Lab 06 - Jenkins and Nginx

In this lab, you will learn how modern applications are deployed using automation, containers, and load balancing.

You will use **Jenkins** to automate the deployment of multiple backend services running in Docker containers, and NGINX to distribute incoming traffic across these services using load balancing.

This lab focuses on conceptual understanding, not complex application development.

What is Load Balancing?

Load balancing is a technique used to distribute incoming network traffic across multiple backend servers. This improves:

- Performance
- Reliability
- Fault tolerance

In this lab, **NGINX** acts as a load balancer and forwards requests to multiple backend containers.

Overview of the Lab:

You will perform the following tasks:

1. Set up Jenkins using Docker
2. Create a backend application that identifies itself
3. Dockerize the backend application
4. Configure NGINX for load balancing
5. Use Jenkins to deploy multiple backend containers
6. Verify load balancing through a web browser

Before we begin, create a PUBLIC github repo named CC_Lab-6

Task 1: Set Up Jenkins Using Docker

Aim

To run Jenkins as a Docker container.

Deliverables

This is what you should have by the end of this task.

Screenshot of Jenkins dashboard running in the browser **(SS1)**

```
[LF]>
[LF]> ************************************************************
[LF]> ************************************************************
[LF]> ************************************************************
[LF]>
[LF]> Jenkins initial setup is required. An admin user has been created and a password generated.
[LF]> Please use the following password to proceed to installation:
[LF]>
[LF]> a85c3fc683ee46278c1b17b491d5a0a4
[LF]>
[LF]> This may also be found at: /var/jenkins_home/secrets/initialAdminPassword
[LF]>
[LF]> ************************************************************
[LF]> ************************************************************
[LF]> ************************************************************
```

- **Use the code:** `docker logs jenkins` **once docker is installed to get this SS.**

Steps

1. Pull the Jenkins image:

```
docker pull jenkins/jenkins:lts
```

2. Build a custom jenkins image, use the file Dockerfile.jenkins provided to you

   From inside the folder run:

   docker build -t jenkins-docker -f Dockerfile.jenkins .

   (This will take a while)

```
+] Building 98.8s (6/6) FINISHED                                                                    docker:desktop-linux
=> [internal] load build definition from Dockerfile.jenkins                                                        0.0s
=> => transferring dockerfile: 178B                                                                                0.0s
=> [internal] load metadata for docker.io/jenkins/jenkins:lts                                                      0.1s
=> [internal] load .dockerignore                                                                                   0.0s
=> => transferring context: 2B                                                                                     0.0s
=> [1/2] FROM docker.io/jenkins/jenkins:lts@sha256:d1ea795c6facd7f549a21c40e5e43ffcc5fbc5f48683d9b24750f26e8079d   1.8s
=> => resolve docker.io/jenkins/jenkins:lts@sha256:d1ea795c6facd7f549a21c40e5e43ffcc5fbc5f48683d9b24750f26e8079d   1.6s
=> [2/2] RUN apt-get update && apt-get install -y     docker.io     make     g++     curl                       44.6s
=> exporting to image                                                                                             52.0s
=> => exporting layers                                                                                            36.4s
=> => exporting manifest sha256:b27db89f15f01b8b79f4a3c3db6dab9c31e70ceae34183c721c45a717559701d                   0.0s
=> => exporting config sha256:d0b10a5e6aa8aa3b955d87af7c6e214a3cfc97f6f8efe088ee6a9e2ddbe8633b                     0.0s
=> => exporting attestation manifest sha256:89ebb2aa7383ddcc6ef7d32f306329eeb6dafb01bc1630d4a3265836d72cc73e       0.0s
=> => exporting manifest list sha256:b6039ad4387c8ce63b3a1e474527dd5bae9251423b28f059b7751361b415fde3              0.0s
=> => naming to docker.io/library/jenkins-docker:latest                                                           0.0s
=> => unpacking to docker.io/library/jenkins-docker:latest                                                       15.4s

iew build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/oqverh049be01izamr4lo1zz9
```

3. Run Jenkins container:

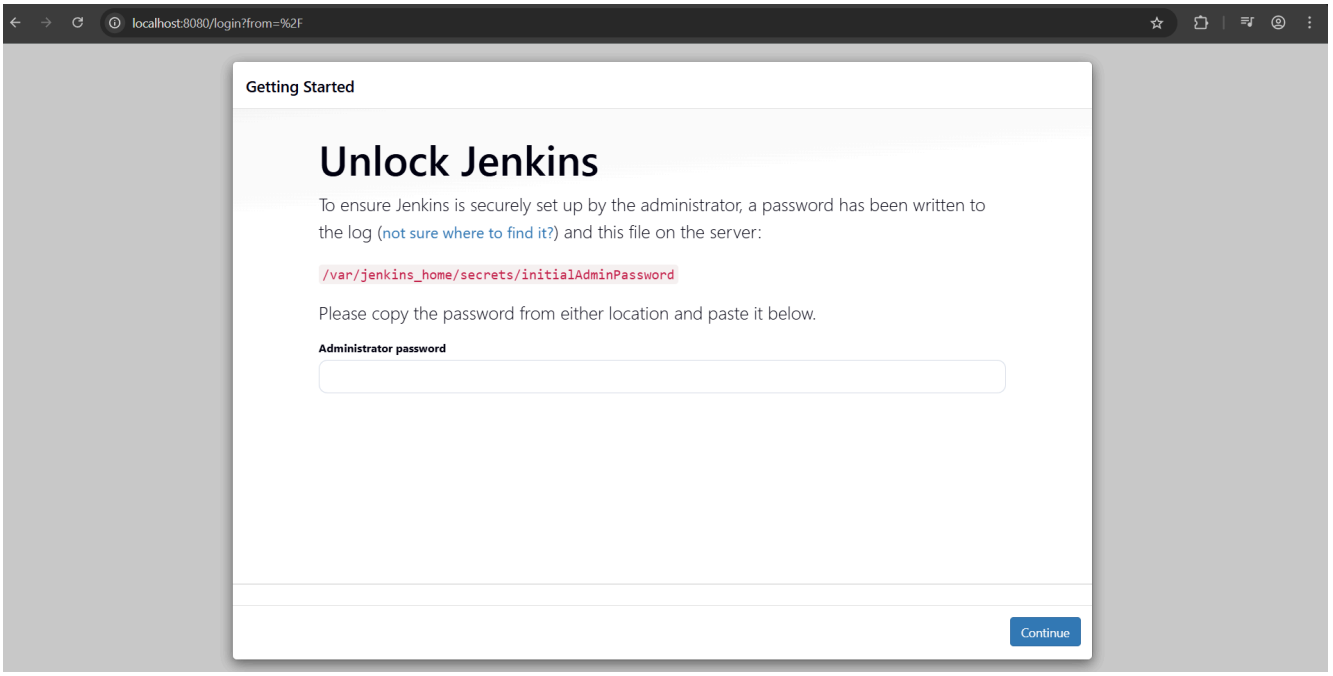
docker run -d -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home -v /var/run/docker.sock:/var/run/docker.sock --name jenkins jenkins-docker

This will start Jenkins using the custom image with Docker support and persistent storage.

The Jenkins container is run with a Docker volume to persist all Jenkins data across restarts.

3. Next, open your browser and navigate to:

http://localhost:8080



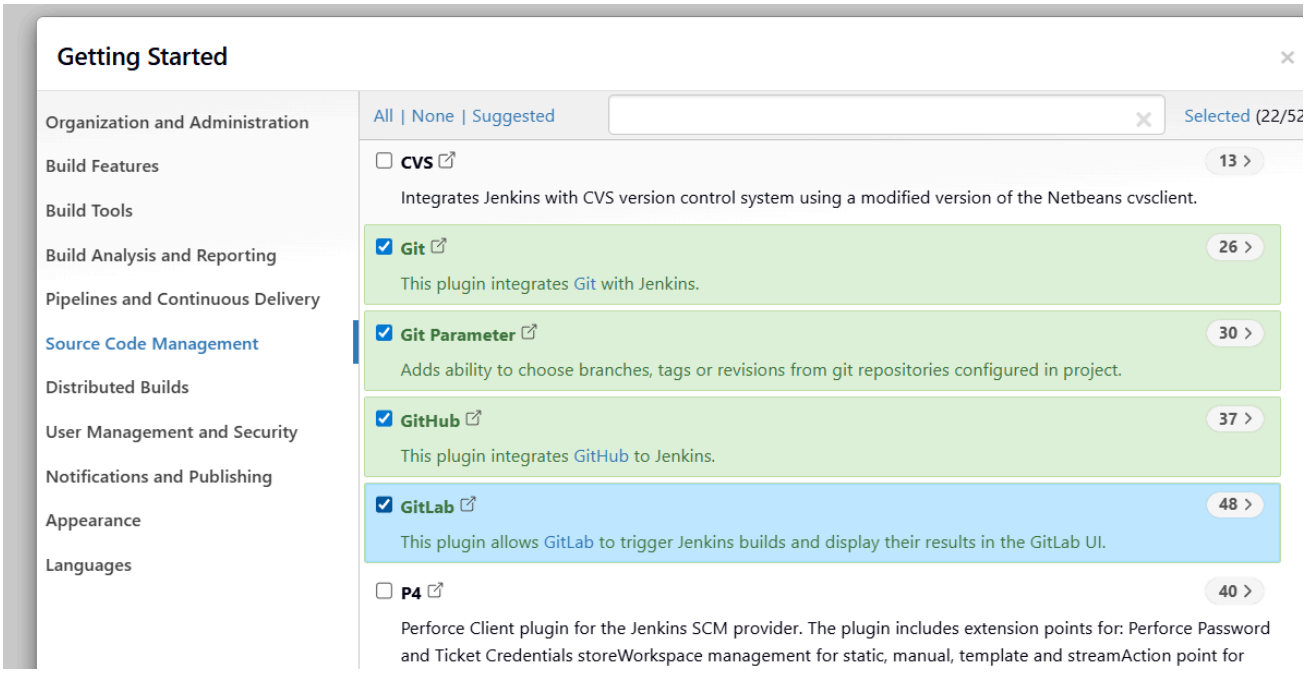
4. Retrieve the initial admin password:

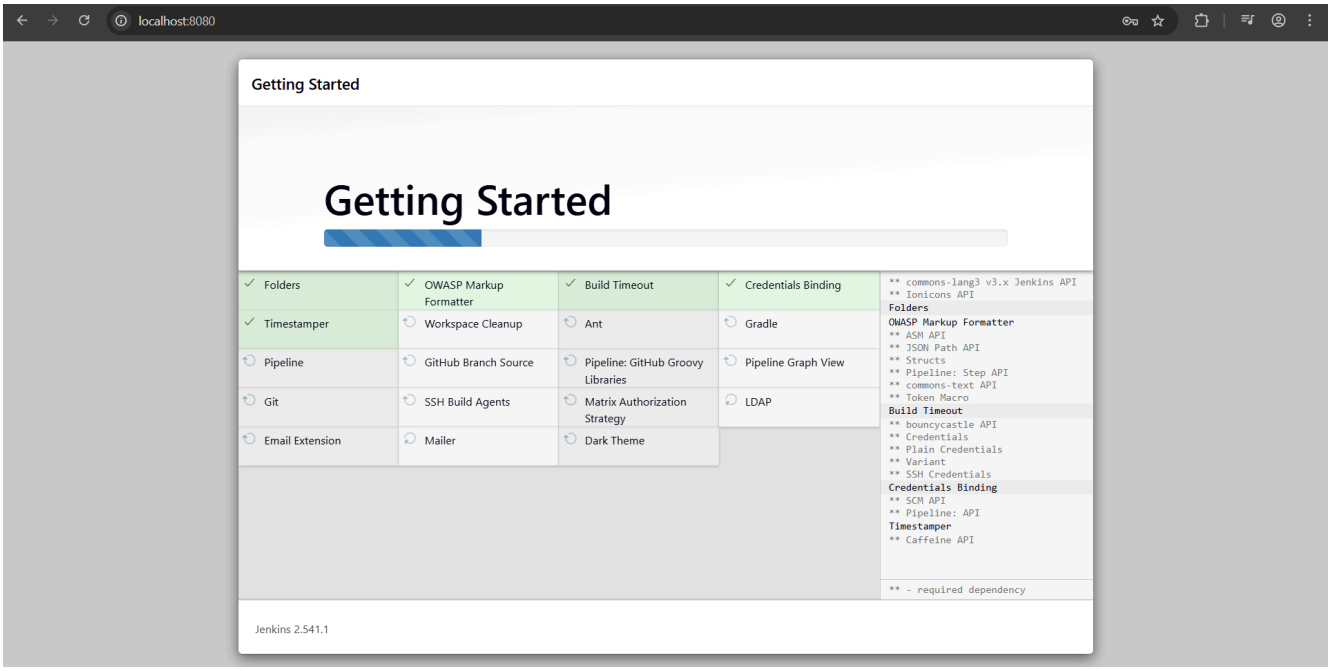
```
docker exec jenkins cat /var/jenkins_home/secrets/initialAdminPassword
```

docker exec jenkins cat /var/jenkins_home/secrets/initialAdminPassword

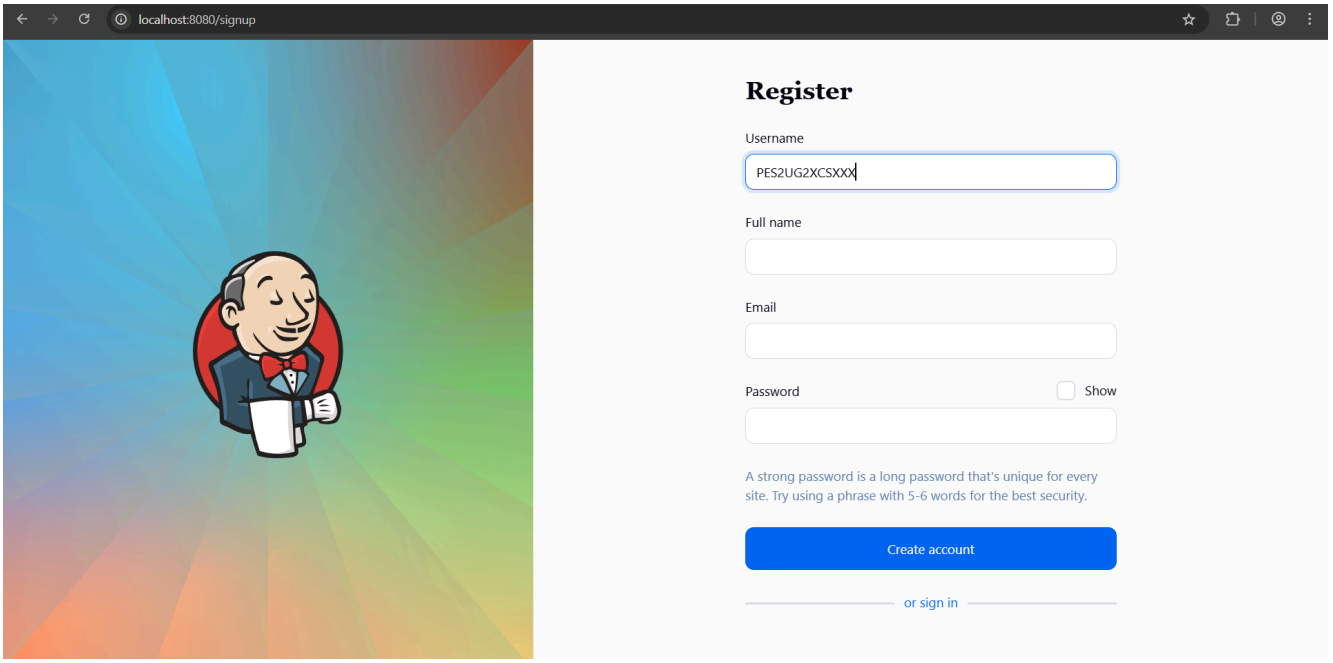
5. Install Suggested Plugins and complete the Jenkins setup.

   When prompted for plugin installation, click on “Select Plugins to Install” and then search for **GitHub** and check the **GitHub** options. (This step may take a few minutes to complete)

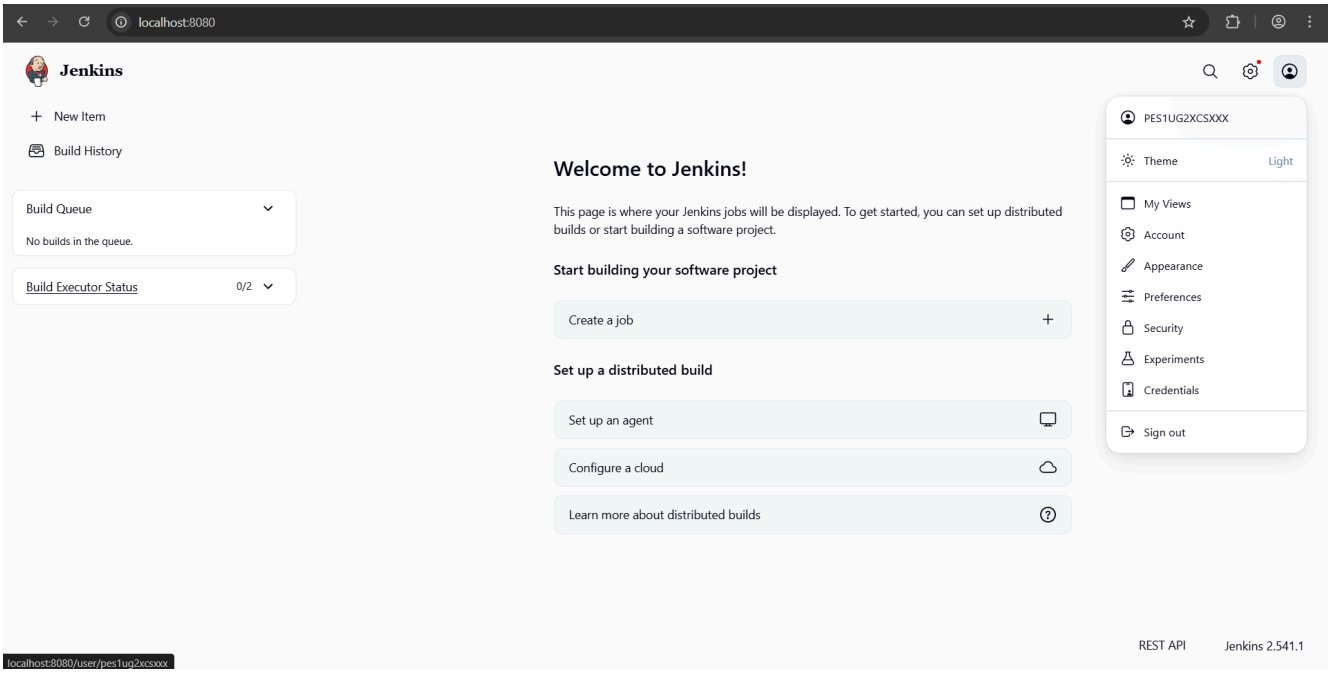




Sign in with your **SRN**, when you are creating the first admin account, otherwise if you missed this step, go to **Settings>Security>Security Realm>Allow Users** to sign up.



Screenshot 2 This would be your Jenkins **dashboard page** with your **SRN**.



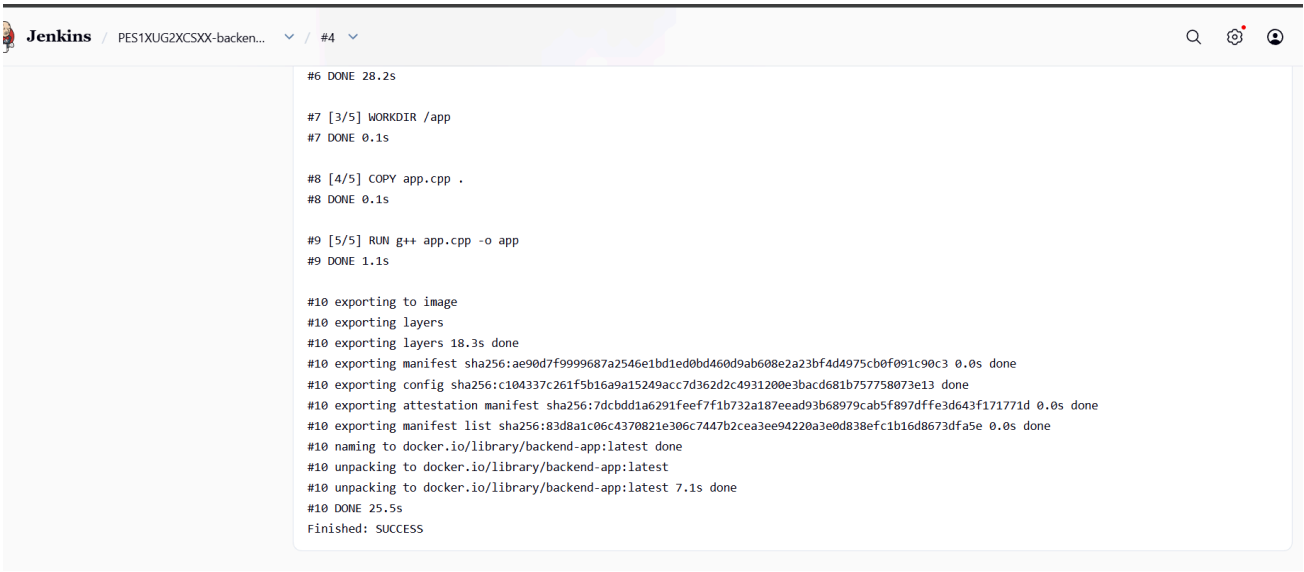
Task-2: Create a Jenkins Job to Build the Backend

Aim

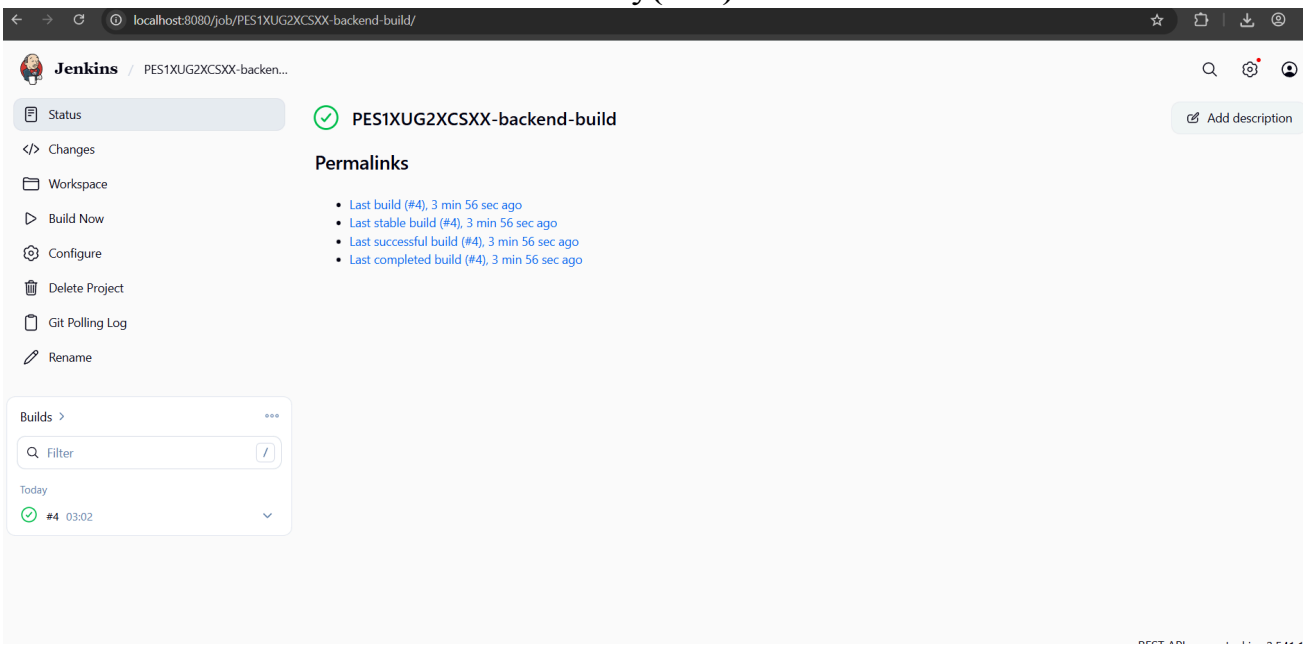
To configure Jenkins to automatically build and test a backend application from a GitHub repository.

Deliverables

- Screenshot of Jenkins **Console Output(SS3)**



- Screenshot of **Stable** build in Build History**(SS4)**



Steps

1. Extract the provided ZIP file and push **all files and folders** to your GitHub repository and name it as your SRN_Jenkins.
2. Use the following commands on git bash

git init

git checkout -b main

git remote add origin <your-repository-url>

git add .

git commit -m "Initial Jenkins lab setup"

git push -u origin main

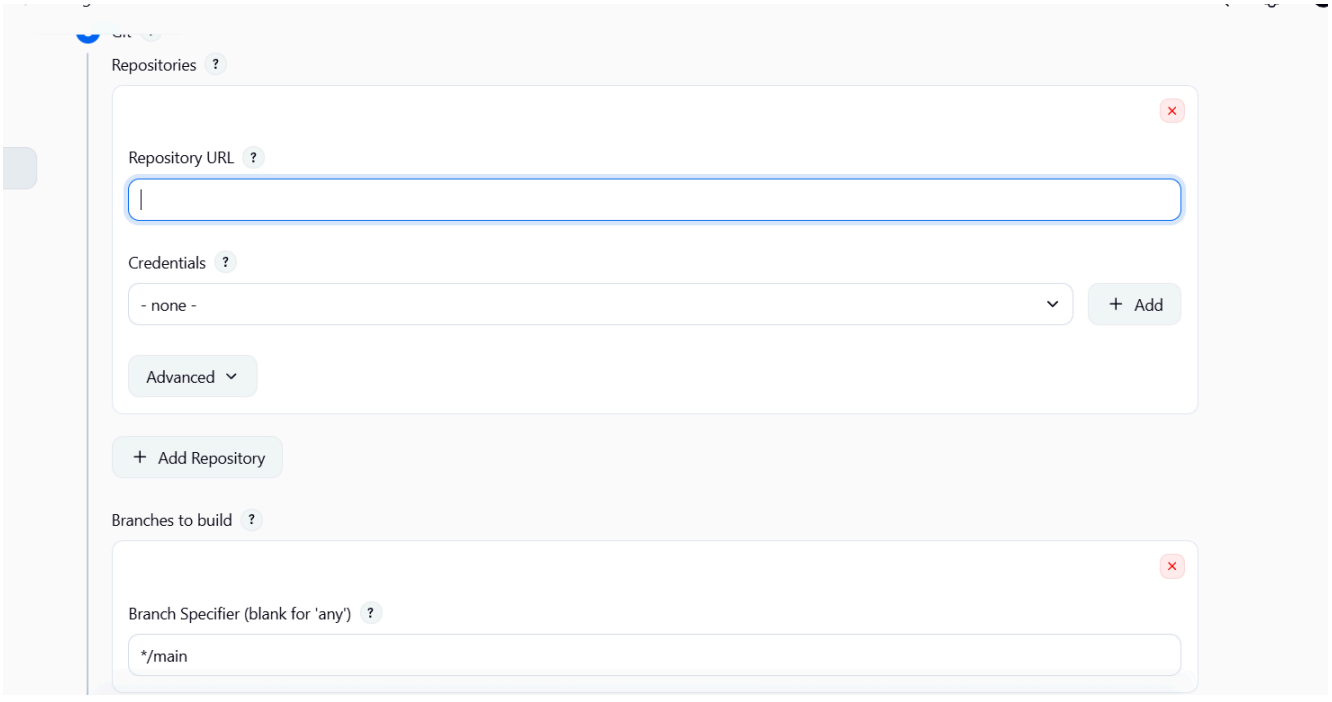
3. Open Jenkins Dashboard → **New Item**

Name the job:
<SRN>-backend-build

4. Select Freestyle Project

5. Under **Source Code Management**:

    - Select **Git**
    - Repository URL: **<your GitHub repo>**
    - Branch: ***/main**

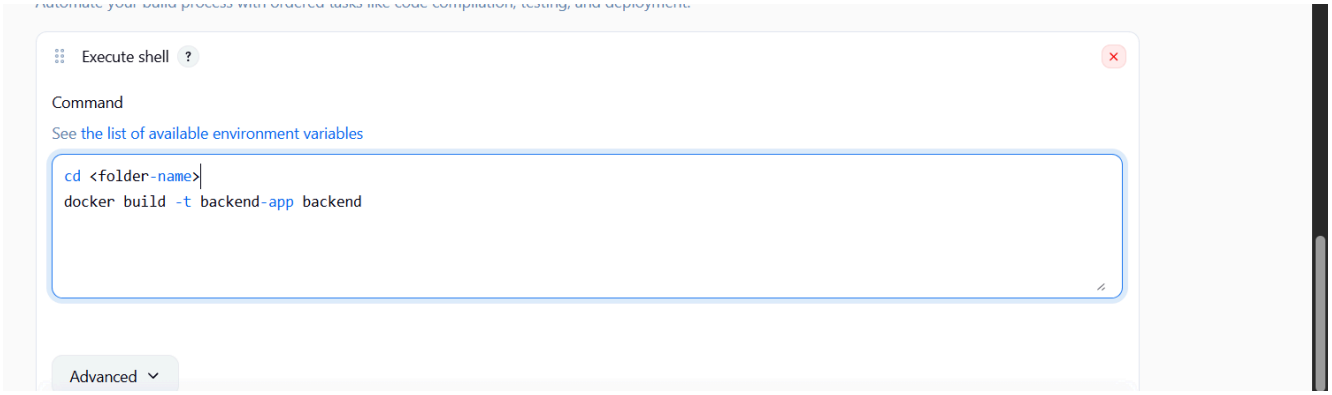


6. Under **Build Triggers**:

    - Select **Poll SCM**

Schedule: **H/5 * * * ***

7. Under **Build**:
    - Click **Execute Shell**



PASTE THIS IN THE SHELL

cd CC_LAB-6

docker build -t backend-app backend

8. Click **Save**
9. Click **Build Now**

Take Screenshots 3 and 4 showing the build was successful.

Task-3: Parameterized Jenkins Job

Aim

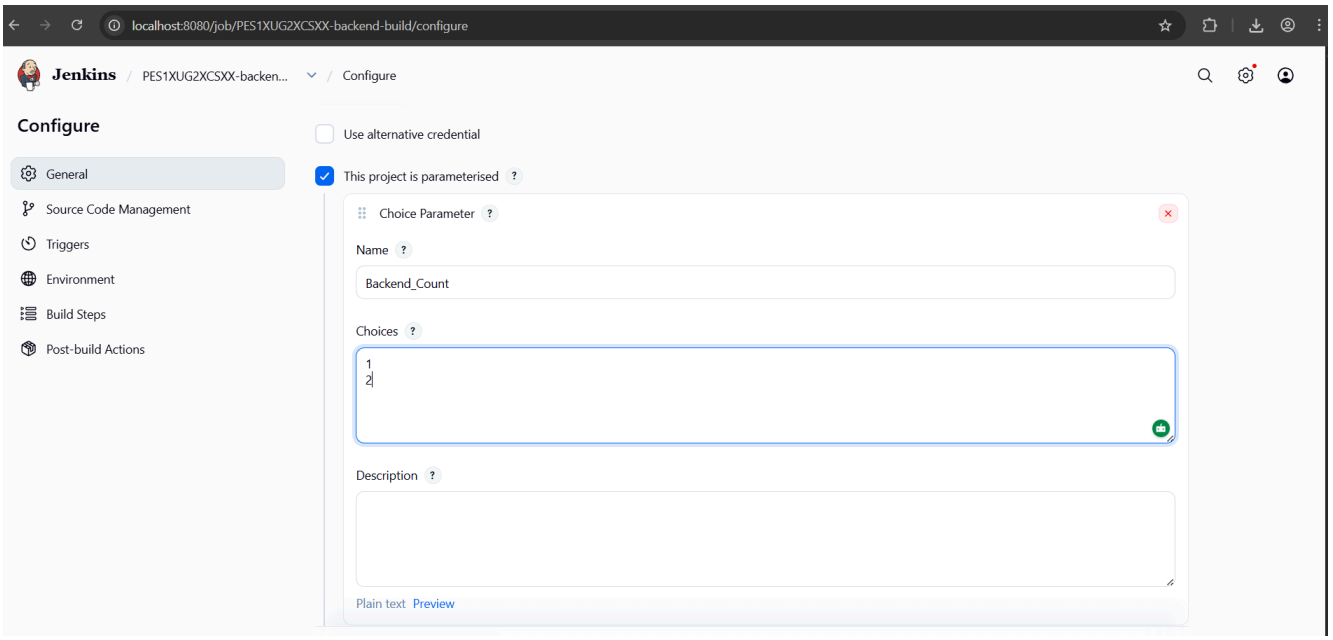
To understand how Jenkins jobs can be parameterized to control application deployment behavior.

Steps

1. Open the Jenkins dashboard and select the job created in Task-2.
2. Click Configure.
3. Enable the option This project is parameterized.
4. Add a Choice Parameter with the following details:

Name: **Backend_Count**

Under choices, add **1 and 2** one after the other, as shown in the below screenshot.



5. Scroll down to the Build section and modify the Execute Shell command as shown below:

cd CC_LAB-6

docker rm -f backend1 backend2 || true

if ["$BACKEND_COUNT" = "1"]; then

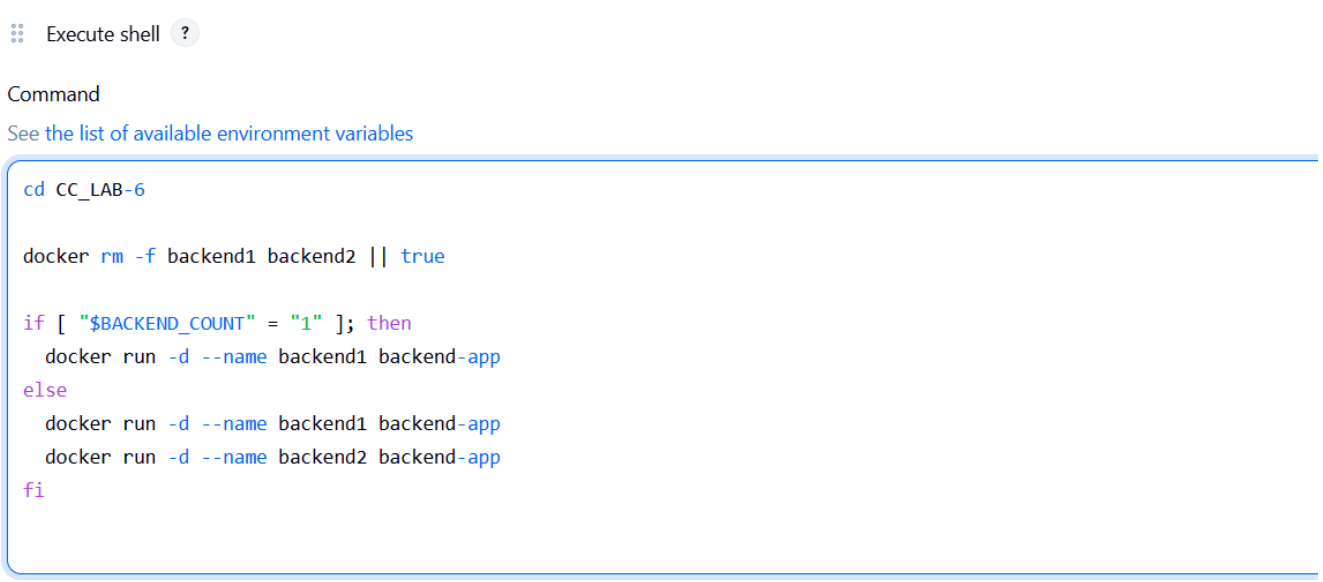
docker run -d --name backend1 backend-app

else

docker run -d --name backend1 backend-app

docker run -d --name backend2 backend-app

fi

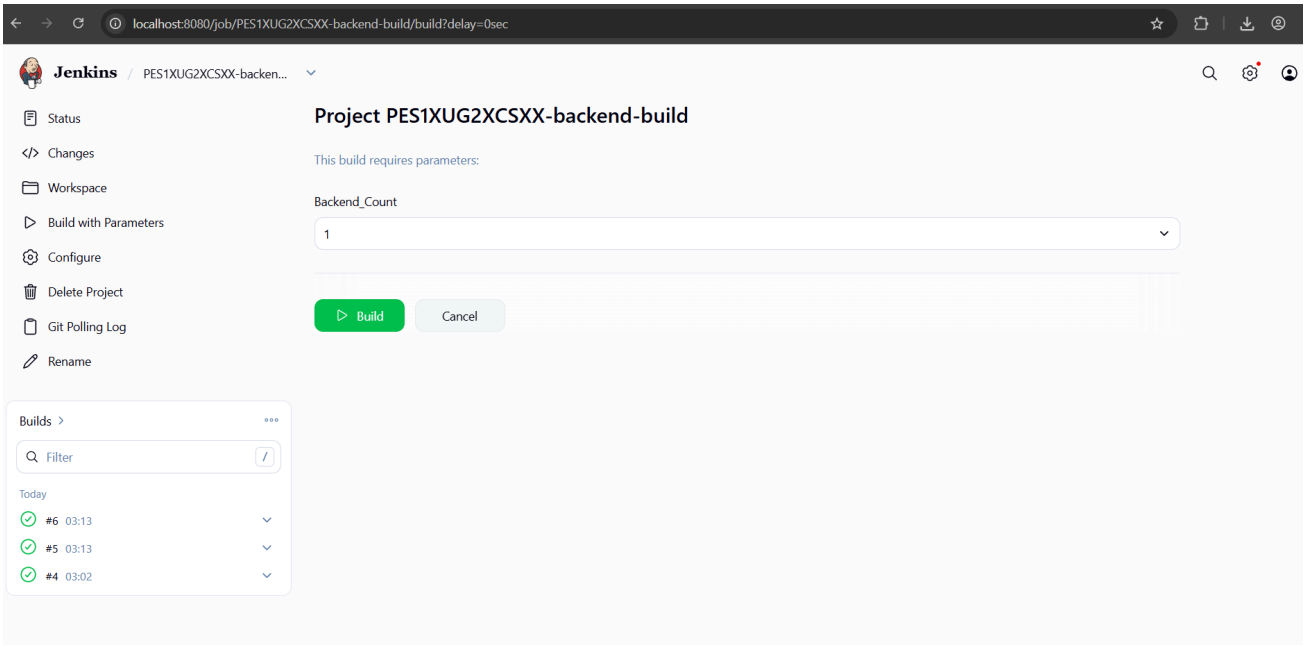


6. Click **Save**.
7. Click **Build with Parameters**.
8. Run the job twice:
   Once with BACKEND_COUNT = 1

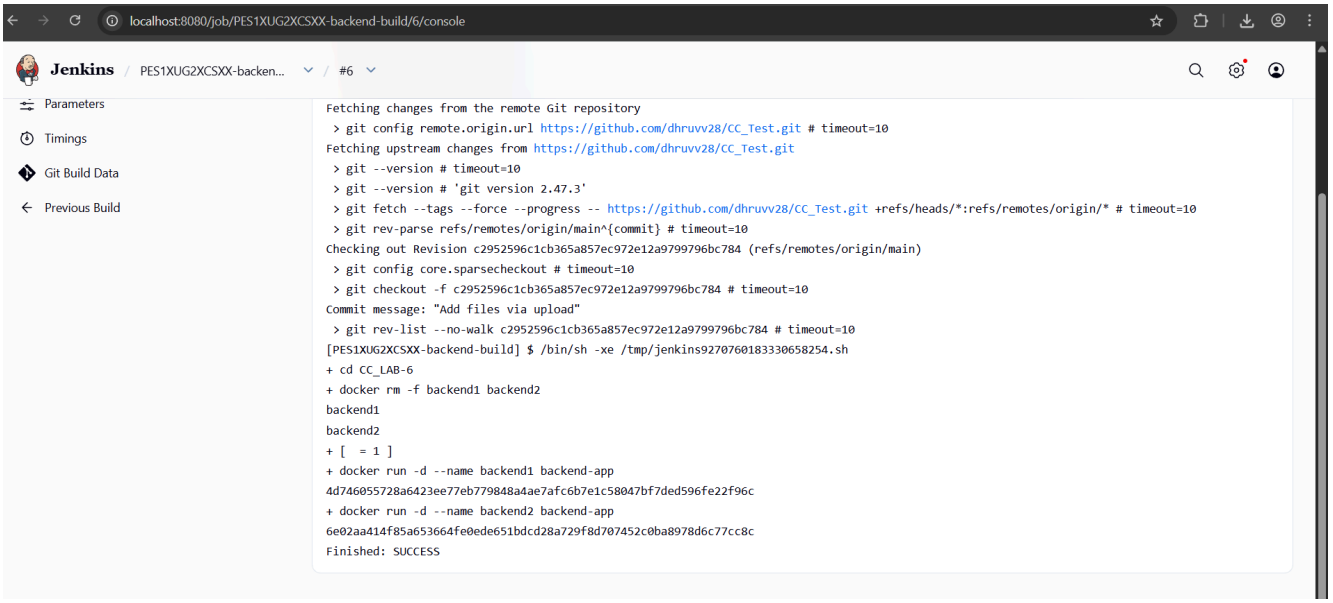
   Once with BACKEND_COUNT = 2

Deliverables

- Screenshot of **Build with Parameters** page.**(SS5)**



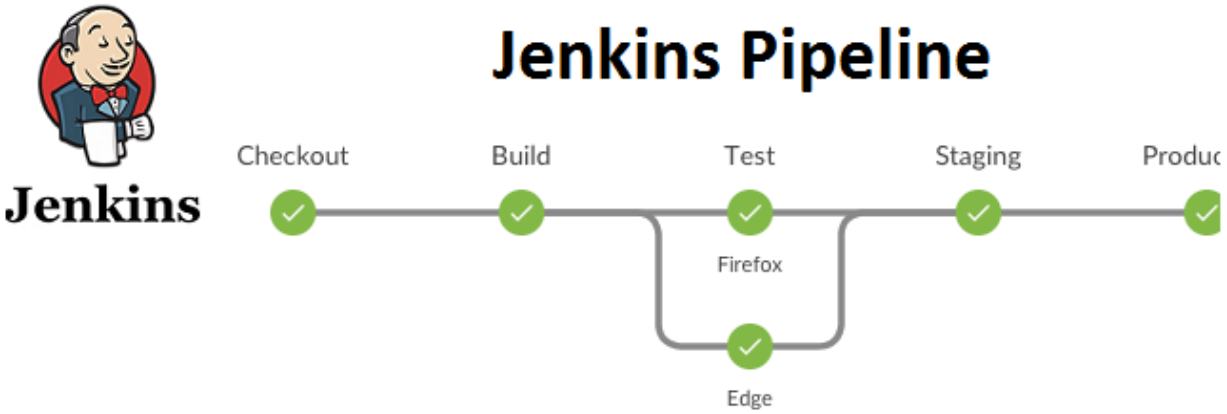
- Screenshot of console output for both builds.**(SS6)**



At this stage, Jenkins is successfully automating the build and deployment of the backend application.

Different configurations are now controlled directly from Jenkins without changing the source code, demonstrating how CI systems manage and validate application behavior.

What is Jenkins Pipeline?



In simple words, Jenkins Pipeline is a combination of plugins that support the integration and implementation of continuous delivery pipelines using Jenkins. The pipeline as Code describes a set of features that allow Jenkins users to define pipelined job processes with code, stored and versioned in a source repository.

Why do we need to use Jenkins Pipeline?

- Pipelines are better than freestyle jobs, you can write a lot of complex tasks using pipelines when compared to Freestyle jobs.
- You can see how long each stage takes to execute so you have more control compared to freestyle.
- Pipeline is a Groovy based script that has a set of plug-ins integrated for automating the builds, deployment and test execution.
- Pipeline defines your entire build process, which typically includes stages for building an application, testing it and then delivering it.
- You can use a snippet generator to generate pipeline code for the stages where you don't know how to write groovy code.

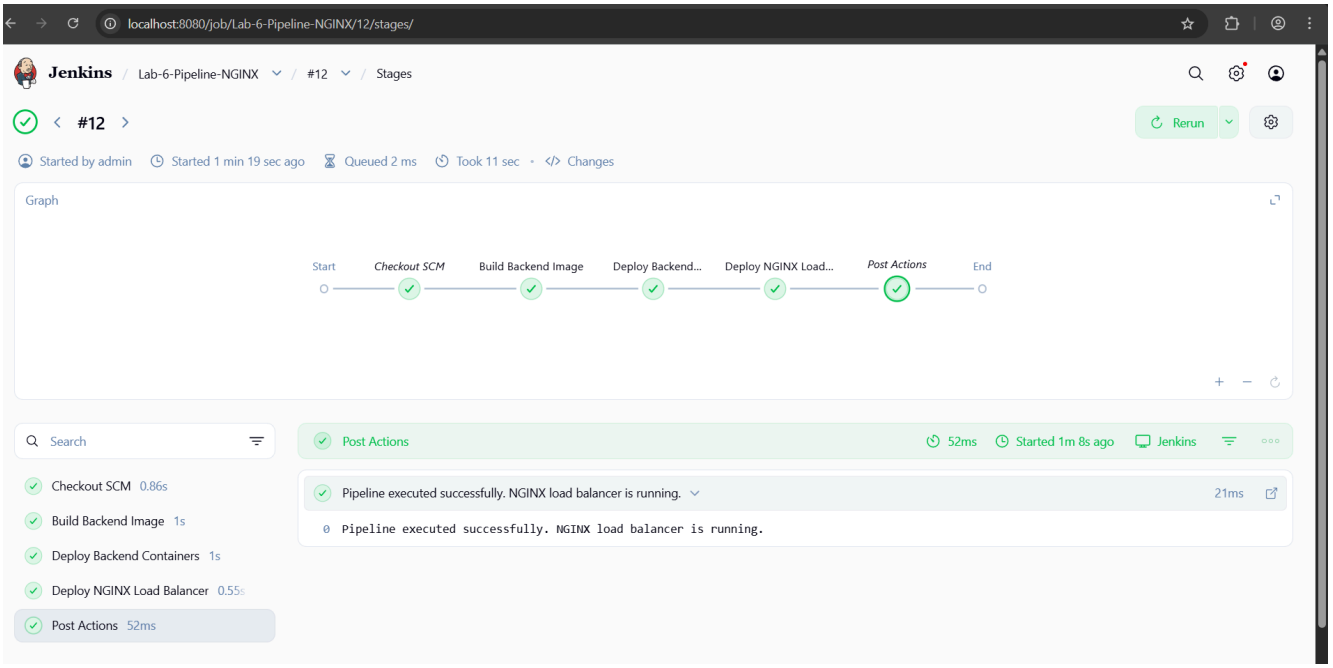
Task-4: Jenkins Pipeline for Automated Deployment

Aim

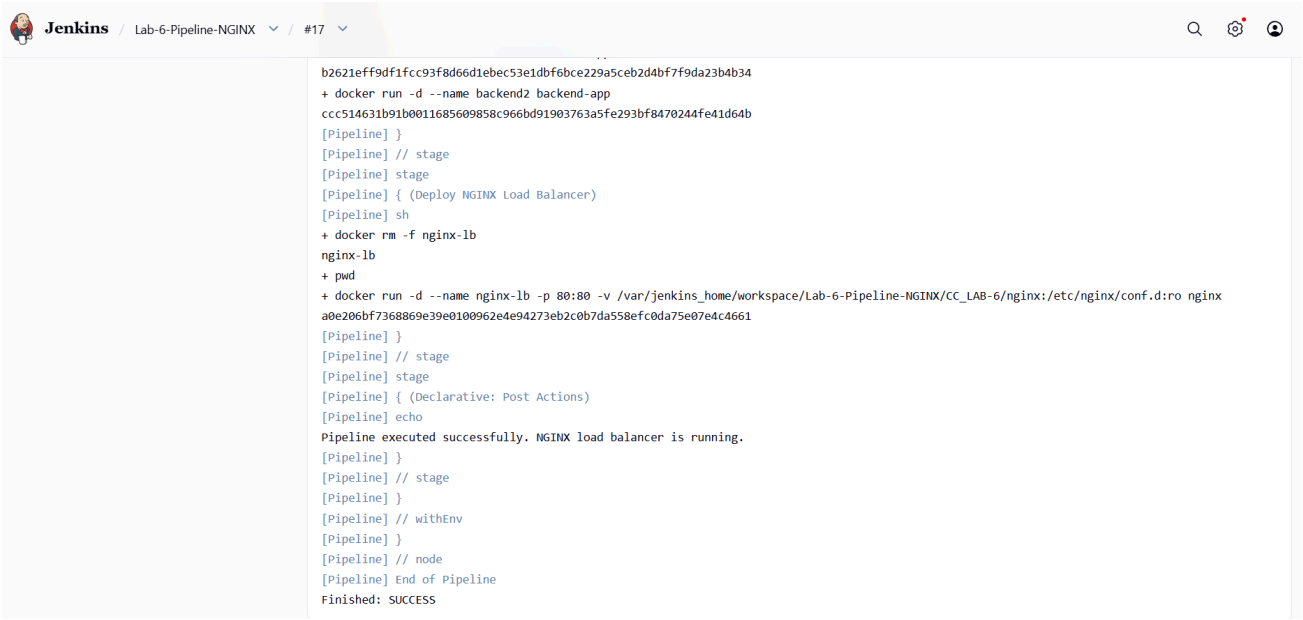
To define the build and deployment process as code using a Jenkins Pipeline and observe automated deployment behavior.

Deliverables

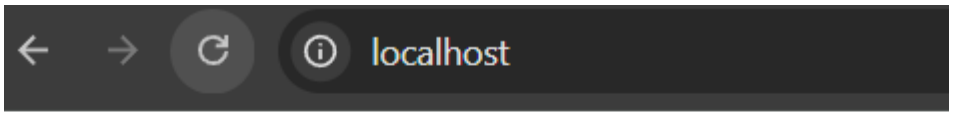
- Screenshot of **Jenkins Stage View**



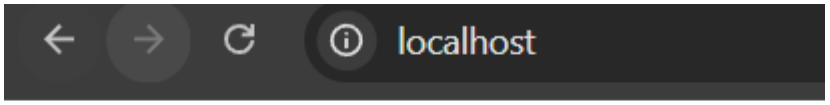
- Screenshot of **Console Output**



- Browser screenshot showing backend responses (you have to refresh the page and see another similar response - add both the screenshots)



Served by backend: 7c12c222d0a7



Served by backend: 44fad774dc8b

Procedure

Step 1: Create a Jenkins Pipeline Job

1 . Open the **Jenkins Dashboard**.

2 . Click **New Item**.

Enter the job name:

3 . Select **Pipeline** and click **OK**.

Step 2: Configure Pipeline from SCM

1 . Scroll to the **Pipeline** section.

Set **Definition** to:

“Pipeline script from SCM”

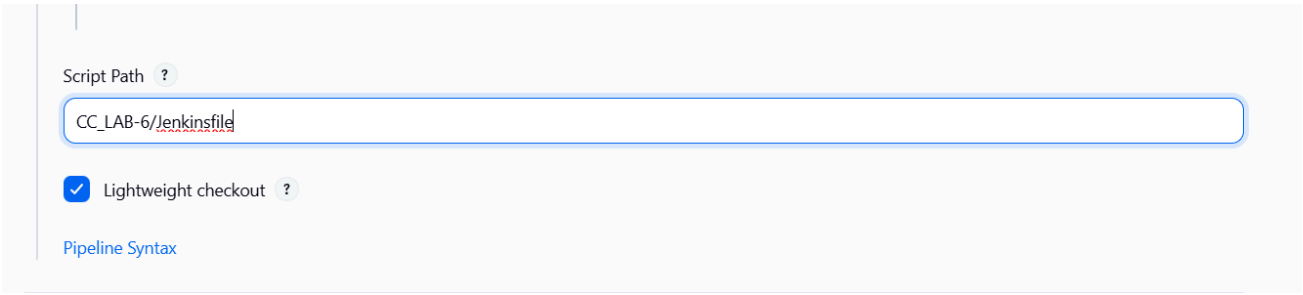
2 . Select **SCM** as **Git**.

3 . Enter your **GitHub Repository URL**.

Set **Branch Specifier** to:

***/main**

4 . Click **Save**.

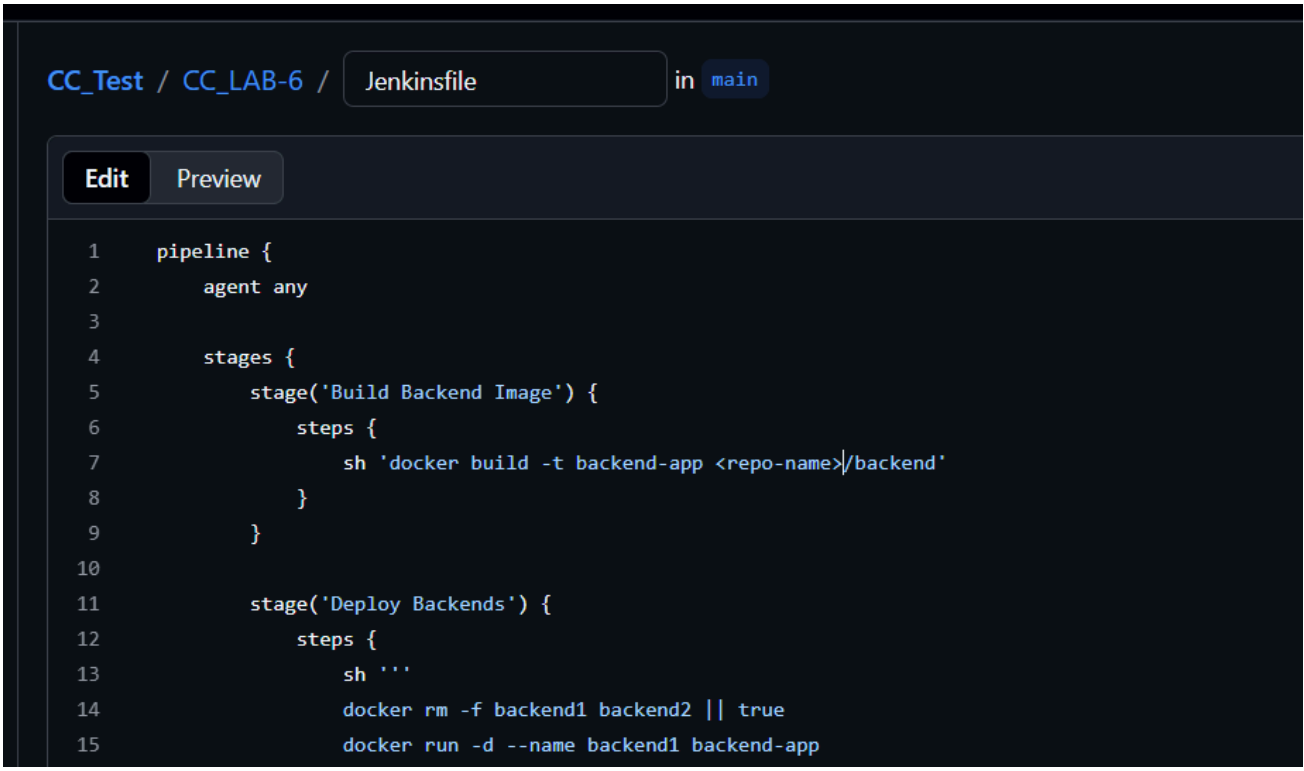


Note:

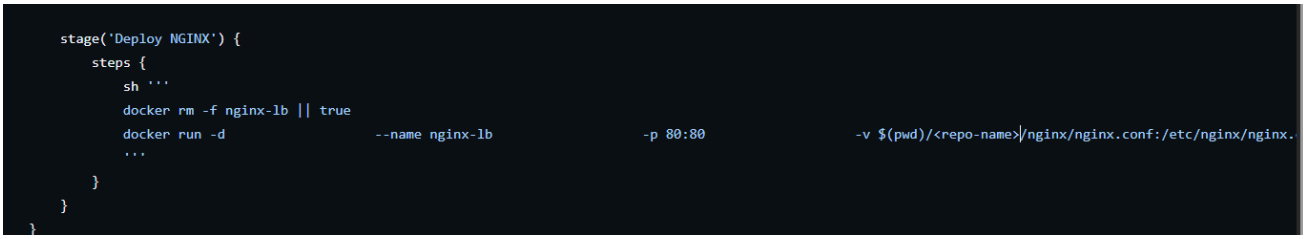
Give the exact path of the jenkins file, otherwise the build will fail!

Make sure the paths with your repo match in the Jenkins file.

For simplicity, ensure the name in the repo is CC_LAB-6.



(In my case it would be CC_Lab-6)



Change it here as well

Step 3: Run the Pipeline

1 . Click **Build Now**.

2. Observe the pipeline execution.

3. Ensure all stages complete successfully (green).

Task 5: Understanding NGINX Load Balancing Strategies

Aim

To study how NGINX distributes client requests using different load-balancing strategies and observe their impact on backend selection.

Deliverables

- Screenshot of modified nginx/default.conf showing different load balancing methods

Procedure

Step 1: Understand the Current Setup

Your current NGINX configuration uses **Round-Robin** load balancing by default.

Open the file: **nginx/default.conf**

You should see:

```
nginx
upstream backend_servers {
   server backend1:8080;
   server backend2:8080;
}

server {
   listen 80;

   location / {
      proxy_pass http://backend_servers;
   }
}
```

Note: Round-Robin means NGINX sends requests alternately to each backend server in sequence.

Step 2: Verify Round-Robin Behavior

1. Open your browser and navigate to:
http://localhost

2. Refresh the page multiple times (at least 5-6 times)

3. Observe the responses alternate between:

Served by backend: backend1
Served by backend: backend2
Served by backend: backend1
Served by backend: backend2

Step 3: Test Least Connections Strategy

This strategy sends new requests to the backend with the fewest active connections.

1. Modify nginx/default.conf:

```
nginx
upstream backend_servers {
   least_conn;
   server backend1:8080;
   server backend2:8080;
}

server {
   listen 80;

   location / {
      proxy_pass http://backend_servers;
   }
}
```

2. Save the file
3. Commit and push changes to GitHub:

bash
git add nginx/default.conf
git commit -m "Changed to least_conn load balancing"
git push origin main

4. Go to Jenkins Dashboard and click **Build Now** on your pipeline
5. Wait for the pipeline to complete successfully
6. Refresh your browser at **http://localhost** multiple times
7. **Take a screenshot** showing the backend responses

Expected Behavior: Since both backends handle requests equally fast in this simple setup, you'll still see fairly even distribution, but NGINX is now tracking connection counts rather than just alternating.

In case '502 Bad Gateway' shows up, it's okay. Add the screenshot.

Step 4: Test IP Hash Strategy

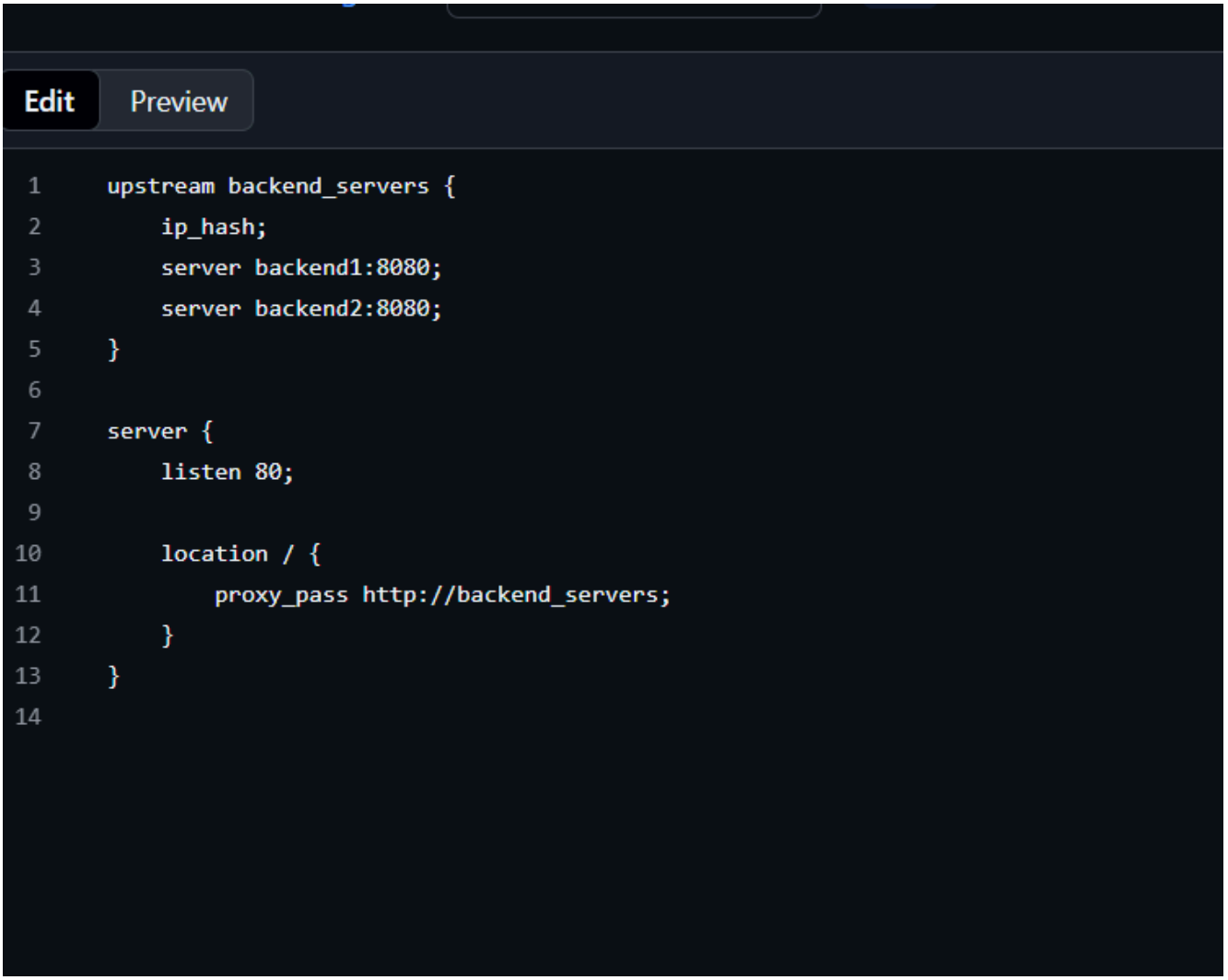
This strategy ensures requests from the same client IP always go to the same backend.

1. Modify **nginx/default.conf**:

```
nginx
upstream backend_servers {
   ip_hash;
   server backend1:8080;
   server backend2:8080;
}

server {
   listen 80;

   location / {
      proxy_pass http://backend_servers;
   }
}
```



2. **Save, commit, and push**:

bash
git add nginx/default.conf
git commit -m "Changed to ip_hash load balancing"
git push origin main

3. Trigger the Jenkins pipeline again
4. Wait for completion
5. Refresh your browser at **http://localhost** multiple times
6 . **Take a screenshot** showing responses

Expected Behavior: All requests from your browser should consistently go to the **same backend** (either backend1 OR backend2, but not alternating). This is because your IP address is being used to determine routing.

Troubleshooting

If NGINX shows "502 Bad Gateway":

- Check that backend containers are running: **docker ps**
- Check backend logs: **docker logs backend1**
- Verify backends are listening on port 8080
- Make sure all containers are on the same network

If changes don't take effect:

- Make sure you committed and pushed to GitHub
- Verify Jenkins pipeline completed successfully
- Check NGINX reloaded config: docker logs nginx-lb

In case 502 still shows up, add a screenshot anyways. We just want to see that you've learnt something :)