

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.

NOTE: A couple of common errors that may be faced are mentioned at the end of the document. Refer to it.

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


```

+ ] Building 98.8s (6/6) FINISHED
=> [internal] load build definition from Dockerfile.jenkins
=> => transferring dockerfile: 178B
=> [internal] load metadata for docker.io/jenkins/jenkins:lts
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/2] FROM docker.io/jenkins/jenkins:lts@sha256:d1ea795c6facd7f549a21c40e5e43ffcc5fbc5f48683d9b24750f26e8079d
=> => resolve docker.io/jenkins/jenkins:lts@sha256:d1ea795c6facd7f549a21c40e5e43ffcc5fbc5f48683d9b24750f26e8079d
=> [2/2] RUN apt-get update && apt-get install -y docker.io make g++ curl
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:b27db89f15f01b8b79f4a3c3db6dab9c31e70ceae34183c721c45a717559701d
=> => exporting config sha256:d0b10a5e6aa8aa3b955d87af7c6e214a3cfc97f6f8efe088ee6a9e2ddbe8633b
=> => exporting attestation manifest sha256:89ebb2aa7383ddcc6ef7d32f306329eeb6dafb01bc1630d4a3265836d72cc73e
=> => exporting manifest list sha256:b6039ad4387c8ce63b3ale474527dd5bae9251423b28f059b7751361b415fde3
=> => naming to docker.io/library/jenkins-docker:latest
=> => unpacking to docker.io/library/jenkins-docker:latest
view build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/oqverh049be01izamr4lo1zz9

```

3. Run Jenkins container:

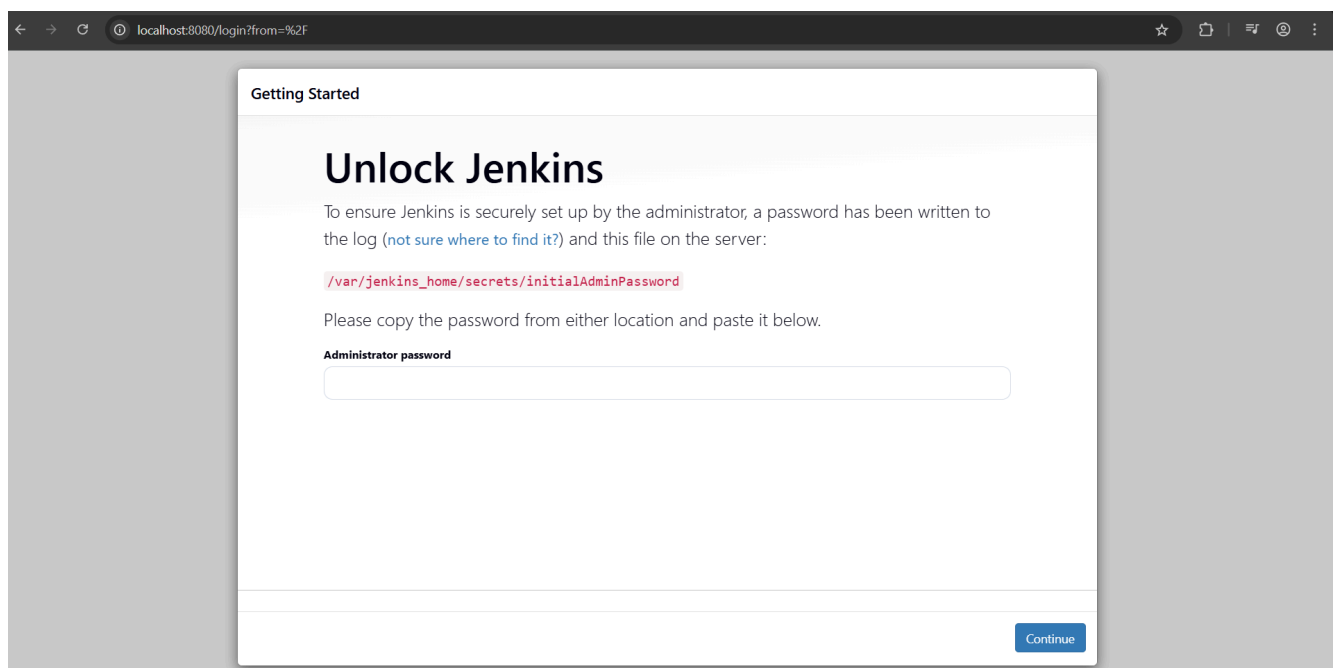
```
docker run -d -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home -v /var/run/docker.sock:/var/run/docker.sock --name jenkins jenkins-docker
```

This will start Jenkins using the custom image with Docker support and persistent storage. **(This might also be the reason your build fails so we would have to rerun as root, please refer to the errors part in the last part of the document if it does fail during Task-2)**

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



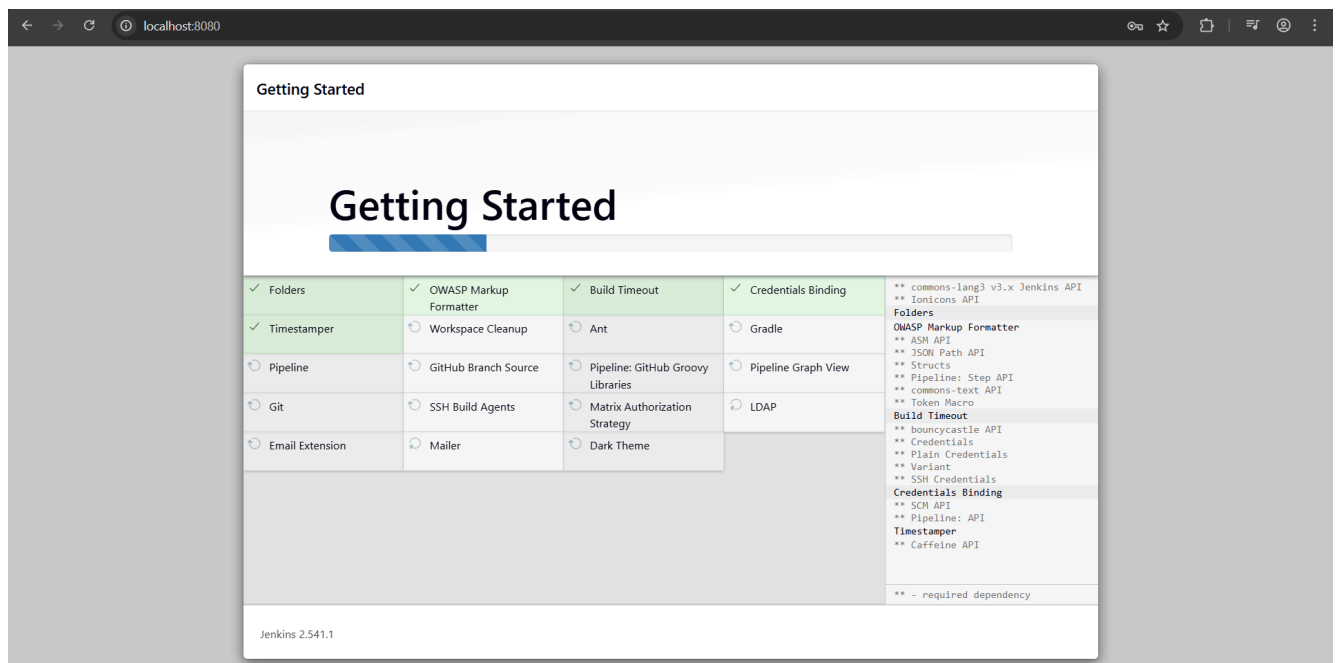
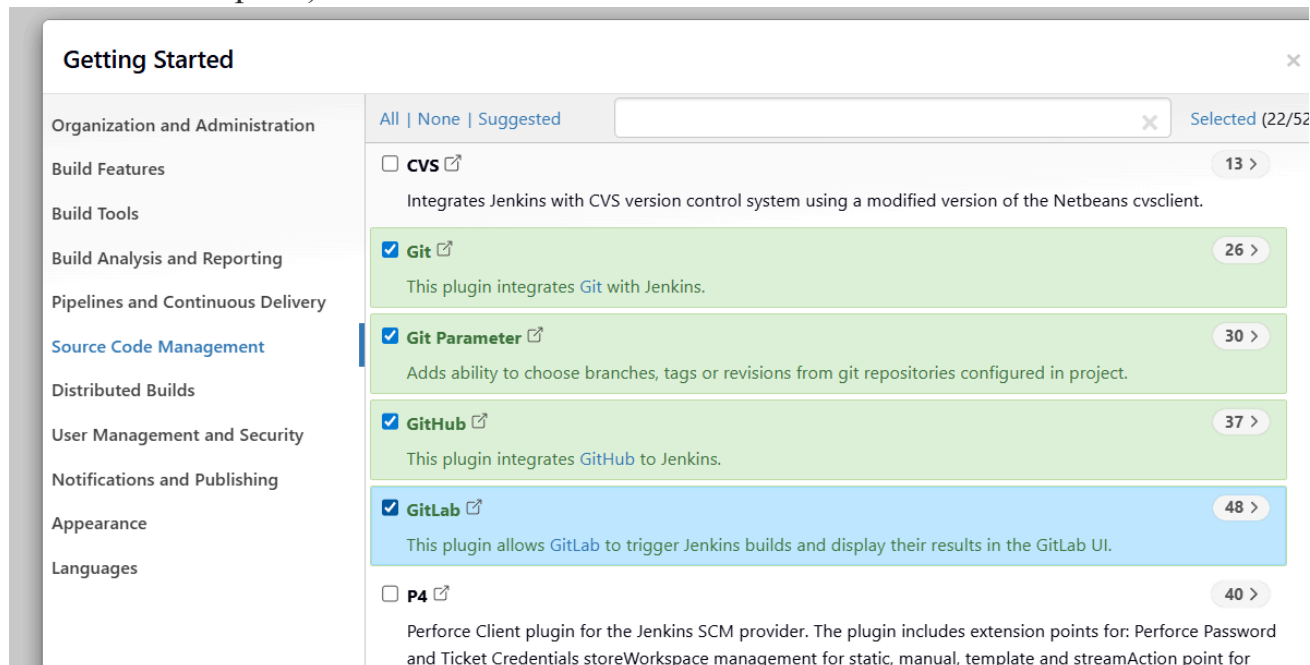
4. Retrieve the initial admin password:

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

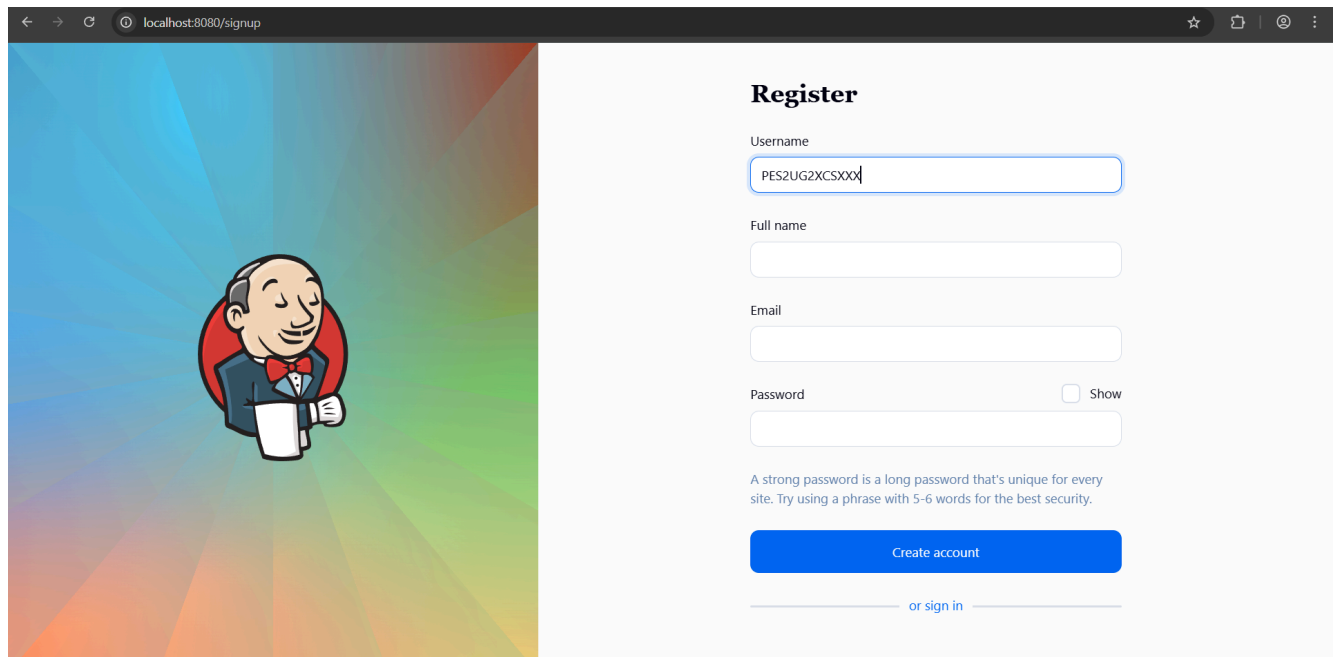
```
docker exec jenkins cat /var/jenkins_home/secrets/initialAdminPassword
```

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.



A screenshot of the Jenkins registration page. On the left is a large illustration of the Jenkins mascot, a man in a blue suit and red bow tie, holding a white document. The background of this illustration is a colorful, low-poly geometric pattern. On the right is the registration form titled "Register". It includes input fields for "Username" (containing "PES2UG2XCSXXX"), "Full name", "Email", and "Password" (with a "Show" checkbox). Below the password field is a note: "A strong password is a long password that's unique for every site. Try using a phrase with 5-6 words for the best security." At the bottom of the form is a blue "Create account" button and a link "or sign in". The browser's address bar shows "localhost:8080/signup".

localhost:8080/signup

Register

Username
PES2UG2XCSXXX

Full name

Email

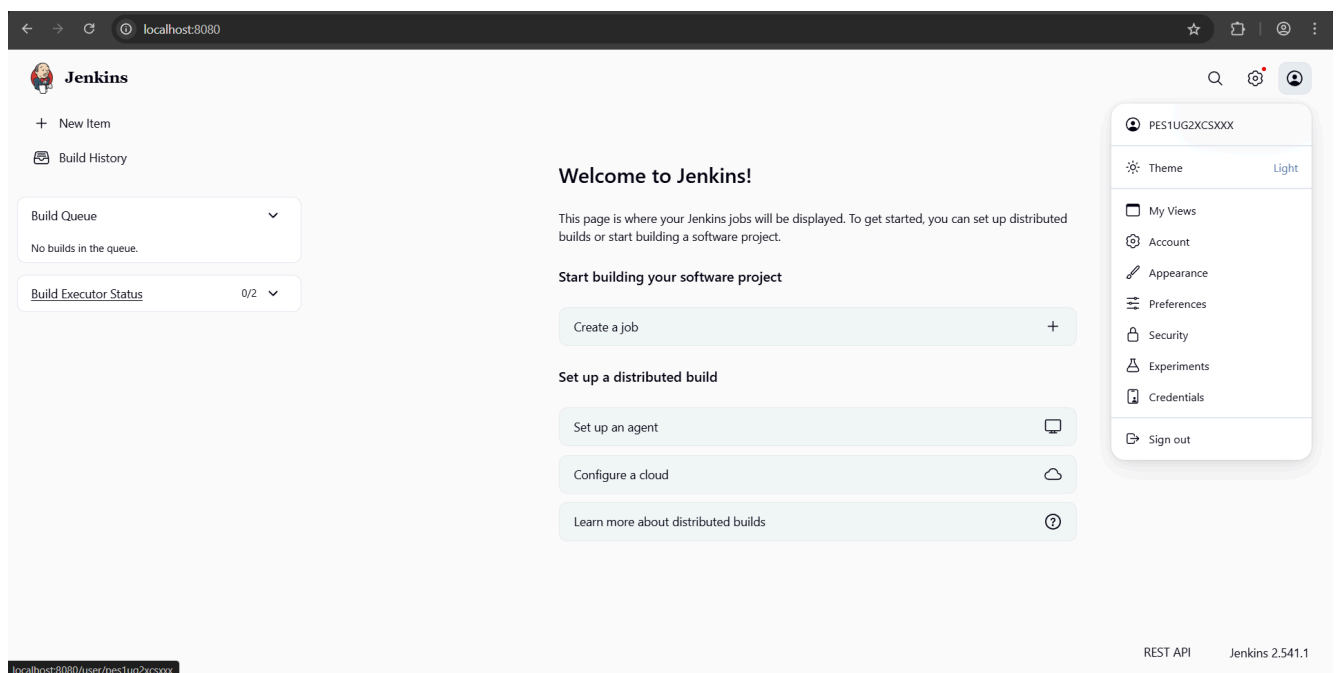
Password ☐ Show

A strong password is a long password that's unique for every site. Try using a phrase with 5-6 words for the best security.

Create account

[or sign in](#)

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



A screenshot of the Jenkins dashboard. The top left shows the Jenkins logo and navigation links: "+ New Item" and "Build History". Below these are two status boxes: "Build Queue" (No builds in the queue) and "Build Executor Status" (0/2). The main content area is titled "Welcome to Jenkins!" and contains instructions: "This page is where your Jenkins jobs will be displayed. To get started, you can set up distributed builds or start building a software project." Below this are three sections: "Start building your software project" with a "Create a job" button; "Set up a distributed build" with buttons for "Set up an agent", "Configure a cloud", and "Learn more about distributed builds". On the right is a user profile dropdown menu for "PES1UG2XCSXXX" with options: Theme (Light), My Views, Account, Appearance, Preferences, Security, Experiments, Credentials, and Sign out. The bottom of the page shows the browser address bar "localhost:8080/user/pes1ug2xcsox" and the footer "REST API Jenkins 2.541.1".

localhost:8080

Jenkins

+ New Item

Build History

Build Queue
No builds in the queue.

Build Executor Status
0/2

Welcome to Jenkins!

This page is where your Jenkins jobs will be displayed. To get started, you can set up distributed builds or start building a software project.

Start building your software project

Create a job

Set up a distributed build

Set up an agent

Configure a cloud

Learn more about distributed builds

PES1UG2XCSXXX

Theme Light

My Views

Account

Appearance

Preferences

Security

Experiments

Credentials

Sign out

localhost:8080/user/pes1ug2xcsox

REST API Jenkins 2.541.1

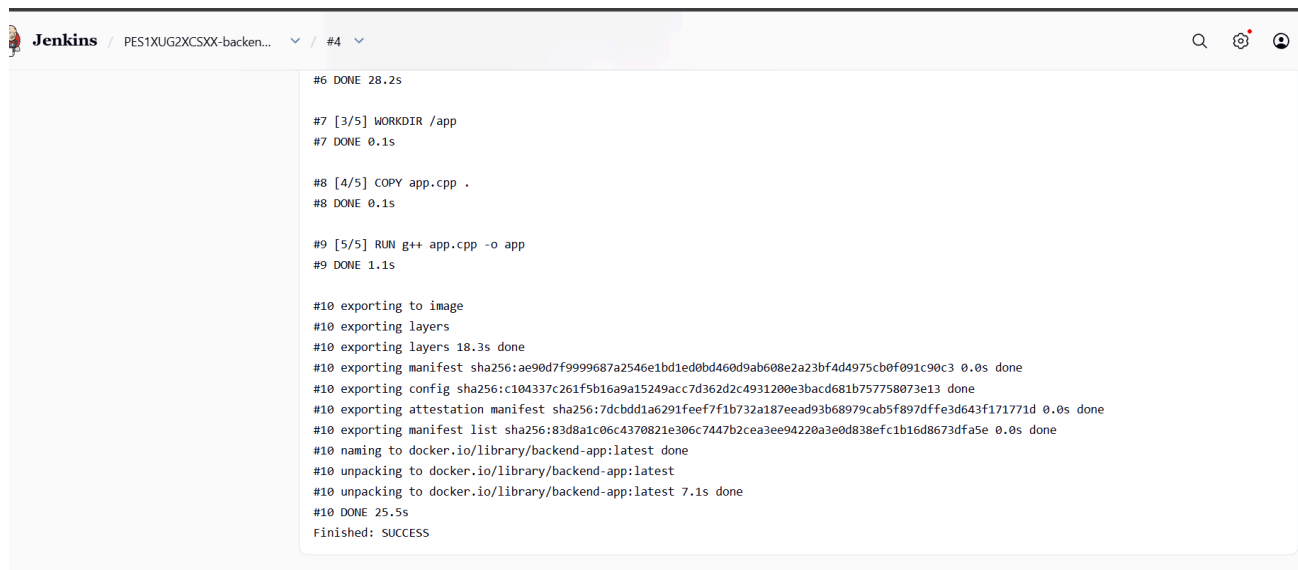
Task-2: Create a Jenkins Job to Build the Backend

Aim

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

Deliverables

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



```
Jenkins / PES1XUG2XCSXX-backend... / #4

#6 DONE 28.2s

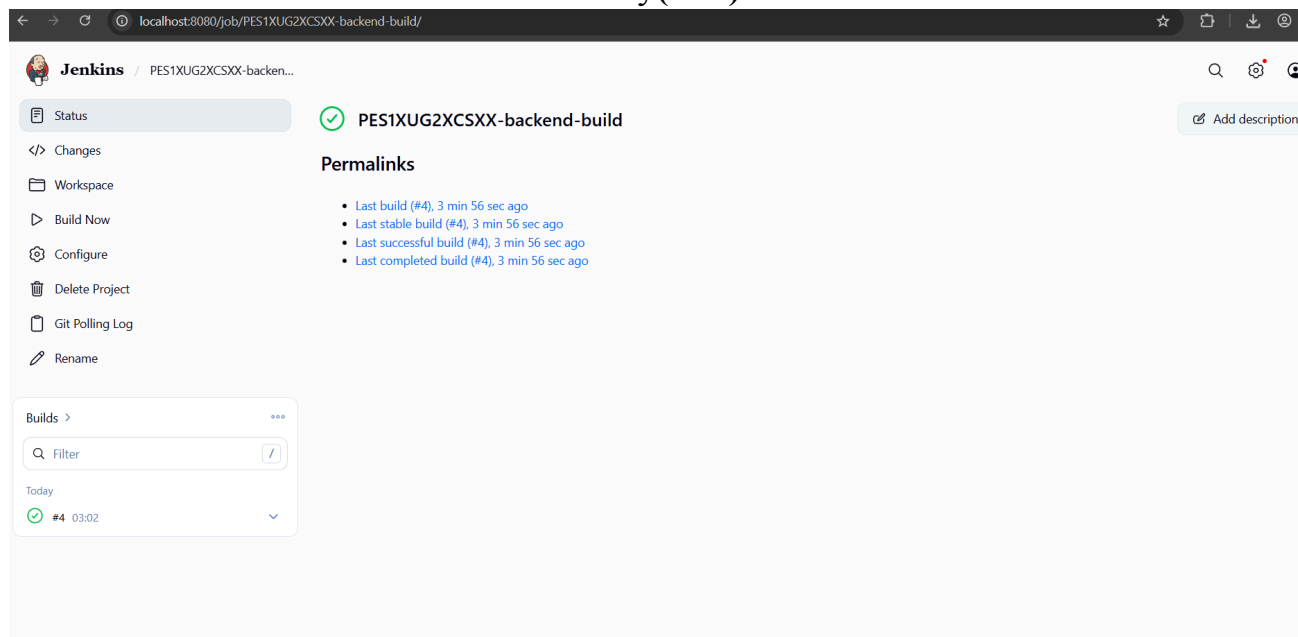
#7 [3/5] WORKDIR /app
#7 DONE 0.1s

#8 [4/5] COPY app.cpp .
#8 DONE 0.1s

#9 [5/5] RUN g++ app.cpp -o app
#9 DONE 1.1s

#10 exporting to image
#10 exporting layers
#10 exporting layers 18.3s done
#10 exporting manifest sha256:ae90d7f9999687a2546e1bd1ed0bd460d9ab08e2a23bf4d4975cb0f091c90c3 0.0s done
#10 exporting config sha256:c104337c261f5b16a9a15249acc7d362d2c4931200e3bacc681b757758073e13 done
#10 exporting attestation manifest sha256:7dcbdd1a6291feef7f1b732a187eead93b68979cab5f897dffe3d643f171771d 0.0s done
#10 exporting manifest list sha256:83d8a1c06c4370821e306c7447b2cea3ee94220a3e0d838efc1b16d8673dfa5e 0.0s done
#10 naming to docker.io/library/backend-app:latest done
#10 unpacking to docker.io/library/backend-app:latest
#10 unpacking to docker.io/library/backend-app:latest 7.1s done
#10 DONE 25.5s
Finished: SUCCESS
```

- 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 **CC_Lab-6 (this naming convention will make it easier for you in the subsequent steps)**

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**

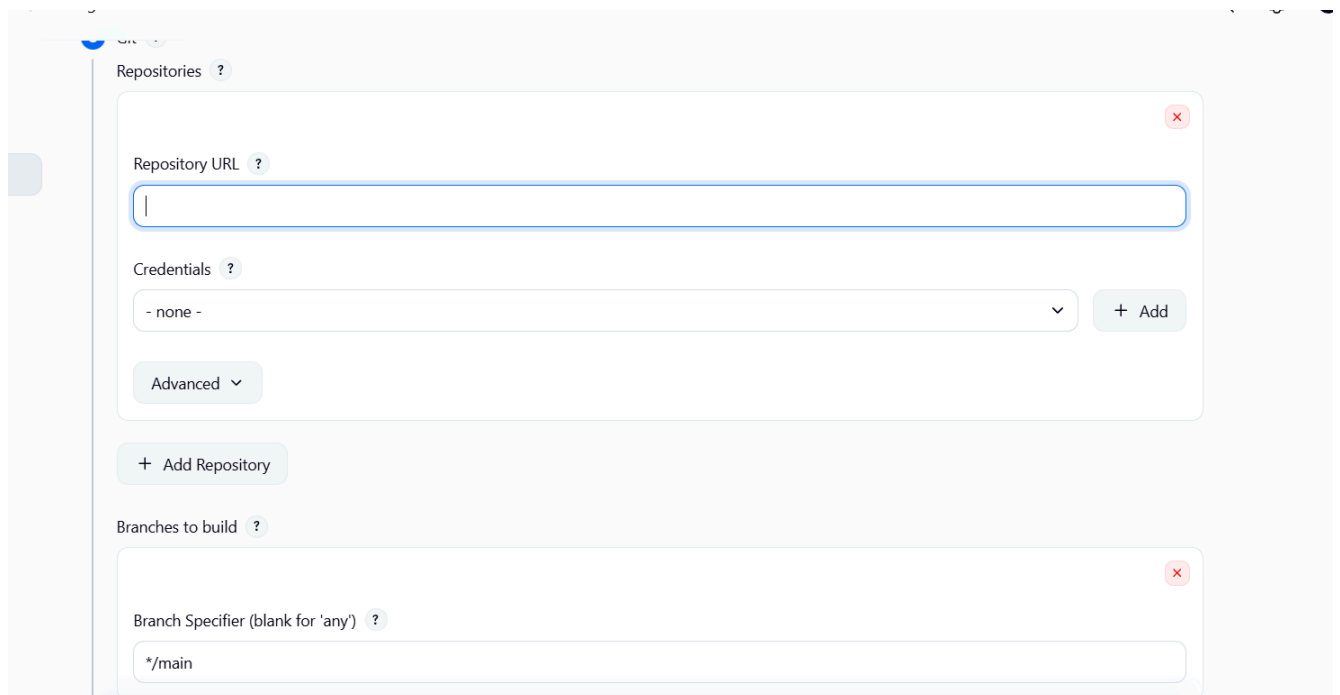
Name the job:

<SRN>-backend-build

4. Select Freestyle Project

5. Under **Source Code Management:**

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

The image shows a screenshot of the Jenkins 'Add Repository' dialog box. The dialog is titled 'Repositories' and contains several input fields. The 'Repository URL' field is empty and has a blue border. Below it, the 'Credentials' dropdown menu is set to '- none -'. There is an 'Advanced' dropdown menu and an '+ Add' button. At the bottom of the dialog, there is a '+ Add Repository' button. Below the main dialog, there is a 'Branches to build' section with a 'Branch Specifier (blank for \'any\')' field containing the text '*/main'. The dialog has a red 'x' icon in the top right corner.

6. Under **Build Triggers:**

- Select **Poll SCM**

Poll SCM ?

Schedule ?

H/5 * * * *

Would last have run at Saturday, February 14, 2026, 2:52:00 AM Coordinated Universal Time; would next run at Saturday, February 14, 2026, 2:57:00 AM Coordinated Universal Time.

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

7. Under **Build:**

- Click **Execute Shell**

Automate your build process with ordered tasks like code compilation, testing, and deployment.

Execute shell ?

Command

See the list of available environment variables

```
cd <folder-name>
docker build -t backend-app backend
```

Advanced ▾

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

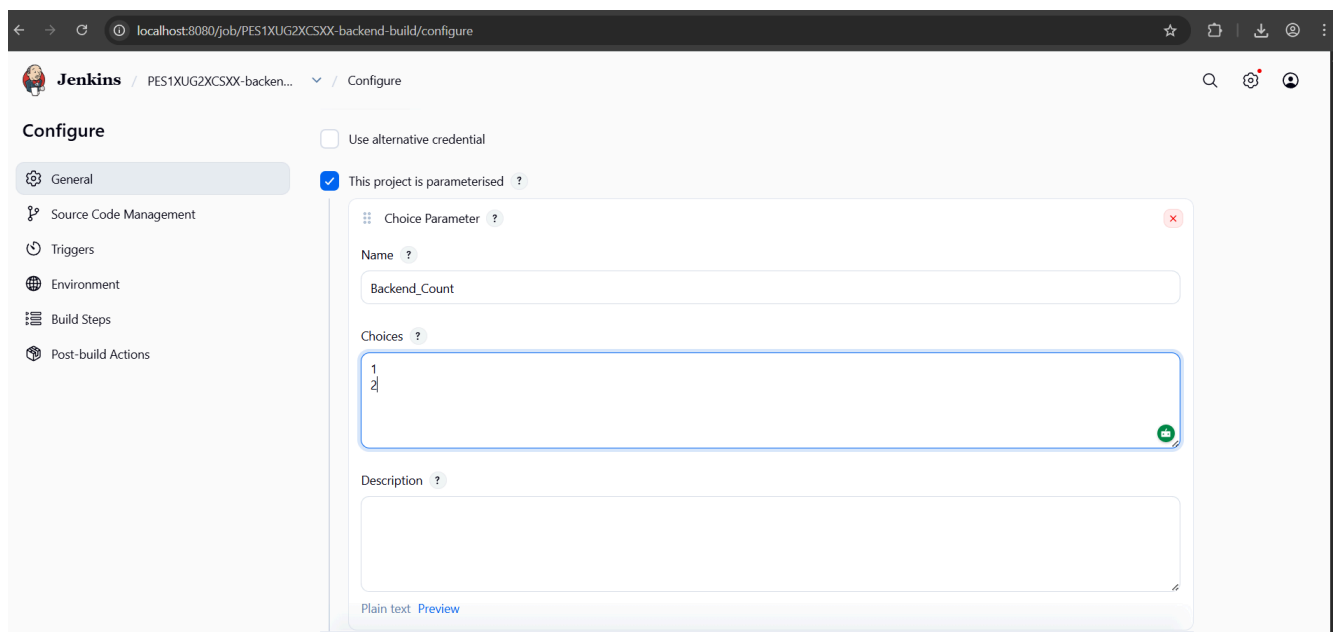
To understand how Jenkins jobs can be parameterized to control application deployment behavior.

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

Execute shell ?

Command

See [the list of available environment variables](#)

```
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**.

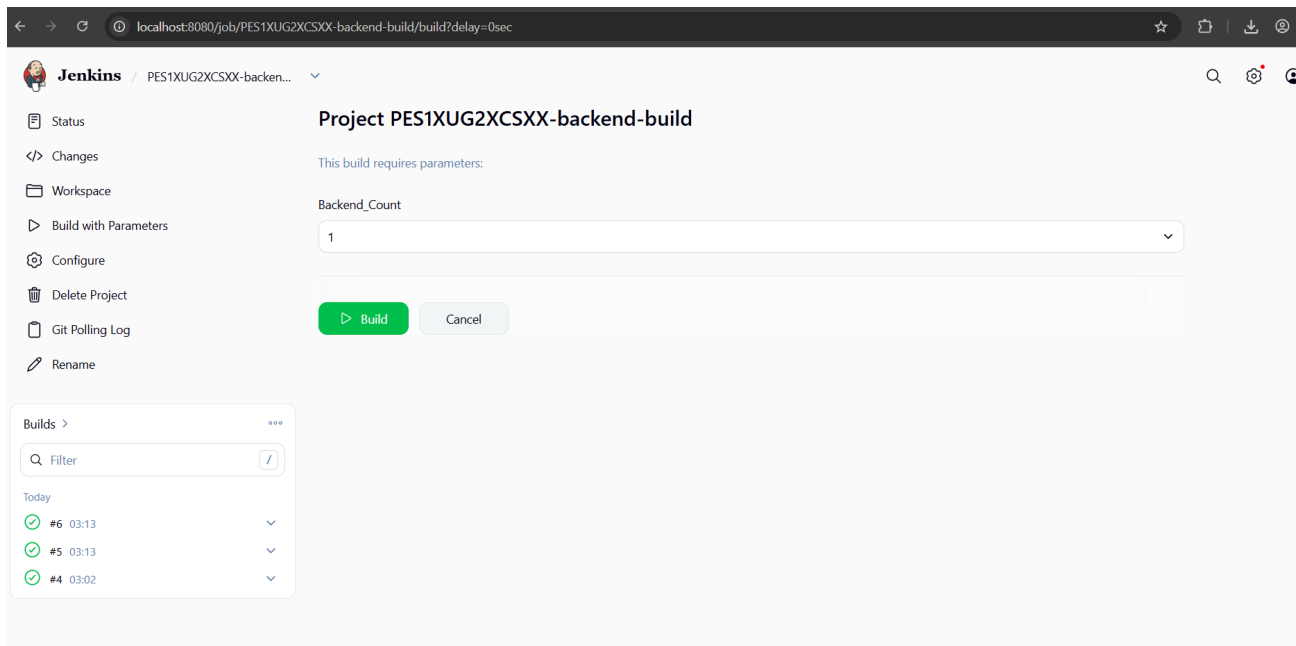
8. Run the job twice:

Once with BACKEND_COUNT = 1

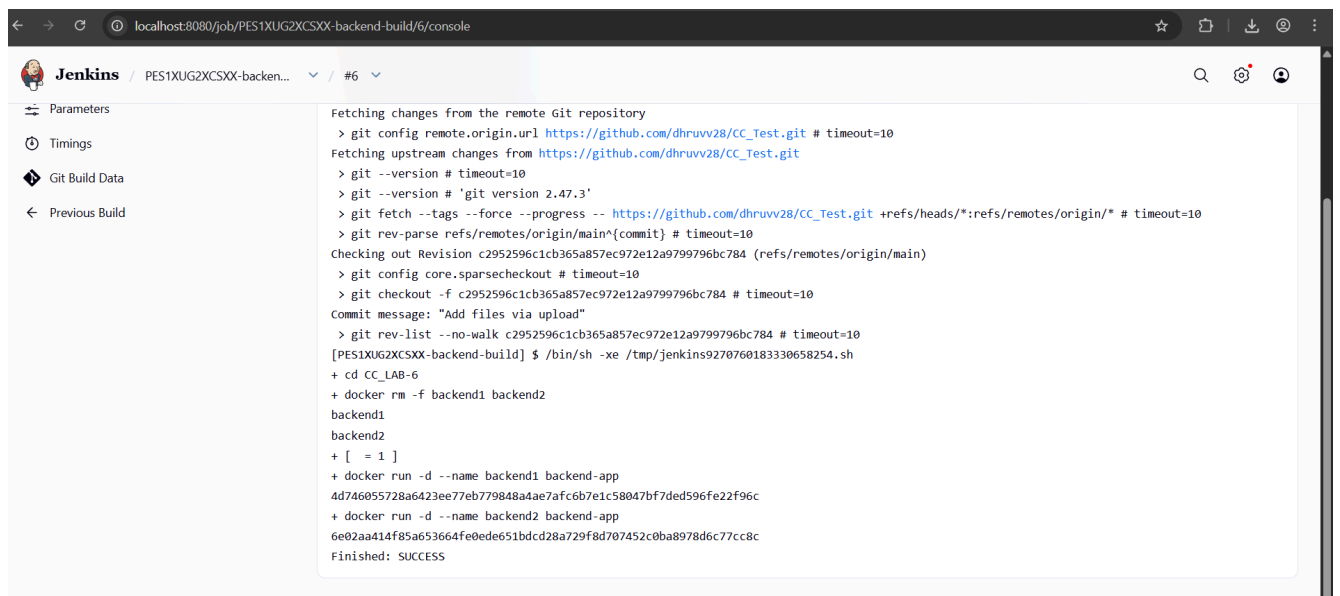
Once with BACKEND_COUNT = 2

Deliverables

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



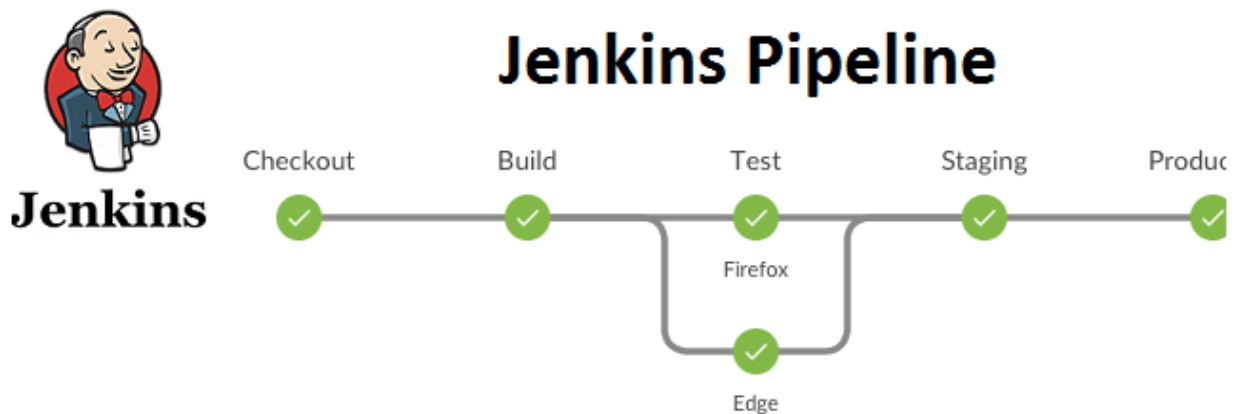
- Screenshot of console output for both builds.(SS6)



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.

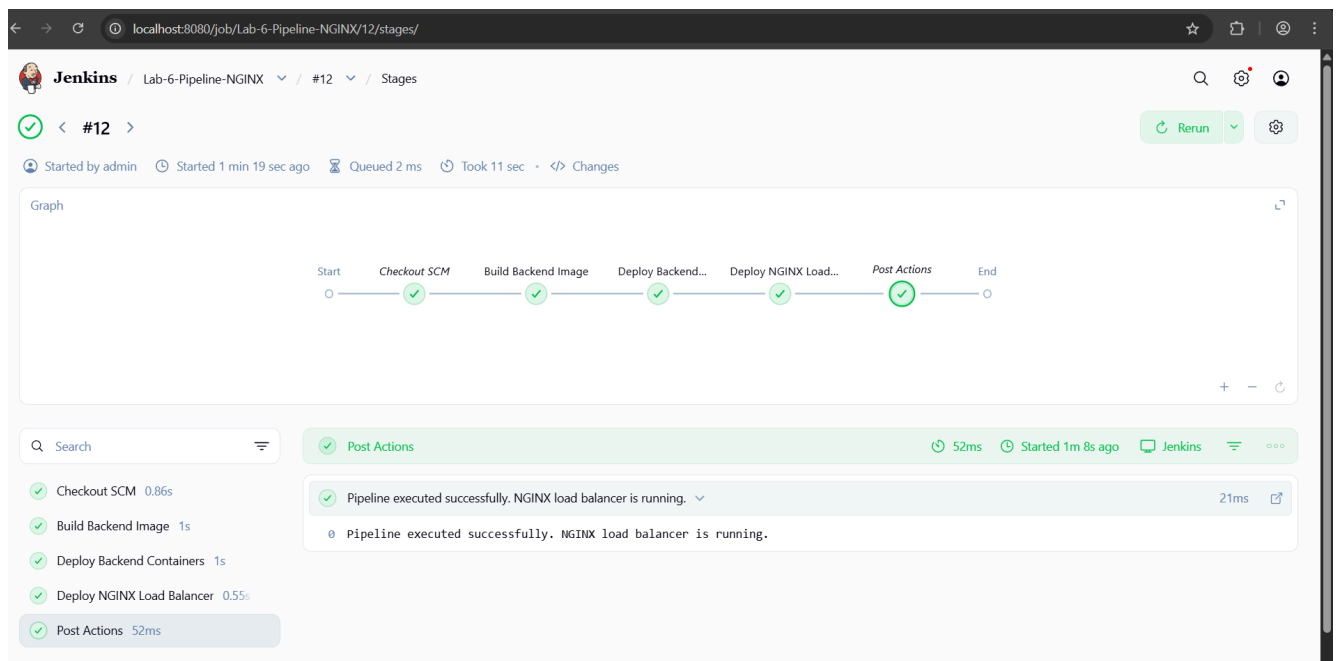
Task-4: Jenkins Pipeline for Automated Deployment

Aim

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

Deliverables

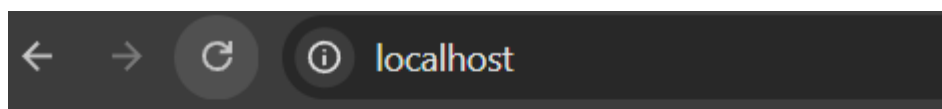
- Screenshot of **Jenkins Stage View**



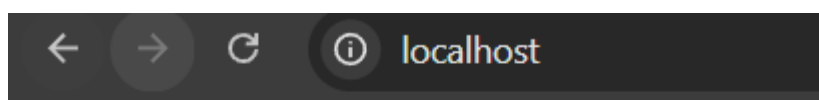
- Screenshot of **Console Output**

```
Jenkins / Lab-6-Pipeline-NGINX / #17
b2621eff9df1fcc93f8d66d1ebec53e1dbf6bce229a5ceb2d4bf7f9da23b4b34
+ docker run -d --name backend2 backend-app
ccc514631b91b0011685609858c966bd91903763a5fe293bf8470244fe41d64b
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy NGINX Load Balancer)
[Pipeline] sh
+ docker rm -f nginx-lb
nginx-lb
+ pwd
+ docker run -d --name nginx-lb -p 80:80 -v /var/jenkins_home/workspace/Lab-6-Pipeline-NGINX/CC_LAB-6/nginx:/etc/nginx/conf.d:ro nginx
a0e206bf7368869e39e0100962e4e94273eb2c0b7da558efc0da75e07e4c4661
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] echo
Pipeline executed successfully. NGINX load balancer is running.
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

- 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:

LAB6-PIPELINE-NGINX

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**.



Pipeline

Define your Pipeline using Groovy directly or pull it from source control.

Definition

Pipeline script from SCM

SCM ?

Git

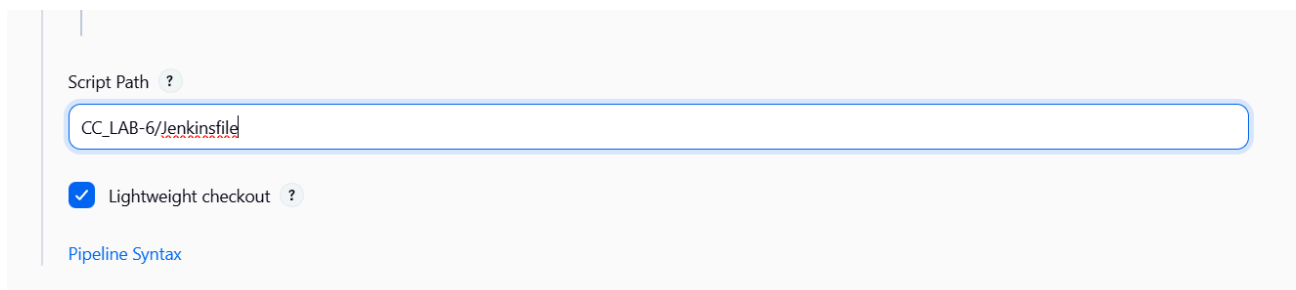
Repositories ?

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

Set **Branch Specifier** to:

***/main**

4 . Click **Save**.



Script Path ?

CC_LAB-6/Jenkinsfile

☒ Lightweight checkout ?

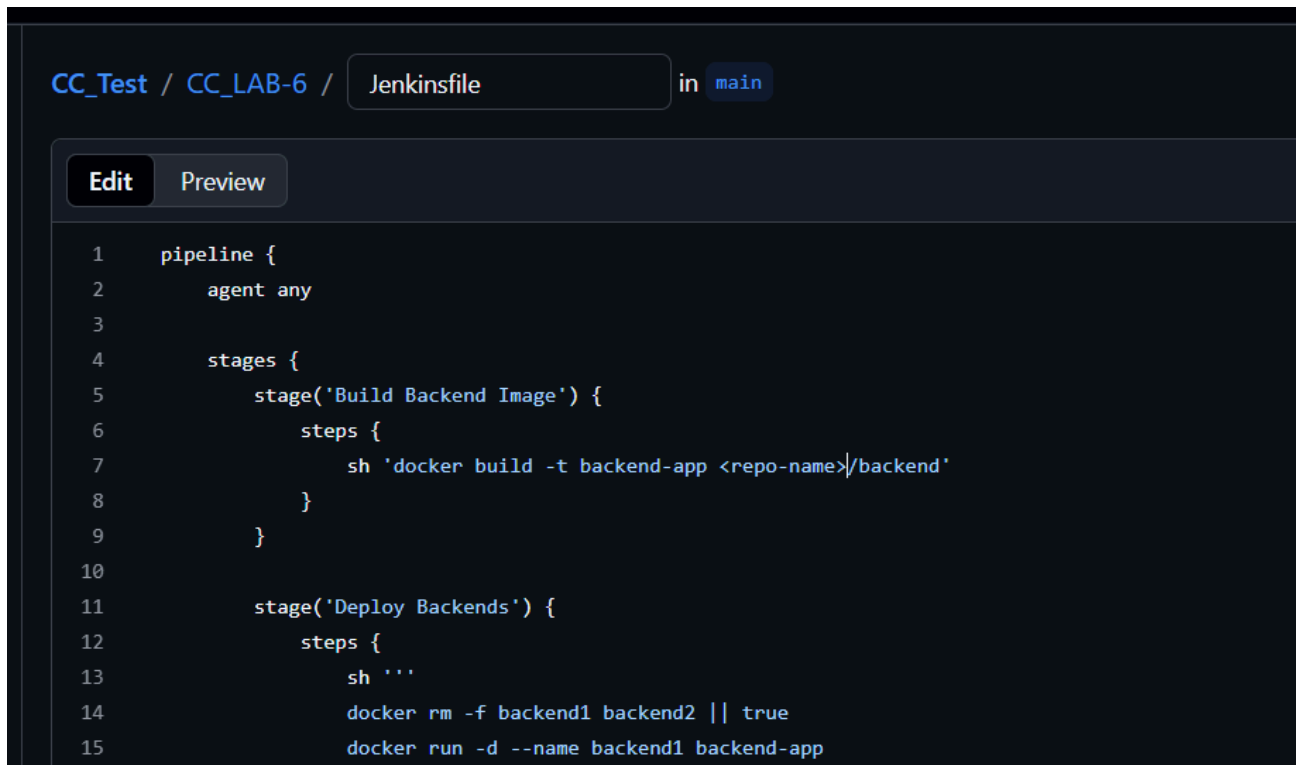
Pipeline Syntax

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.

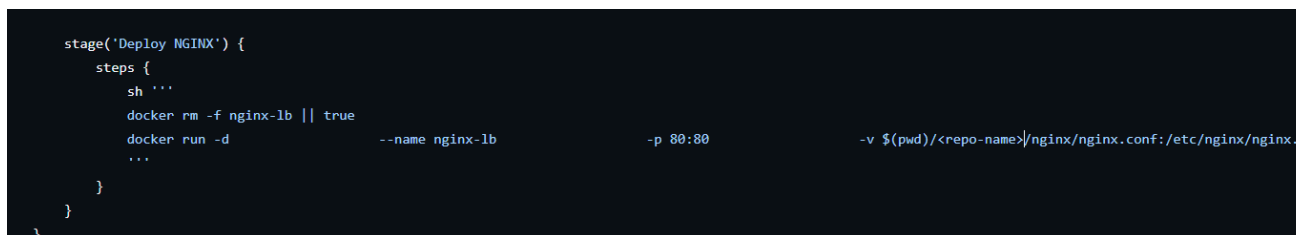
For simplicity, ensure the name in the repo is CC_LAB-6.



The screenshot shows a Jenkinsfile editor interface. At the top, the breadcrumb navigation indicates the file path: `CC_Test / CC_LAB-6 / Jenkinsfile`, with a button to switch to the `main` branch. Below the navigation are `Edit` and `Preview` tabs. The main area displays a Jenkinsfile with the following content:

```
1  pipeline {
2      agent any
3
4      stages {
5          stage('Build Backend Image') {
6              steps {
7                  sh 'docker build -t backend-app <repo-name>/backend'
8              }
9          }
10
11         stage('Deploy Backends') {
12             steps {
13                 sh '''
14                     docker rm -f backend1 backend2 || true
15                     docker run -d --name backend1 backend-app
```

(In my case it would be CC_Lab-6)



The screenshot shows a snippet of a Jenkinsfile, specifically the `stage('Deploy NGINX')` section. The code is as follows:

```
stage('Deploy NGINX') {
  steps {
    sh '''
      docker rm -f nginx-lb || true
      docker run -d          --name nginx-lb          -p 80:80          -v $(pwd)/<repo-name>/nginx/nginx.conf:/etc/nginx/nginx.
    '''
  }
}
```

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:

```
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:

```
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

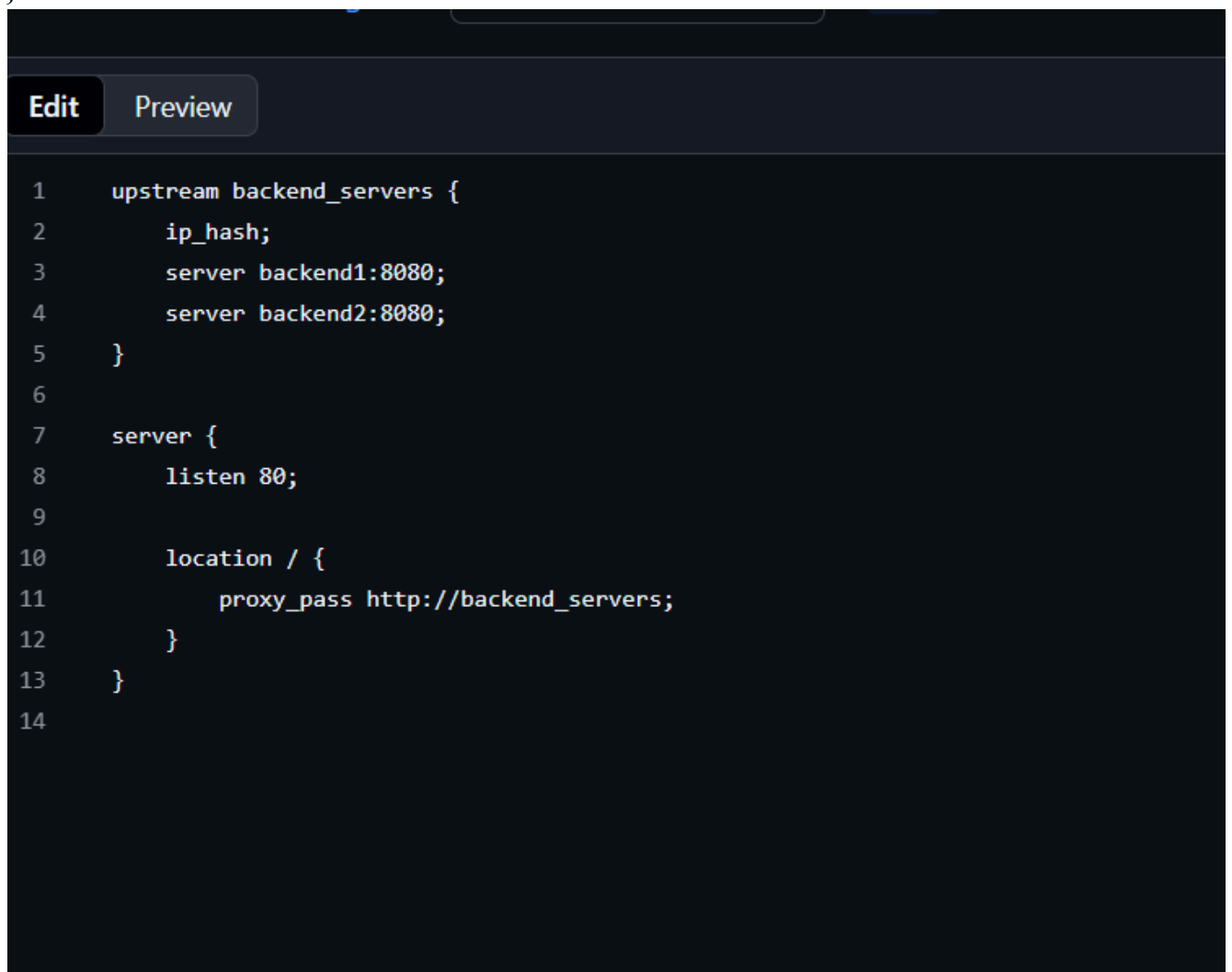
This strategy ensures requests from the same client IP always go to the same backend.

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

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

server {
    listen 80;

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

A screenshot of a code editor with a dark theme. At the top, there are two tabs: 'Edit' (active) and 'Preview'. Below the tabs, the code for 'nginx/default.conf' is displayed. The code is as follows:

```
1  upstream backend_servers {
2      ip_hash;
3      server backend1:8080;
4      server backend2:8080;
5  }
6
7  server {
8      listen 80;
9
10     location / {
11         proxy_pass http://backend_servers;
12     }
13 }
14
```

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 :)



Couple of common errors during builds

1)

```
Started by user admin
Running as SYSTEM
.
.
.
+ cd CC_LAB-6
+ docker build -t backend-app backend
ERROR: permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Head "http://%2Fvar%2Frun%2Fdocker.sock/_ping": dial unix /var/run/docker.sock: connect: permission denied
Build step 'Execute shell' marked build as failure
Finished: FAILURE
```

This is a **Docker socket permission issue**. The Jenkins container doesn't have the necessary permissions to access the Docker daemon socket.

Run this in your terminal where you created and pulled Jenkins:

docker stop jenkins

docker rm jenkins

docker stop jenkins docker rm jenkins docker run -d -p 8080:8080 -p 50000:50000

`-v jenkins_home:/var/jenkins_home` -v

/var/run/docker.sock:/var/run/docker.sock `--user root `--name jenkins

jenkins-docker

If this still doesn't work then ask Claude :)

2)

```
Started by user admin
ERROR: Unable to find Jenkinsfile from git
https://github.com/usn/CC_Lab-6.git
Finished: FAILURE
```

This means that your Jenkinsfile is not found. In the Jenkinsfile, make sure that the file path is correct.

3)

In case you face multiple file path issues, run:

docker exec -u root jenkins rm -rf

/var/jenkins_home/workspace/PES1UG2XCSXXX-backend-build

If you restarted Jenkins as root, your workspace may have been corrupted. This command deletes the corrupted workspace directory.

4)

In case you get something like this when building your pipeline, it is because NGINX is trying to reload its config **before** it can resolve the backend container names on the network.

```
Status: Downloaded newer image for nginx:latest
c9c0bb68307b09985d60f2caecfd164d1bbc4343f17040951ff7771080a486be
+ docker cp CC_LAB-6/nginx/default.conf nginx-lb:/etc/nginx/conf.d/default.conf
+ docker exec nginx-lb nginx -s reload
2026/02/15 14:24:27 [emerg] 37#37: host not found in upstream "backend1:8080" in
/etc/nginx/conf.d/default.conf:2
nginx: [emerg] host not found in upstream "backend1:8080" in
/etc/nginx/conf.d/default.conf:2
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] echo
Pipeline failed. Check console logs for errors.
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
ERROR: script returned exit code 1
Finished: FAILURE
```

The fix is to add a small delay in the Jenkins file.

- **Add sleep 3 after deploying backend containers**
- **Add sleep 2 after starting nginx but before copying config**

Make sure you commit your changes to the file.

Then click **Build Now** in Jenkins.

The pipeline should complete successfully now! After it succeeds, you can test the load balancer by opening your browser and refreshing multiple times. You should see it alternate between backend 1 and backend 2.