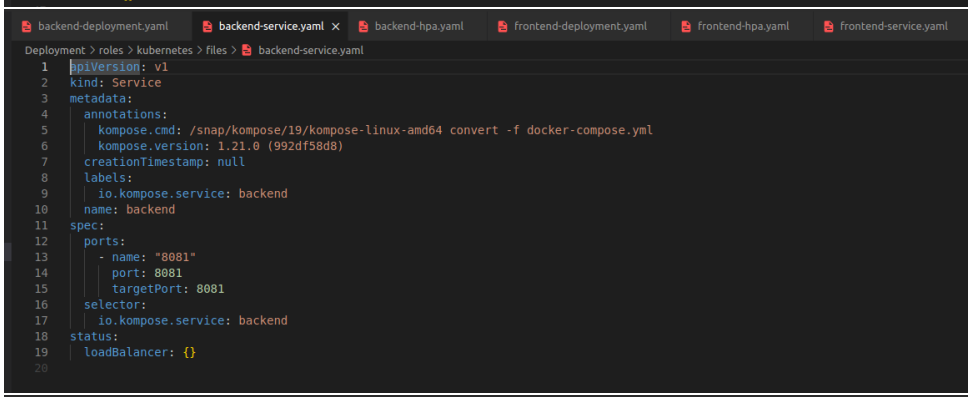
(b) NodePort for frontend: Exposed the frontend to the host system for external access.

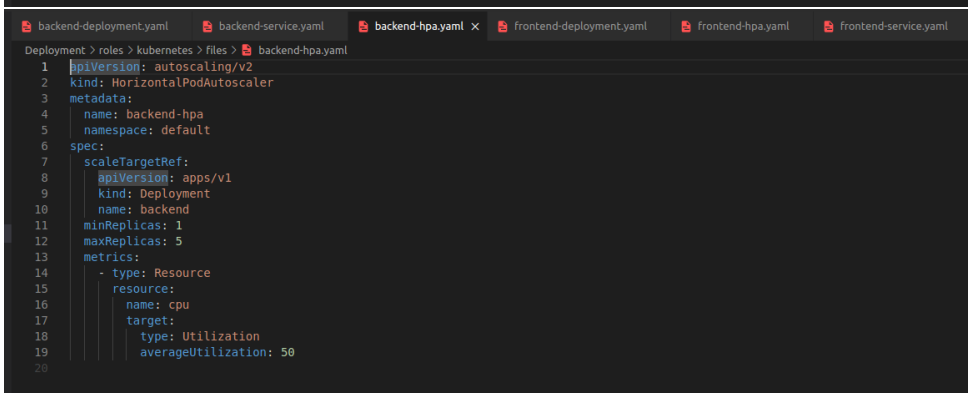
3. **Horizontal Pod Autoscaler (HPA):** HPA was implemented to scale pods dynamically based on resource usage (e.g., CPU). The frontend-hpa.yaml configured the minimum and maximum number of pods and the CPU utilization target.

The files are shown in screenshots below:

– Backend Yaml files:







– Frontend Yaml files:

```yaml
# frontend-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f docker-compose.yml
    kompose.version: 1.21.0 (992df58d8)
  creationTimestamp: null
  labels:
    io.kompose.service: frontend
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      io.kompose.service: frontend
  strategy: {}
  template:
    metadata:
      annotations:
        kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f docker-compose.yml
        kompose.version: 1.21.0 (992df58d8)
      creationTimestamp: null
      labels:
        io.kompose.network/app_network: "true"
        io.kompose.service: frontend
    spec:
      containers:
        - env:
            - name: REACT_APP_BACKEND_URL
              value: http://backend:8081
          image: yuktax/frontend:latest
          imagePullPolicy: ""
          name: frontend-container
          ports:
            - containerPort: 3000
          resources:
            requests:
              cpu: "250m"
              memory: "512Mi"
            limits:
              cpu: "500m"
              memory: "1Gi"
      restartPolicy: Always
      serviceAccountName: ""
      volumes: null
status: {}
```

```yaml
# frontend-service.yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f docker-compose.yml
    kompose.version: 1.21.0 (992df58d8)
  creationTimestamp: null
  labels:
    io.kompose.service: frontend
  name: frontend
spec:
  ports:
    - name: "3000"
      port: 3000
      targetPort: 3000
  selector:
    io.kompose.service: frontend
status:
  loadBalancer: {}
```

```yaml
# frontend-hpa.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: frontend-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: frontend
  minReplicas: 1
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

- **Deployment Process**:

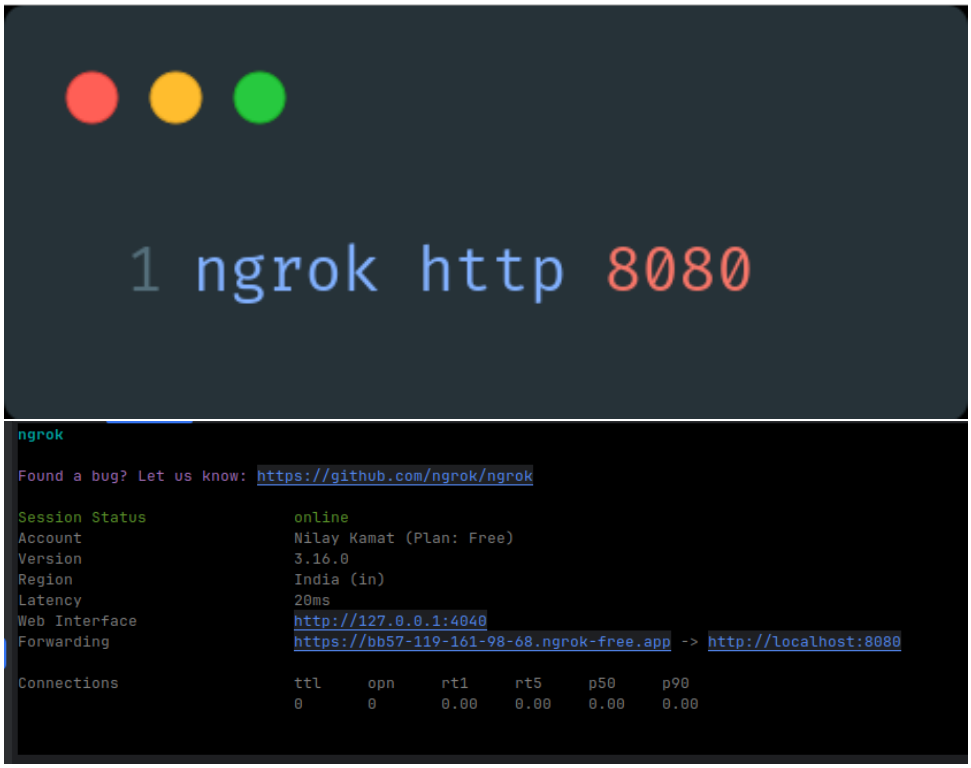
1. **Build and Push Docker Images:** Jenkins pipelines were used to build and push the Docker images to Docker Hub.

2. **Apply Kubernetes Manifests:** The kubectl apply -f command was used to deploy the resources defined in the YAML files.

3. **Monitor and Verify:** The kubectl get pods and kubectl get services commands verified that the pods and services were running as expected.

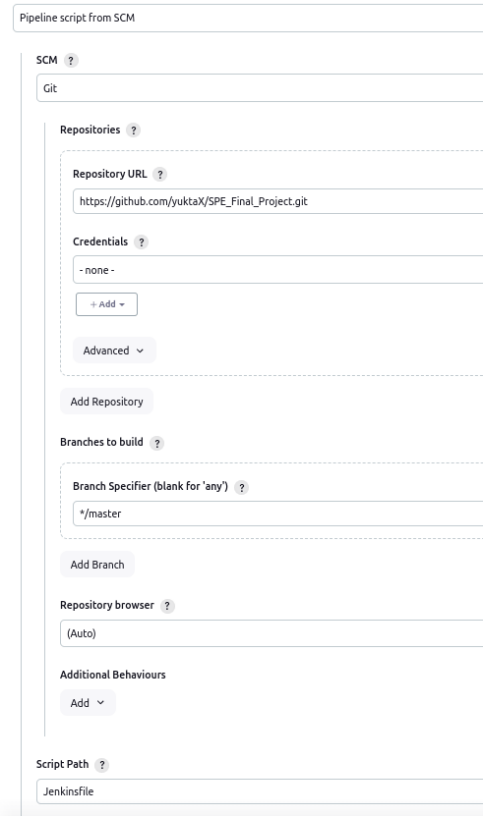
2.9 Git SCM Polling and Build Automation

- Open a terminal window and enter the below command. This command will establish an HTTP tunnel using Ngrok, exposing the local server running on port 8080 to the internet.

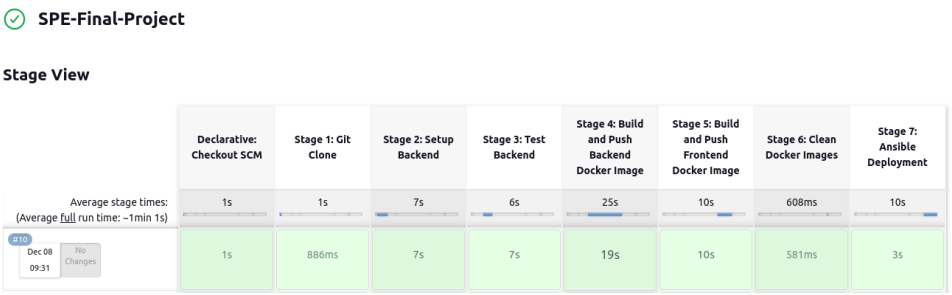


- Copy the forwarding URL provided by Ngrok. Subsequently, create a GitHub webhook and utilise this URL as the payload URL for the webhook configuration. GitHub initiates a test connection, and upon successful configuration, a '200 OK' message confirms the proper setup.

- Now, we'll update the Jenkins Pipeline script to a Jenkinsfile configure a build trigger for Git SCM polling. This setup ensures that our pipeline automatically initiates the build process whenever Jenkins detects a new commit made to the associated GitHub repository.



- Upon making any commits, the Jenkins Pipeline automatically initiates the build process and the final pipeline+build is as follows:



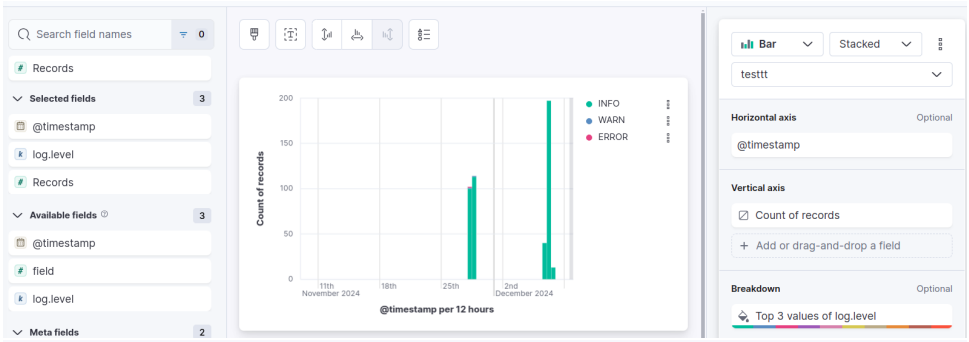
2.10 Using Kibana for Visualizing Application Logs

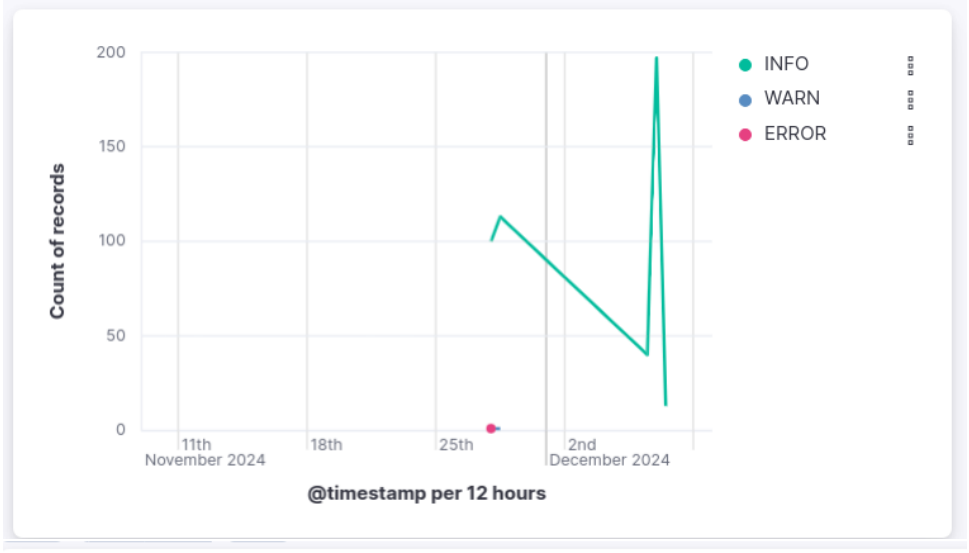
Kibana is a powerful data visualization and exploration tool that works seamlessly with Elasticsearch, enabling users to analyze and monitor application logs efficiently. By leveraging its capabilities, developers and system administrators can gain real-time insights into system performance, user activity, and potential issues.

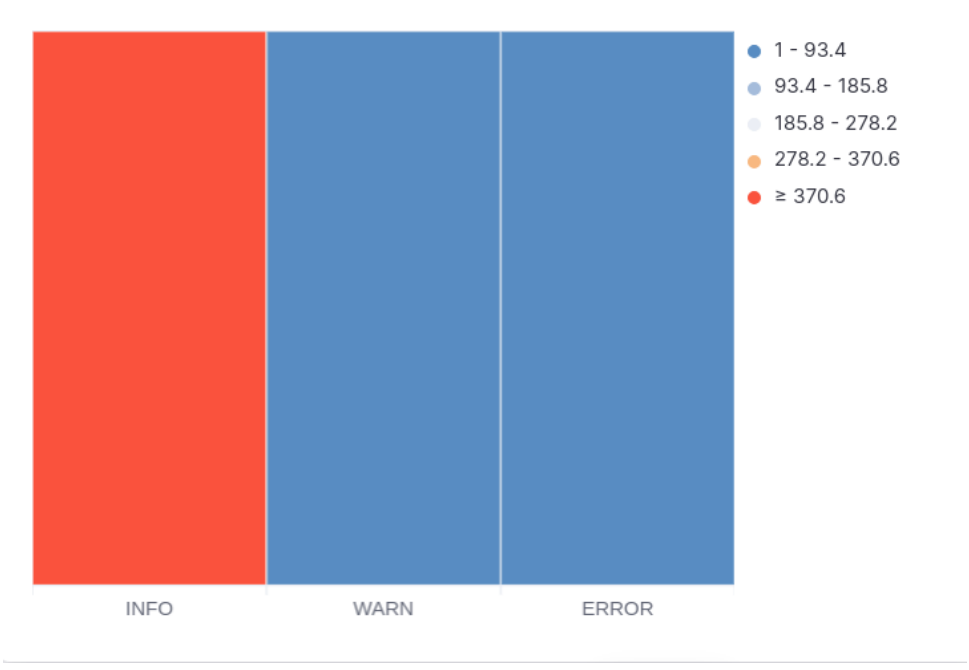
Steps for Visualizing Logs in Kibana

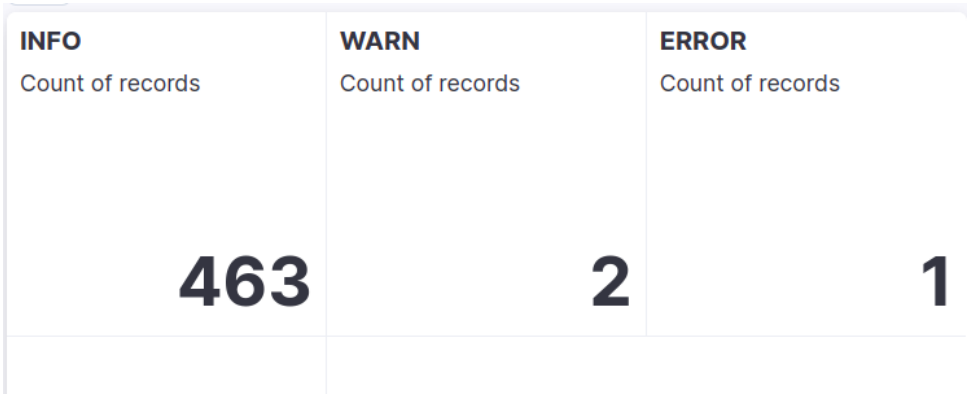
The following steps outline how to use Kibana for visualizing application logs:

1. **Log Ingestion:** Import log data into Elasticsearch using Logstash or Beats. For example, application logs in Grok-compatible format can be parsed and structured using Logstash pipelines. Here we simply uploaded the log file generated
2. **Index Creation:** Define an index pattern in Kibana that matches the imported log data. This pattern helps Kibana identify and query the logs.
3. **Building Visualizations:** Use Kibana's visualization tools to create charts such as line graphs, bar charts, pie charts, and heatmaps. For instance, logs with user activity can be visualized as time-series data to monitor peak usage hours.
4. **Dashboard Design:** Combine multiple visualizations into a cohesive dashboard for an overarching view of application behavior. For example, display logs related to error rates, active users, and WebSocket sessions on the same dashboard.


Search field names
Records
Selected fields
@timestamp
log.level
Records
Available fields
@timestamp
field
log.level
Meta fields
Bar
Stacked
testtt
Horizontal axis
Optional
@timestamp
Vertical axis
Count of records
Add or drag-and-drop a field
Breakdown
Optional
Top 3 values of log.level
INFO
WARN
ERROR
Count of records
@timestamp per 12 hours


200
150
100
50
0
Count of records
11th November 2024
18th
25th
2nd December 2024
@timestamp per 12 hours
INFO
WARN
ERROR


1 - 93.4
93.4 - 185.8
185.8 - 278.2
278.2 - 370.6
≥ 370.6
INFO
WARN
ERROR

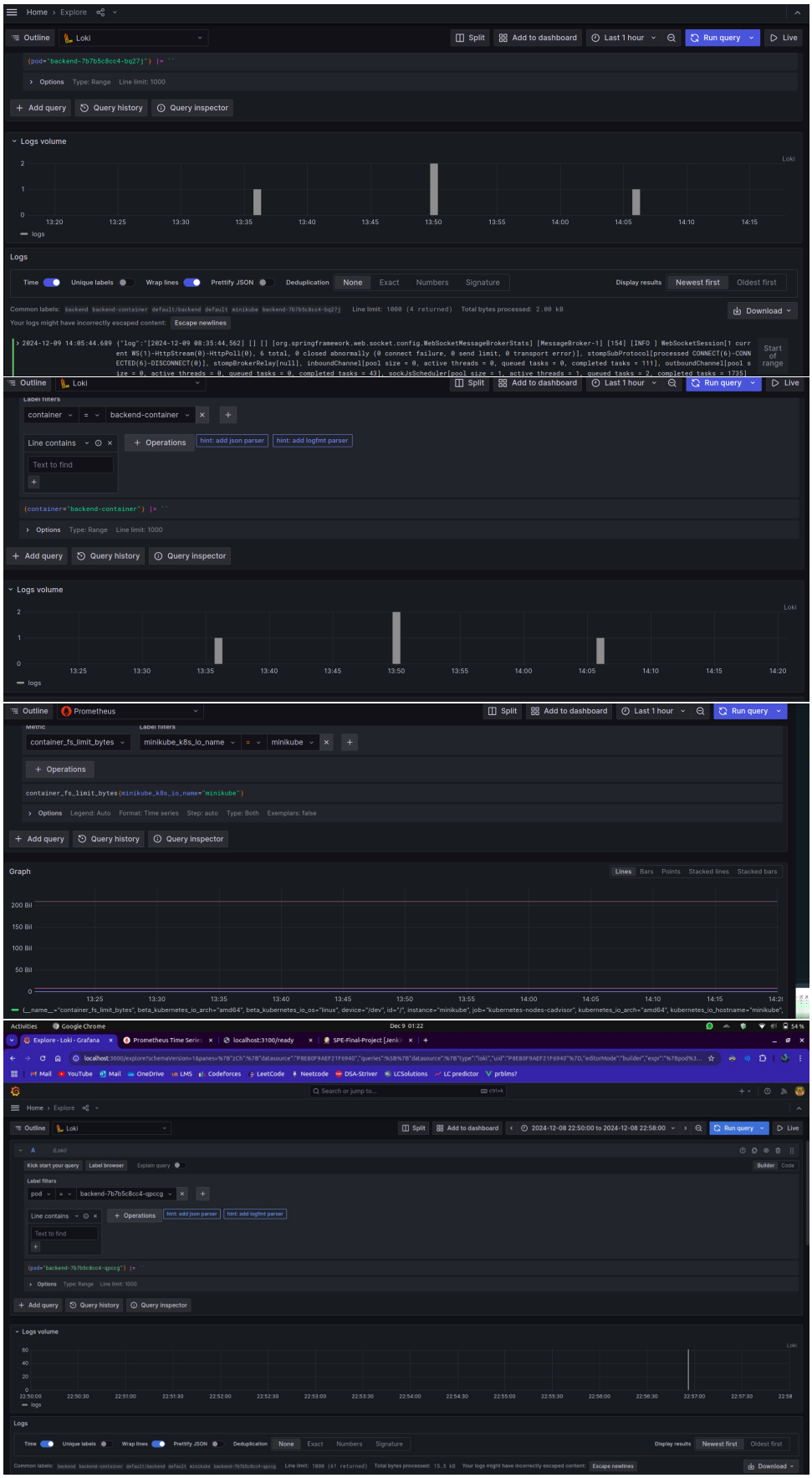


By integrating Kibana into the application's logging workflow, developers can streamline operations, ensure system reliability, and enhance user satisfaction.

2.11 Loki & Prometheus - Monitoring and Observability

To enable effective monitoring and observability of the application, we integrated **Prometheus**, **Loki**, and **Grafana** to interact with the k8s cluster where the application is running. The following steps outline the process. To find the exact commands used - Setup

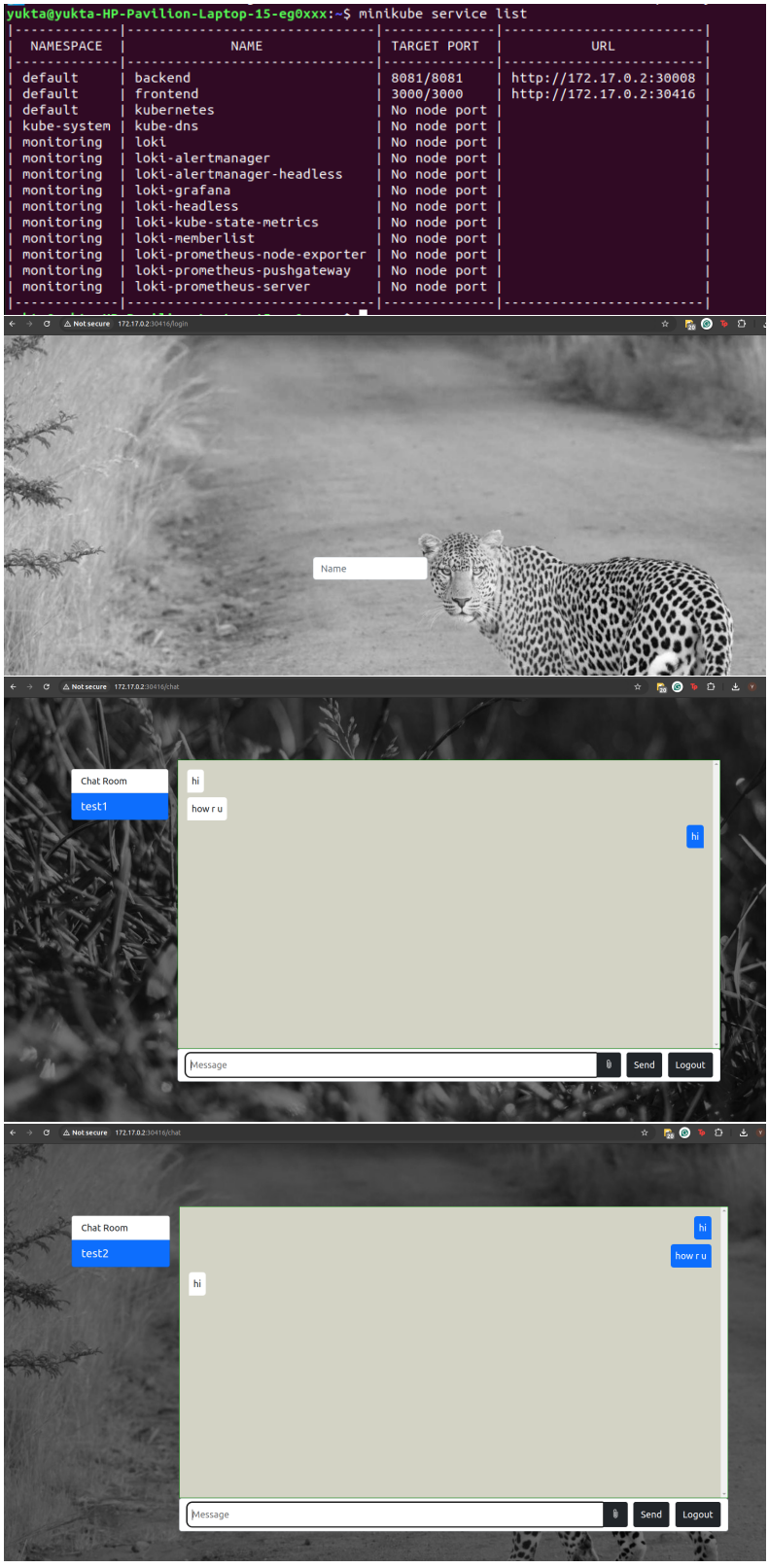
1. **Setup Prometheus:** Prometheus was deployed to scrape metrics from the backend and frontend services as well as Kubernetes resources. Custom configurations were added to monitor CPU usage, memory consumption, and request latencies.
2. **Configure Loki:** Loki was deployed for centralized log aggregation. Promtail agents were installed to forward logs from application pods and nodes to Loki, tagging them with metadata for easy querying.
3. **Integrate Grafana:** Grafana was set up with Prometheus and Loki as data sources. Pre-built dashboards were configured for application metrics, while custom dashboards were created to visualize log patterns and performance trends.
4. **Testing and Validation:** The setup was validated by generating synthetic traffic and ensuring metrics and logs appeared correctly in Grafana dashboards. Alerts were configured for critical metrics to enable proactive monitoring.



This monitoring stack provided a comprehensive observability solution for the deployed chat

application, ensuring proactive issue detection and performance optimization.

2.12 Working of Application & Links



1. GitHub repository - SPE-Final-Project Repository

2. Docker images and repository - Frontend and Backend Images