Assignment 3 – Containerizing Microservices

1. Spring Boot Application

Project Setup

Created a new Spring Boot project using the Spring Initializer : https://start.spring.io/ and included essential dependencies such as JPA and MySQL.

API Endpoints

Defined the API endpoints by creating a controller class with appropriate methods and annotations. To achieve this:
- We used the @RestController annotation to define the controller class.
- Implemented methods using annotations like @GetMapping, @PostMapping, @PutMapping, and @DeleteMapping to handle HTTP GET, POST, PUT, and DELETE requests, respectively.

Domain Model

Defined the domain model classes to represent the data that the API interacts with. Specifically, we created a model for the students table.

Service Layer

To encapsulate the business logic of the application, I developed a service class. This class is responsible for operations such as adding and deleting data from the students table.

Data Access Layer

Implemented a data access layer using JPA to interact with the database. This layer facilitates seamless integration between the application and the database.

Running the Application

After building the application, we ran it successfully. Accessing the application at localhost:8080 displayed a white error page initially since no static data was present.

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Mon Dec 02 19:50:20 EST 2024
There was an unexpected error (type=Not Found, status=404).

Endpoint Testing

To test the functionality:
- Accessed the endpoint /survey/1, which initially returned an empty array [].

Database Configuration

For database setup, we configured Amazon RDS to host the MySQL database. The connection was integrated seamlessly with the Spring Boot application.

2. Amazon RDS Setup

Steps to Create and Configure a MySQL DB Instance

Open AWS Management Console
- Navigate to the AWS Management Console and select RDS from the left navigation pane.

Select Region
- Choose the appropriate AWS Region where you want to create the database instance.

Create Database
- In the Create database section, click on Create database.

Select Engine
- Choose MySQL as the database engine.
- Select the Free Tier template.

Configure DB Instance
- Set the DB instance identifier, master username, and password.

- Choose the db.t2.micro instance class.
- Allocate 20 GB of storage with General Purpose (SSD) as the storage type.

Network Settings

- Use the default subnet group.
- Set Public accessibility to Yes.

Authentication and Backup

- Select Password authentication as the user authentication method.
- Configure the backup retention period to 1 day.

Create Database

- Click on Create database to initiate the creation of the MySQL DB instance.

Verify Setup

- Download and install a MySQL client like DBeaver.
- Use the client to connect to the database and verify table creation.

3. Spring Boot Application Database Connection

Configuring the Database Connection

Overview

- After completing the basic setup of the Spring Boot application, I configured the application to connect to the Amazon RDS instance.

Application Properties

- Used the application.properties file to define the database connection details. The configuration ensures seamless integration with the RDS instance hosted on AWS.

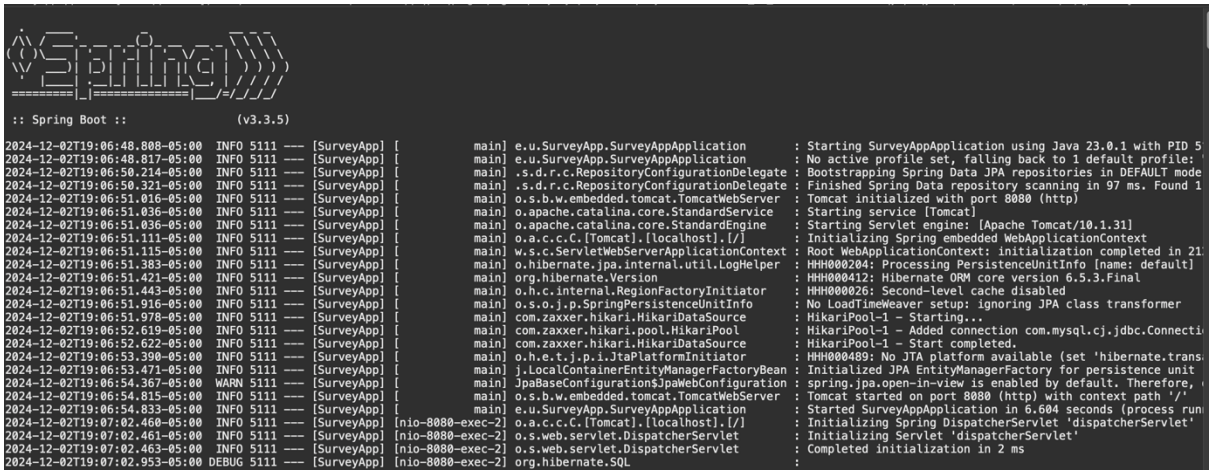
Details of the Configuration

- Database URL: The spring.datasource.url points to the Amazon RDS endpoint and includes the database name surveydb.
- Authentication: The spring.datasource.username and spring.datasource.password fields are set to the master username and password of the RDS instance.
- JPA Settings:
  - spring.jpa.hibernate.ddl-auto=update: Ensures that the schema is updated automatically.
  - spring.jpa.show-sql=true and spring.jpa.properties.hibernate.format_sql=true: Enable formatted SQL output in logs.
- Logging:

- logging.level.org.hibernate.SQL=DEBUG: Enables detailed SQL query logging.
- logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE: Provides additional information about SQL parameter binding.

Result

- This configuration established a successful connection to the Amazon RDS instance, allowing the Spring Boot application to interact with the surveydb database.

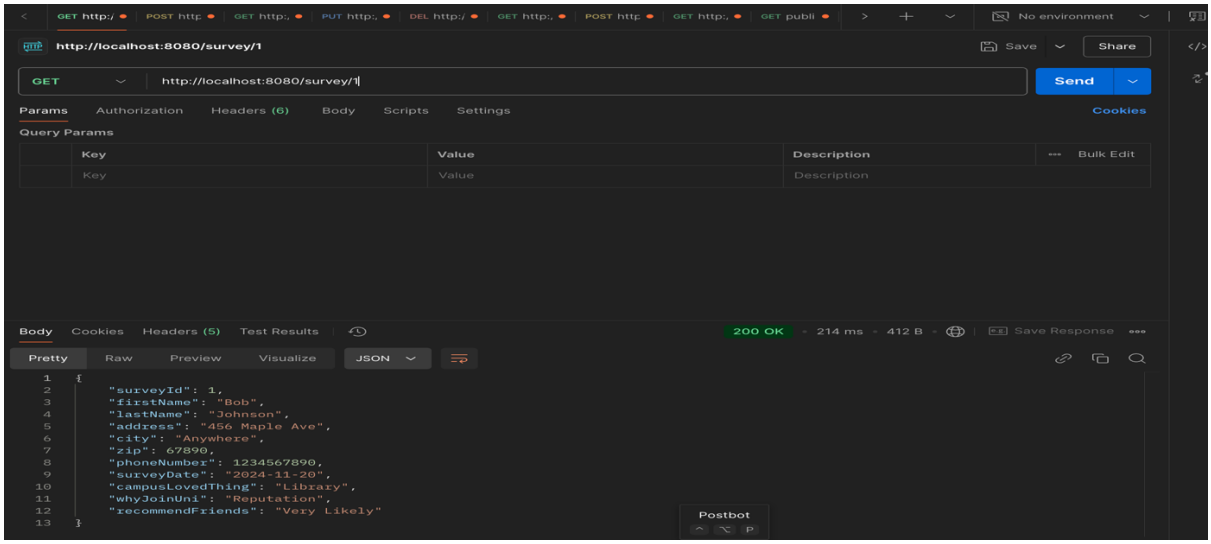


4. Verifying Database Operations on Localhost

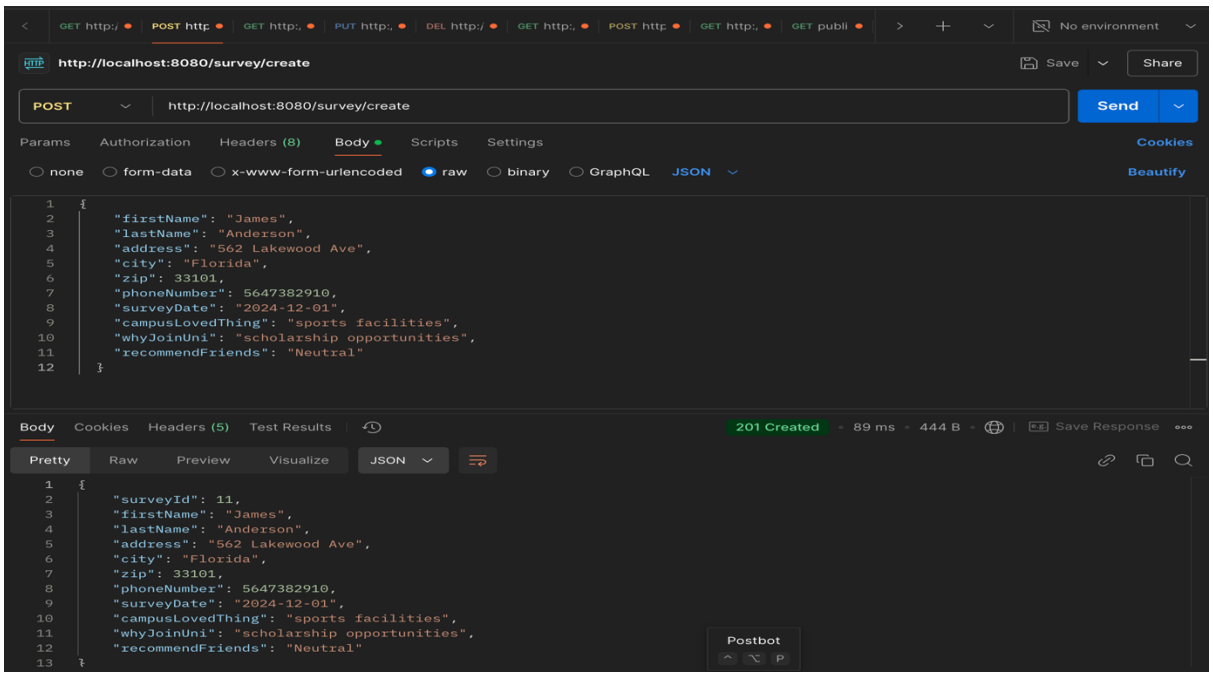
Using Postman for Validation

1. Setup Postman
- I used Postman to validate the API endpoints and ensure the connection to the database hosted on the Amazon RDS instance.
- A Postman collection was created to organize and manage the requests.

2. GET Request
- Endpoint: http://localhost:8080/survey/1
- This request fetches the survey details for the student with survey_id = 1.
- Changing the `survey_id` in the endpoint allows fetching records for other students.



3. POST Request
   - Endpoint: `https://localhost:8080/survey/create`
   - This request is used to add a new survey record.
   - The body parameter is provided in JSON format. For example:



4. DELETE Request
   - Endpoint: `https://localhost:8080/survey/{id}`
   - This request deletes the survey record associated with the given `id` in the endpoint.

Video

5. **PUT Request**

- Endpoint: https://localhost:8080/survey/{id}
- This request updates an existing survey record associated with the given id.
- The body parameter is provided in JSON format and should include the updated data.

Video

6 . Specific Record Fetching
  - Endpoint: `https://localhost:8080/survey/{id}`
  - A GET request with this endpoint fetches only the survey record for the specified `id`.

Results

- Using Postman, we verified that all endpoints successfully interact with the RDS database.
- Data can be added, fetched, and deleted seamlessly using the defined API endpoints.

5. Setting up Git

1. Create an empty repository on GitHub (https://github.com/rohini420/Assignment3.git).
2. Clone the repository to your local machine.
3. Add your project folders to the repo and use git add and git commit to stage and commit your changes.
4. Push your code to GitHub using git push.

6.Creating a Docker Image and Pushing It to DockerHub

1. Ensure that Docker is installed on your machine and create an account on https://hub.docker.com/

2. Create a Dockerfile in your code editor:
  - The Dockerfile is a text file that contains the instructions for building a Docker image. It serves as a blueprint for creating your application's container.
  - In code editor, create a new file named "Dockerfile" (without any file extension) in the same folder as your Java application's JAR file.

3. Add the following commands to the Dockerfile:

```
 Use an official OpenJDK image as a base
FROM openjdk:17-jdk-slim

 Expose the port Spring Boot is running on
EXPOSE 8080
```

Copy the built JAR file into the container
COPY target/SurveyApp-0.0.1-SNAPSHOT.jar SurveyApp-0.0.1-SNAPSHOT.jar

Command to run the application
ENTRYPOINT ["java", "-jar", "/SurveyApp-0.0.1-SNAPSHOT.jar"]

- The `FROM` command specifies the base image for your container, in this case, the official OpenJDK 17 image.
- The `EXPOSE` command declares the port (8080) that your Spring Boot application is running on.
- The `COPY` command copies the built JAR file from your local machine into the container.
- The `ENTRYPOINT` command specifies the command to run when the container starts, which is the Java command to run your application's JAR file.

Dockerfile

```
# Use an official OpenJDK image as a base
FROM openjdk:17-jdk-slim

# Expose the port Spring Boot is running on
EXPOSE 8080

# Copy the built JAR file into the container
COPY target/SurveyApp-0.0.1-SNAPSHOT.jar SurveyApp-0.0.1-SNAPSHOT.jar

# Command to run the application
ENTRYPOINT ["java", "-jar", "/SurveyApp-0.0.1-SNAPSHOT.jar"]
```

4. Build the Docker image:
- Open a command line or terminal and navigate to the folder containing your Dockerfile and JAR file.
- Run the following command to build the Docker image:

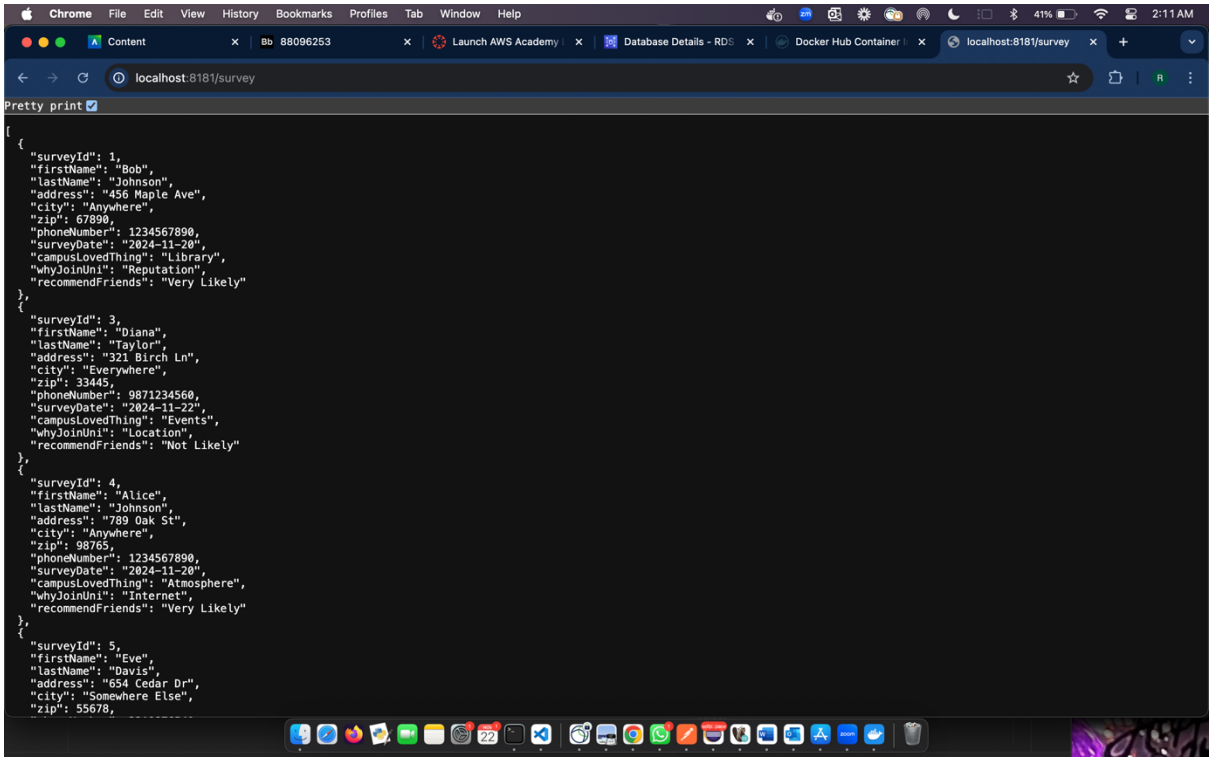
```
docker build -t surveyapp:1.0 .
```

5. Verify the Docker image:
- Run the following command to start a container based on the image you just built:

```
docker run -p 8181:8080 surveyapp:1.0
```

- This command starts a new container, maps port 8080 inside the container to port 8181 on local machine, and runs the container in interactive mode.

- Once the container is running, you should be able to access your Spring Boot application at `http://localhost:8181/survey ` in your web browser.



6. Push the Docker image to Docker Hub:
   - First, log in to Docker Hub using the following command:

     docker login -u <docker_hub_username>

   - Then, tag image with your Docker Hub username and the desired tag:

     docker tag surveyapp:1.0 <docker_hub_username>/surveyapp:1.0

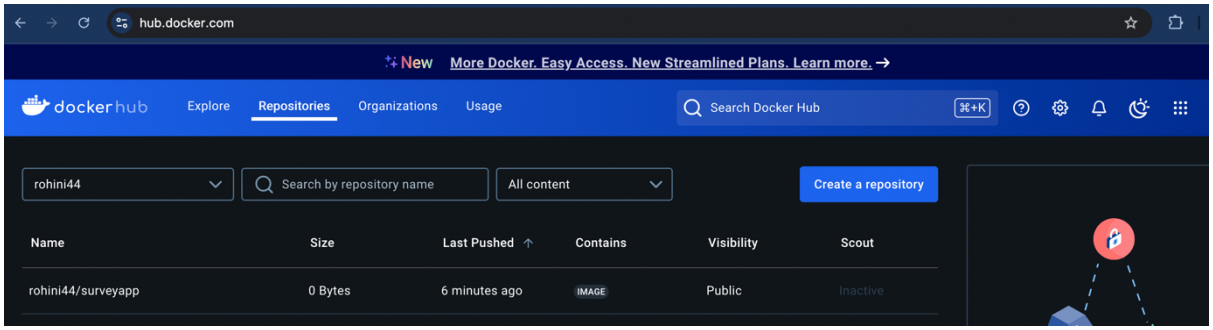
```
(base) hemanthsairambhatlapenumarthi@Hemanths-MacBook-Pro Assignment-3 % docker tag surveyapp:1.0 rohini44/surveyapp:1.0
(base) hemanthsairambhatlapenumarthi@Hemanths-MacBook-Pro Assignment-3 %
```

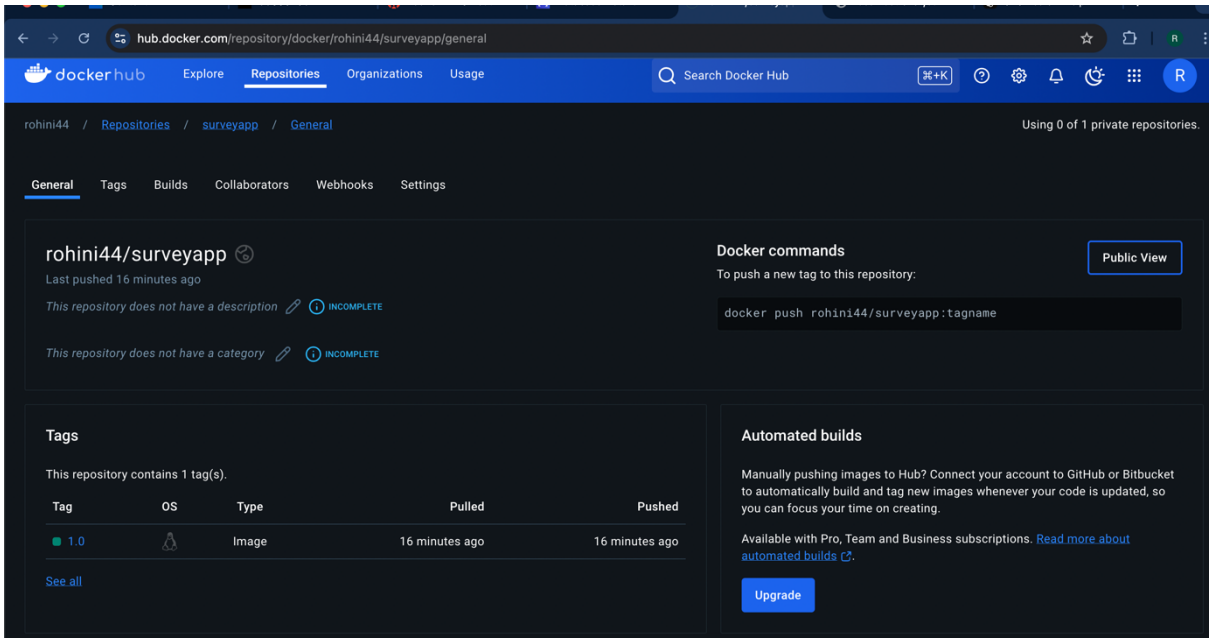
   - Finally, push the image to Docker Hub:

     docker push <docker_hub_username>/surveyapp:1.0

```
(base) hemanthsairambhatlapenumarthi@Hemanths-MacBook-Pro Assignment-3 % docker push rohini44/surveyapp:1.0
The push refers to repository [docker.io/rohini44/surveyapp]
44ee0045c1d5: Pushed
c82e5bf37b8a: Mounted from library/openjdk
2f263e87cb11: Mounted from library/openjdk
f941f90e71a8: Mounted from library/openjdk
1.0: digest: sha256:a5f910a423691001cdc10a38716d53bd2180572d601fe3660793dc4886ade9d0 size: 1165
(base) hemanthsairambhatlapenumarthi@Hemanths-MacBook-Pro Assignment-3 %
```

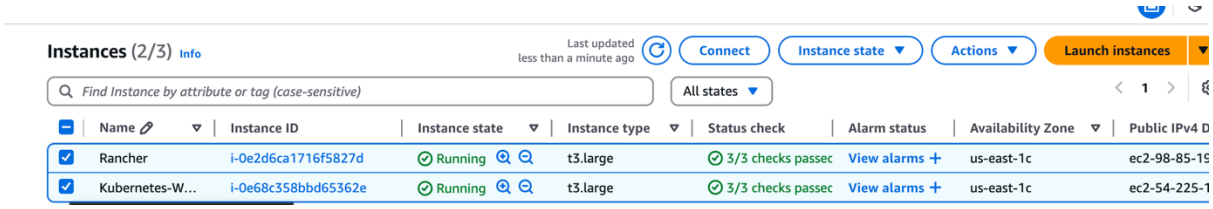
- After the push is complete, you should be able to see your image on the Docker Hub website.





7.Installing Docker and Rancher on AWS EC2 (Username for Rancher: 'admin')

1. **Launch EC2 Instances**:
    - launched 2 EC2 instances from AWS Learner Lab.



2. **Instance Configuration**:
    - Instances were created using default settings with key selections:
        - Instance type: t3.large
        - Security group: Added port ranges 80, 8080, and 443 under TCP protocol, in addition to 22.
3. **Storage Configuration**:
    - Increased storage to 1*30 GB.
4. **Elastic IPs**:
    - Created 2 Elastic IPs for dynamic cloud computing and associated them with the EC2 instances.
5. **Initial Updates**:
    - Connected to both instances and ran ***sudo apt-get update*** for system updates.
6. **Install Docker**:
    - Installed Docker using the command:

sudo apt install docker.io

```
ubuntu@ip-172-31-10-120:~$ docker --version
Docker version 24.0.7, build 24.0.7-0ubuntu2~22.04.1
```

7. **Install Rancher**:
    - Installed Rancher on the first instance using:

sudo docker run --privileged=true -d --restart=unless-stopped -p 80:80 -p 443:443 rancher/rancher

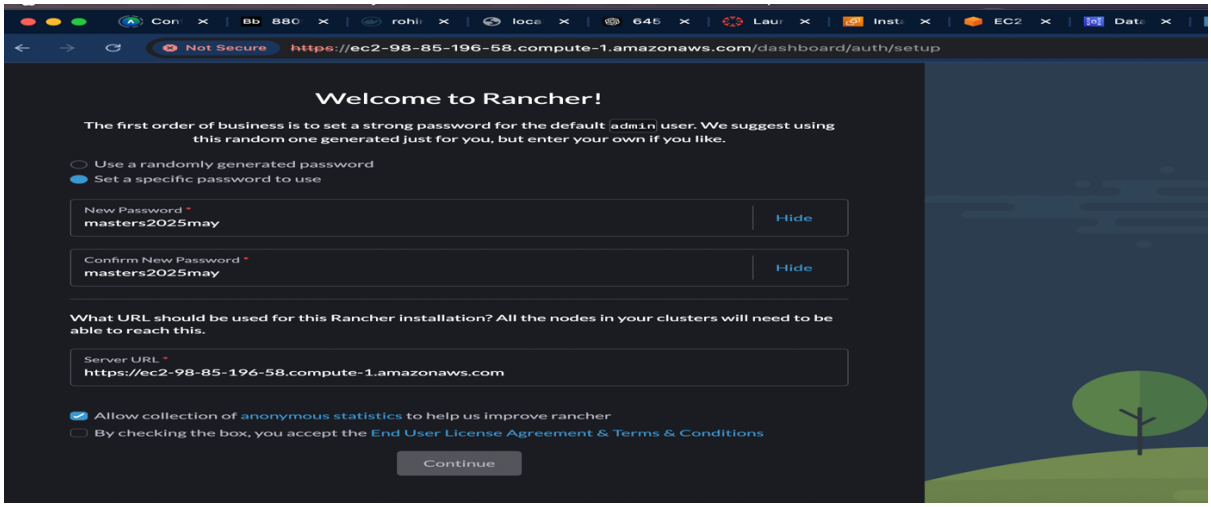
```
ubuntu@ip-172-31-10-120:~$  sudo docker run --privileged -d --restart=unless-stopped -p 80:80 -p 443:443 rancher/rancher
Unable to find image 'rancher/rancher:latest' locally
latest: Pulling from rancher/rancher
```

8. **Container ID**:
    - Retrieved the container ID with:

sudo docker ps

```
ubuntu@ip-172-31-10-120:~$ sudo docker ps
CONTAINER ID   IMAGE              COMMAND            CREATED          STATUS          PORTS                                                                                                                NAMES
c27257ad642b   rancher/rancher    "entrypoint.sh"    16 minutes ago   Up 16 minutes   0.0.0.0:80->80/tcp, :::80->80/tcp, 0.0.0.0:443->443/tcp, :::443->443/tcp                                             recursing_almeida
```

9. **Access Rancher**:
    - Open the public IPv4 DNS of the EC2 instance where Rancher is installed.



10. **Retrieve Password**:
    - Used the following command to generate a password for Rancher login:

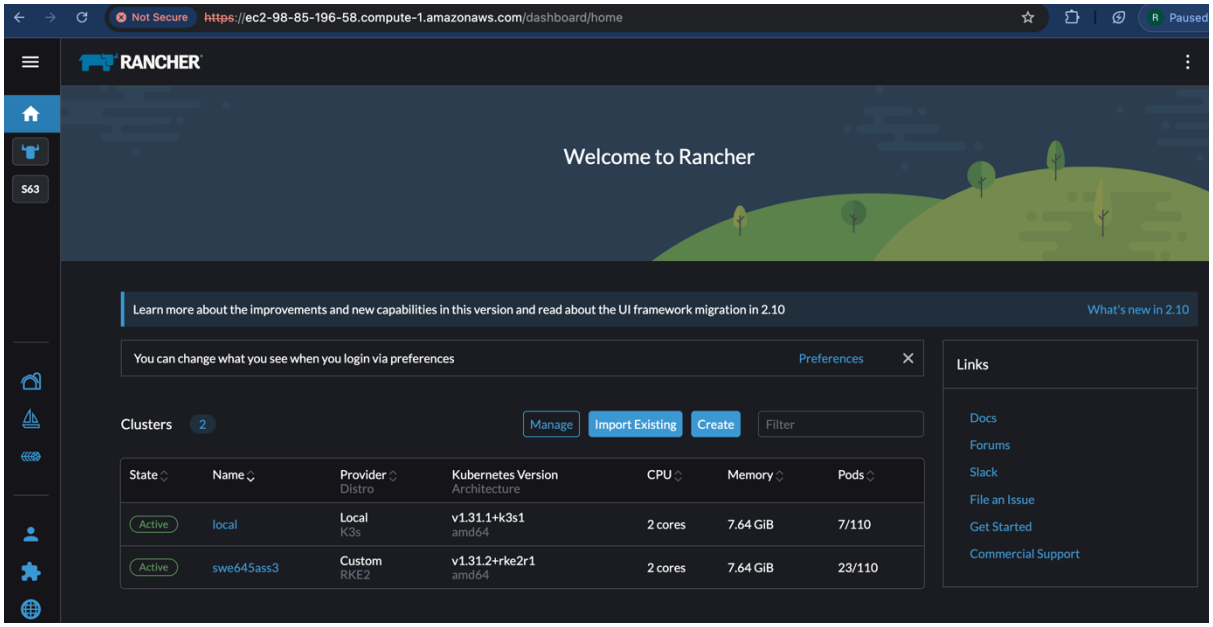
sudo docker logs <container_id> 2>&1 | grep "Bootstrap Password:"

```
ubuntu@ip-172-31-10-120:~$ sudo docker logs c27257ad642b 2>&1 | grep "Bootstrap Password:"
2024/11/23 01:34:01 [INFO] Bootstrap Password: 9x6hcgmp2cd8wmkr5fthrxxdlg4mbqrgj5ljvbtrf5v4nkrp8gztpm
ubuntu@ip-172-31-10-120:~$
```

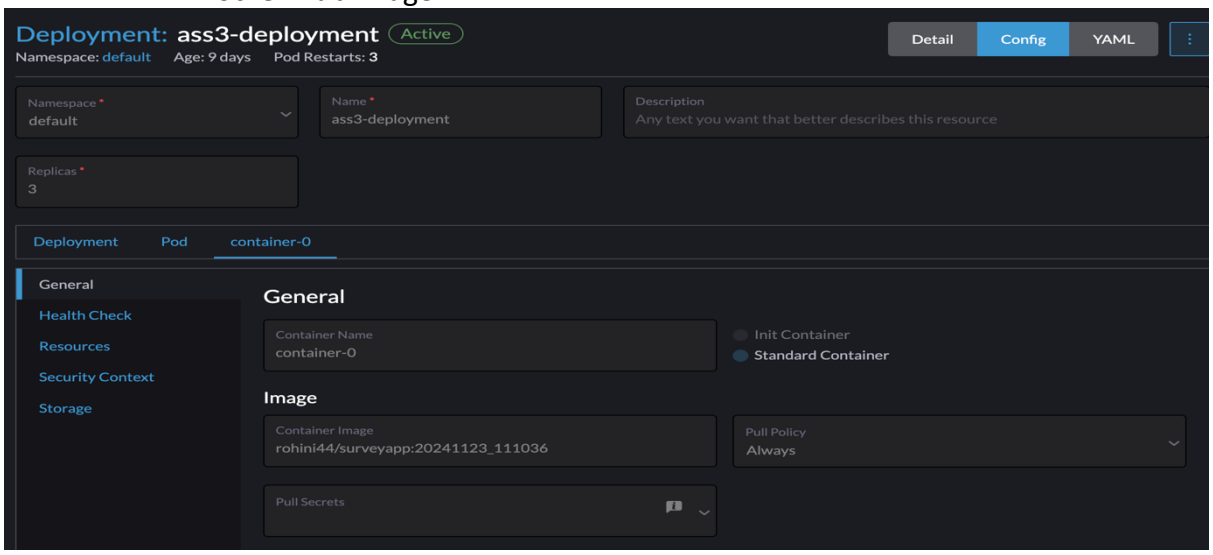
11. **Login to Rancher**:
    - Used the password to log in to the Rancher portal and changed it to a custom password.
12. **Create Cluster**:
    - Clicked "Create" in Rancher and gave the cluster a name.
13. **Configure Cluster**:
    - Verified ETCD, control panel, and worker nodes, then clicked "Next."
14. **Generate Command**:
    - A command was generated to run on the second EC2 instance as the worker node:

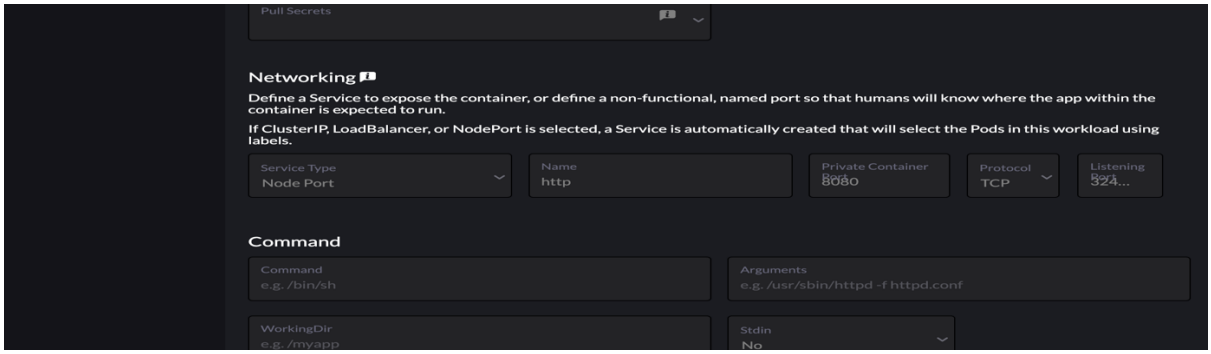
```
curl --insecure -fL https://ec2-98-85-196-58.compute-1.amazonaws.com/system-agent-install.sh | sudo  sh -s - --server https://ec2-98-85-196-58.compute-1.amazonaws.com --label 'cattle.io/os=linux' --token kss4n8jnxqzgrlzzqxgv46c54q4zcmtx227knvfg5rdtn8v9nw48wd --ca-checksum e923fef1f33bbd5915557e6ed559c4ad9d98c88b56dcc8f306c70e7476e4808d --etcd --controlplane –worker
```

15. **Confirm Worker Node**:
    - Returned to Rancher and clicked "Done."
    - Received a green signal indicating the worker node was connected.



16. **Provisioning**:
    - Waited for provisioning, then verified the cluster was active.
17. **Deploy Application**:
    - Accessed the cluster and clicked "Deployments" in the workload section.
    - Created a new deployment with 3 replicas and set it as NodePort. Used the Docker Hub image.

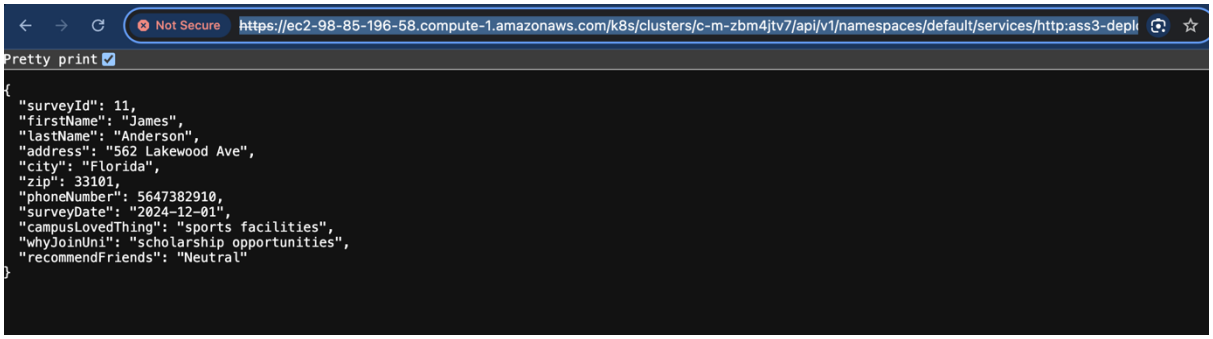




Access Application:

- Once the deployment was active, clicked the NodePort 8080 link under "Services."
- Added the GET endpoint to view the data pushed via the POST API.

https://ec2-98-85-196-58.compute-1.amazonaws.com/k8s/clusters/c-m-zbm4jtv7/api/v1/namespaces/default/services/http:ass3-deployment:8080/proxy/survey/11



8.Steps to Install and Set Up Jenkins

1. Create EC2 Instance:
 - a) Launch an EC2 instance on AWS specifically for Jenkins.
 - b) Create and associate one elastic IP with the EC2 instance for dynamic cloud computing.
 - c) Connect to the EC2 instance.

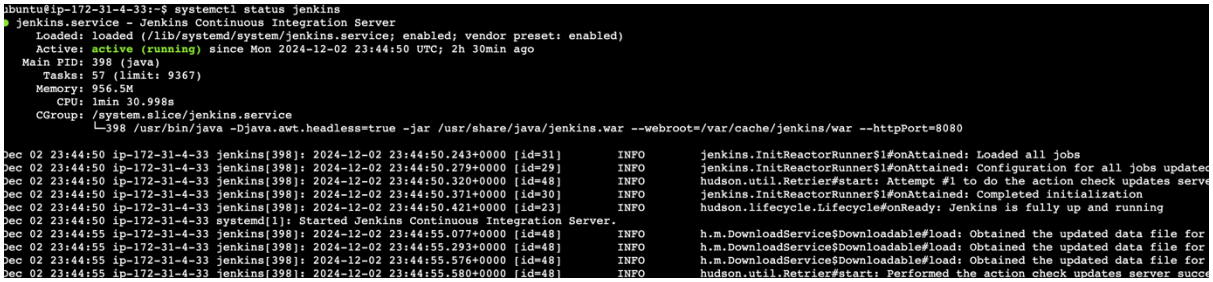
2. Install Docker and Jenkins:
 - a) Install Docker and Jenkins on the EC2 instance using the following commands:
  - Installing Docker:

    sudo apt-get update
    sudo apt install docker.io

  - Installing Jenkins:
    Followed the installation steps provided in the [Jenkins official guide](
https://pkg.jenkins.io/debian/)

3. Check Jenkins Installation:
 - Verify the status of Jenkins by running:

  systemctl status Jenkins



- Open a web browser and navigate to: ` **http://44.193.227.144:8080/`** to access Jenkins.

4. Install kubectl on the Jenkins Instance:
- Run the following commands to install `kubectl` and log in as the Jenkins user:

sudo apt install snapd
sudo snap install kubectl --classic
sudo su jenkins

5. Configure Kubernetes Access for Jenkins:
- In Rancher, click on your cluster and copy the content of the `Kubeconfig` file.
- On the EC2 console, create the `.kube` directory in the Jenkins home directory:

mkdir /home/jenkins/.kube
vi /home/jenkins/.kube/config

- Paste the copied content of the `Kubeconfig` file into the `vi` editor.

6. Verify kubectl Functionality:
- Run the following command to check if `kubectl` is properly set up:

kubectl config current-context

```
jenkins@ip-172-31-4-33:/home/ubuntu$ kubectl config current-context
swe645ass3
```

- Confirm that the output matches the name of your cluster.

9.Steps to Open Jenkins

1) Open Jenkins using **http://44.193.227.144:8080/**in your web browser.
2) If you need to retrieve the initial admin password, access the root directory on the 3rd EC2 instance and run:

- cd /var/lib/jenkins/secrets

vi initialAdminPassword
(Follow the name of the file given there).

- **3)** Copy the password and paste it into the Jenkins setup page.

- **4)** Complete the Jenkins setup by filling in the required details, such as username, password, and email.
- **5)** That's it! we are now set to follow the next steps in Jenkins.

10.Create Credentials for Git and Docker:

1) Navigate to Dashboard >> Manage Jenkins >> Manage Credentials.
2) Click on System under Stores scoped to Jenkins.
3) Click on Global credentials (unrestricted) and add your GitHub and Docker credentials here.

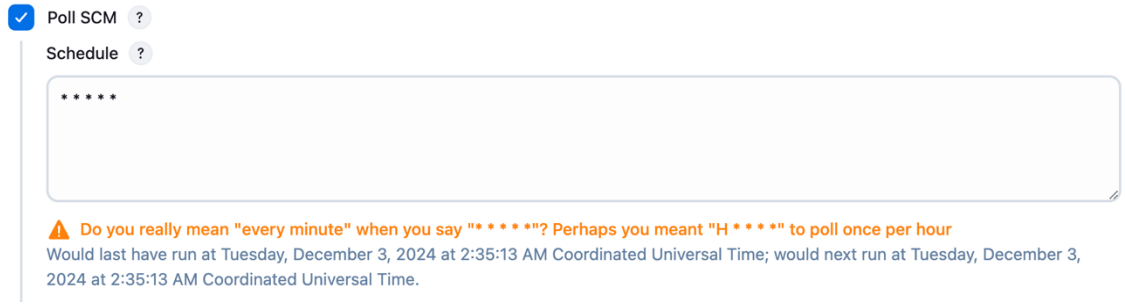
Credentials

T	P	Store ↓	Domain	ID	Name
		System	(global)	dockerhub	rohini44/******
		System	(global)	github	rohini420/******

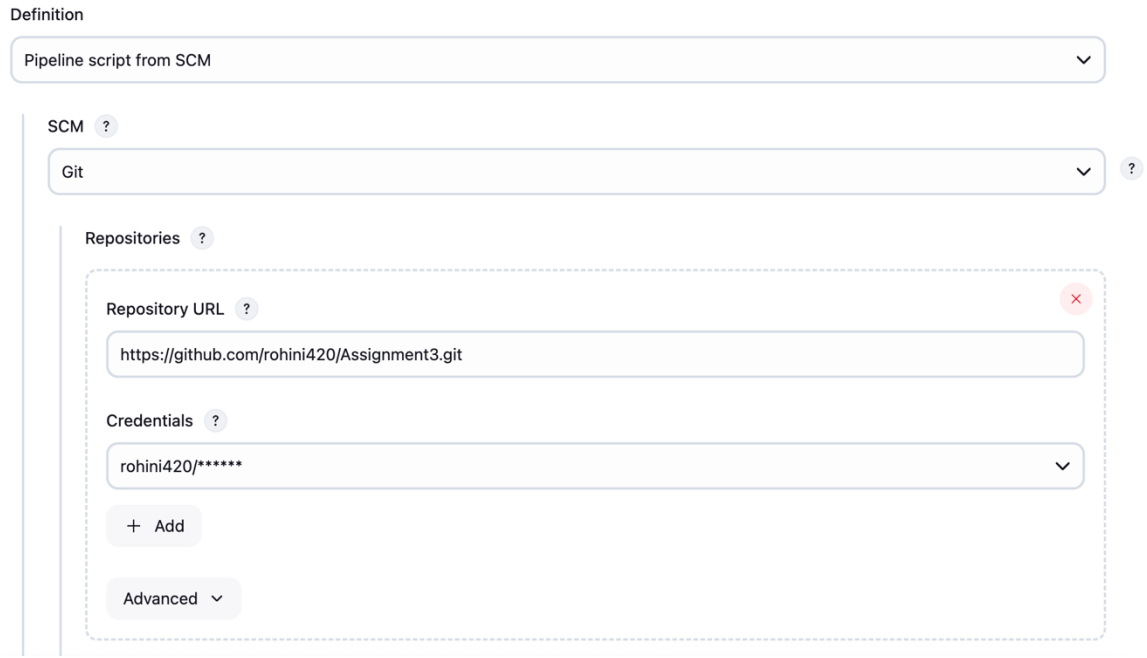
Stores scoped to Jenkins

11.Building Pipeline:

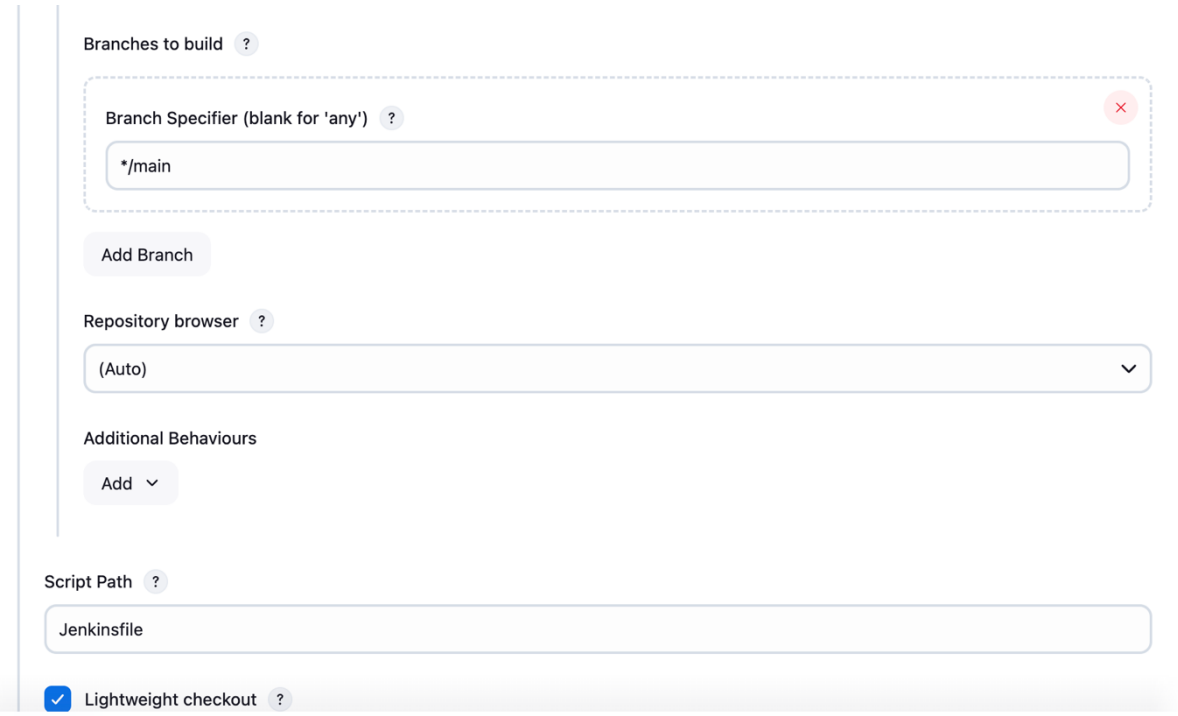
1) In the Jenkins UI, click on New Items, enter a name, and then click on Pipeline.
2) Since we need Jenkins to check for every change, set up a periodic job that runs every minute as a build trigger.



3) Connect to your Git repository by providing the Git URL and credentials as cloud credentials.



4) Specify the location of the Jenkinsfile. In this case, place the Jenkinsfile in the root directory of the Git repository.



5) Leave all other settings as default and save.

12.Required Plugins and Writing a Jenkinsfile

1. Install Necessary Plugins in Jenkins:
    - Go to Manage Jenkins and click on Manage Plugins.
    - Install the following plugins:
        - GitHub plugin

- Docker plugin
- Safe Restart plugin
- Maven Integration Plugin
- Pipeline Maven Integration Plugin

2. Configure Environmental Variables:
    - The DockerHub password should be passed as an environmental variable to the Jenkinsfile.
    - Configure this variable under Manage Jenkins > Configure System.
3. Create a Jenkinsfile:
    - In Integrated Development Environment (IDE), create a Jenkinsfile in the same folder where your Dockerfile is located. This file will define the pipeline steps for Jenkins.

Code Blame 53 lines (51 loc) · 1.27 KB Code 55% faster with GitHub Copilot

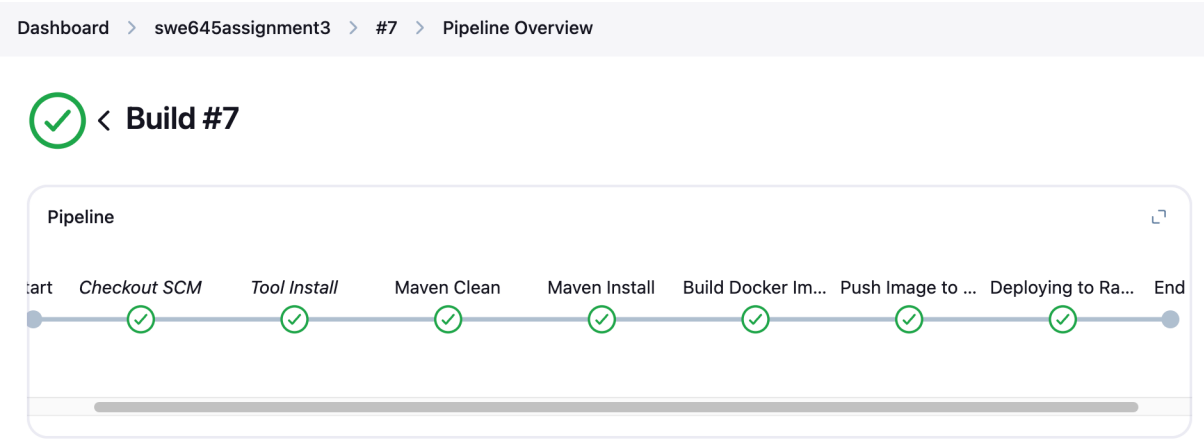
```
pipeline {
   environment {
        registryCredential = 'dockerhub'
        TIMESTAMP = new Date().format("yyyyMMdd_HHmmss")
    }
   agent any
   tools {
    maven 'Maven 3.9.9'
}

   stages {
    stage('Maven Clean') {
            steps {
               script{
                sh 'mvn clean'
               }
            }
        }
        stage('Maven Install') {
            steps {
               script{
                sh 'mvn install -DskipTests'
            }
            }
        }
      stage('Build Docker Image') {
         steps {
            script{
               docker.withRegistry('',registryCredential){
                  def customImage = docker.build("rohini44/surveyapp:${env.TIMESTAMP}")
               }
            }
         }
      }

      stage('Push Image to Dockerhub') {
         steps {
            script{
               docker.withRegistry('',registryCredential){
                  sh "docker push rohini44/surveyapp:${env.TIMESTAMP}"
               }
            }
         }
      }
      stage('Deploying to Rancher to single node(deployed in 3 replicas)') {
         steps {
            script{
               sh "kubectl set image deployment/ass3-deployment container-0=rohini44/surveyapp:${env.TIMESTAMP} -n default"
            }
         }
      }
   }
}
```

13.Running the Pipeline

1. **Push Changes to GitHub**:
    - Make the necessary changes to your code and push them to your GitHub repository.
2. **Pipeline Trigger**:
    - The Jenkins pipeline will automatically trigger upon detecting the code push. This will:
        - Build the .jar file from the source code.
        - Create a Docker image and push it to Docker Hub.
        - Deploy the updated application to the Rancher-managed Kubernetes cluster.
    - The pipeline ensures that the changes are reflected on the container running in the Kubernetes cluster.
3. **Verify Deployment**:
    - Access the deployed service using the following link: https://ec2-98-85-196-58.compute-1.amazonaws.com/k8s/clusters/c-m-zbm4jtv7/api/v1/namespaces/default/services/http:ass3-deployment:8080/proxy/survey/11

    - **Note**: It may take some time for changes to be reflected in the Kubernetes cluster. Refresh the link if necessary to see the updates. This process was demonstrated in our demo as well.

14.Pipeline overview:



URL/Links of the application deployed:

1. GitHub URL : https://github.com/rohini420/Assignment3.git
2. Docker hub URL : https://hub.docker.com/repository/docker/rohini44/surveyapp/general and then go to latest tag

3. Application deployed on cluster URL : https://ec2-98-85-196-58.compute-1.amazonaws.com/k8s/clusters/c-m-zbm4jtv7/api/v1/namespaces/default/services/http:ass3-deployment:8080/proxy/survey/11

Contributions:

Venkata Likhith Kodali (G01406932):

Played a vital role in setting up the foundational infrastructure for the project. Configured the Jenkins pipeline to integrate with the Git repository, ensuring that code could be automatically pulled and built. Assisted in troubleshooting initial issues with the Docker setup and verified that Docker images could be built and deployed correctly within the pipeline.

Joseph Adarsh Kancherla (G01459545):

Enhanced the automation aspects of the pipeline by integrating Maven build steps and configuring the Jenkinsfile to handle multi-stage builds. Ensured that the .jar file was properly generated, tested, and packaged. Assisted in setting up environmental variables in Jenkins for secure handling of DockerHub credentials.

Rohini Swathi Bhatlapenumarthi (G01452093):

Spearheaded the configuration of the Jenkins pipeline, including integrating Docker and setting up necessary plugins such as GitHub, Maven, and Docker plugins. Played a key role in writing and refining the Jenkinsfile to automate the build, test, and deployment stages seamlessly. Worked extensively on setting up credentials management for Git and DockerHub, ensuring secure and effective use within Jenkins. Demonstrated expertise in Kubernetes by configuring the deployment steps, managing the Kubeconfig setup, and verifying successful container deployments. Proactive troubleshooting and issue resolution contributed to the pipeline's efficiency, leading to successful deployments to Rancher.

Contributed significantly to the development of the project by leveraging Spring Boot for building and deploying the backend application. This included creating the core REST APIs, configuring essential security features, and ensuring that the service layer and data access layer were seamlessly integrated with the database. The efficient use of Spring Boot’s annotation-driven architecture enabled the creation of a robust and scalable application structure.

In addition, took responsibility for thorough API testing and validation using Postman. This involved designing comprehensive test suites to verify the functionality of various endpoints, checking request and response formats, and ensuring that all APIs returned the expected status codes and payloads. Postman’s automation features were utilized to run

these test suites, allowing for consistent and rapid feedback on the application's health and functionality. This rigorous testing process helped identify and resolve potential issues early, ensuring a high-quality, bug-free deployment.