

**DEVOPS LAB**

**(ETCA827A)**

**Lab File**

**Submitted To:**  **Submitted By:**

Dr.Surabhi Shankar Sonali Rout (2201560012)  
Assistant Professor MCA III Semester

Dept. Of CSE

[ SOET]

**INDEX**

**List of Practical**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No.** | **Experiment Title** | **Page No.** | **Date** | **Signature** |
| **01** | Exploring Git Commands through Collaborative Coding. |  |  |  |
| **02** | Implement GitHub Operations. |  |  |  |
| **03** | Implement GitLab Operations. |  |  |  |
| **04** | Implement GitLab Operations. |  |  |  |
| **05** | Applying CI/CD Principles to Web Development Using Jenkins, Git, and Local HTTP Server. |  |  |  |
| **06** | Exploring Containerization and Application Deployment with Docker. |  |  |  |
| **07** | Applying CI/CD Principles to Web Development Using Jenkins, Git, using Docker Containers. |  |  |  |
| **08** | Demonstrate Maven Build Life Cycle. |  |  |  |
| **09** | Demonstrate Container Orchestration using Kubernets. |  |  |  |
| **10** | Create the GitHub Account to demonstrate CI/CD pipeline using Cloud Platform. |  |  |  |
|  | | | | |

**Experiment No. 1**

**Title: Exploring Git Commands through Collaborative Coding.**

**Theory:** The objective of this experiment is to familiarise participants with essential Git concepts and commands, enabling them to effectively use Git for version control and collaboration

**Introduction:** Git is a distributed version control system (VCS) that helps developers track changes in their codebase, collaborate with others, and manage different versions of their projects efficiently. It was created by Linus Torvalds in 2005 to address the shortcomings of existing version control systems.

Unlike traditional centralised VCS, where all changes are stored on a central server, Git follows a distributed model. Each developer has a complete copy of the repository on their local machine, including the entire history of the project. This decentralisation offers numerous advantages, such as offline work, faster operations, and enhanced collaboration.

Git is a widely used version control system that allows developers to collaborate on projects, track changes, and manage codebase history efficiently. This experiment aims to provide a hands-on introduction to Git and explore various fundamental Git commands. Participants will learn how to set up a Git repository, commit changes, manage branches, and collaborate with others using Git.

# Key Concepts:

* **Repository:** A Git repository is a collection of files, folders, and their historical versions. It contains all the information about the project's history, branches, and commits.
* **Commit:** A commit is a snapshot of the changes made to the files in the repository at a specific point in time. It includes a unique identifier (SHA-1 hash), a message describing the

changes, and a reference to its parent commit(s).

* **Branch:** A branch is a separate line of development within a repository. It allows developers to work on new features or bug fixes without affecting the main codebase. Branches can be

merged back into the main branch when the changes are ready.

* **Merge:** Merging is the process of combining changes from one branch into another. It integrates the changes made in a feature branch into the main branch or any other target

branch.

* **Pull Request:** In Git hosting platforms like GitHub, a pull request is a feature that allows developers to propose changes from one branch to another. It provides a platform for code

review and collaboration before merging.

* **Remote Repository:** A remote repository is a copy of the Git repository stored on a server, enabling collaboration among multiple developers. It can be hosted on platforms like GitHub,

GitLab, or Bitbucket.

# Basic Git Commands:

* **git init**: Initialises a new Git repository in the current directory.
* **git clone**: Creates a copy of a remote repository on your local machine.
* **git add**: Stages changes for commit, preparing them to be included in the next commit.
* **git commit**: Creates a new commit with the staged changes and a descriptive message.
* **git status**: Shows the current status of the working directory, including tracked and untracked files.
* **git log**: Displays a chronological list of commits in the repository, showing their commit

messages, authors, and timestamps.

* **git branch**: Lists, creates, or deletes branches within the repository.
* **git checkout**: Switches between branches, commits, or tags. It's used to navigate through the repository's history.
* **git merge**: Combines changes from different branches, integrating them into the current

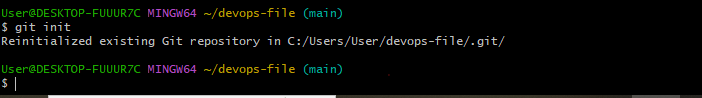
branch.

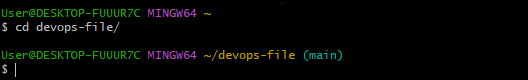
* **git pull**: Fetches changes from a remote repository and merges them into the current branch.
* **git push**: Sends local commits to a remote repository, updating it with the latest changes.

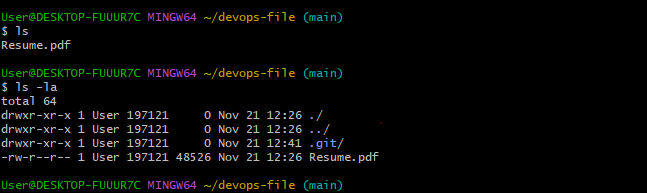
## **Software Requirements:**

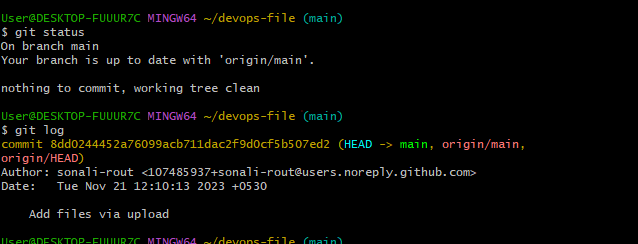
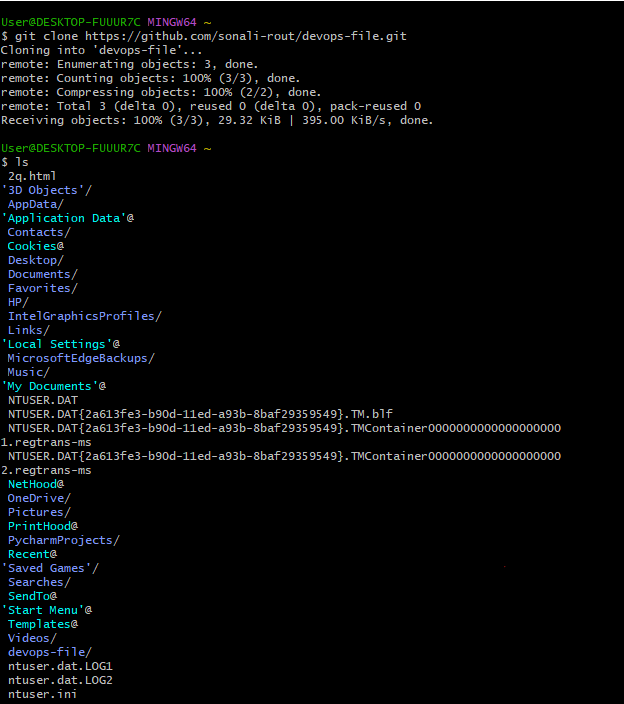
* Computer with Git installed (https://git-scm.com/downloads)
* GitHub account (https://github.com/)
* Internet connection

**Git Command Screenshot-**

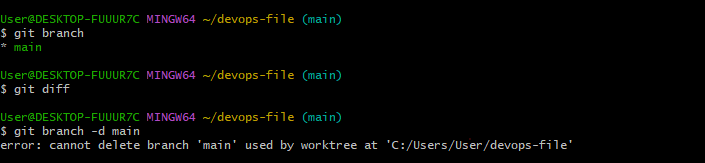
****

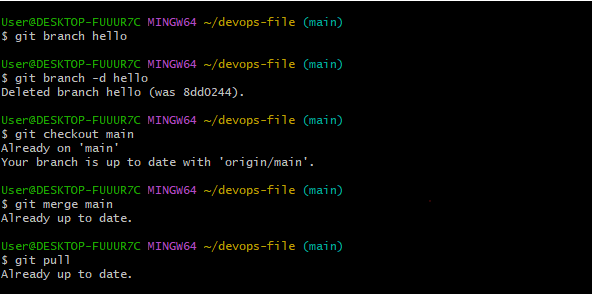
****

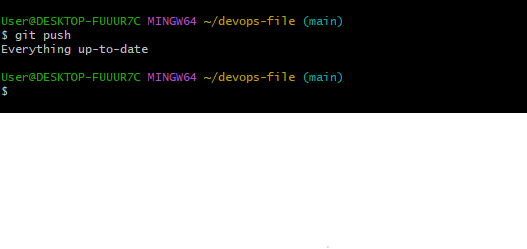
****

****

****

****

****

****

**Experiment No. 2**

**Title: Implement Git Hub Operations using Git.**

**Theory:** The objective of this experiment is to guide you through the process of using Git commands to interact with GitHub, from cloning a repository to collaborating with others through pull requests.

**Introduction:** GitHub is a web-based platform that offers version control and collaboration services for software development projects. It provides a way for developers to work together, manage code, track changes, and collaborate on projects efficiently. GitHub is built on top of the Git version control system, which allows for distributed and decentralised development.

**Key Concept:**

* **Version Control:** GitHub uses Git, a distributed version control system, to track changes to source code over time. This allows developers to collaborate on projects while maintaining a

history of changes and versions.

* **Repositories**: A repository (or repo) is a collection of files, folders, and the entire history of a project. Repositories on GitHub serve as the central place where code and project-related

assets are stored.

* **Collaboration:** GitHub provides tools for team collaboration. Developers can work together on the same project, propose changes, review code, and discuss issues within the context of

the project.

* **Pull Requests:** Pull requests (PRs) are proposals for changes to a repository. They allow developers to submit their changes for review, discuss the changes, and collaboratively

improve the code before merging it into the main codebase.

* **Issues and Projects:** GitHub allows users to track and manage project-related issues, enhancements, and bugs. Projects and boards help organize tasks, track progress, and manage

workflows.

* **Forks and Clones:** Developers can create copies (forks) of repositories to work on their own versions of a project. Cloning a repository allows developers to create a local copy of the

project on their machine.

* **Branching and Merging:** GitHub supports branching, where developers can create separate lines of development for features or bug fixes. Changes made in branches can be merged back

into the main codebase.

* **Actions and Workflows:** GitHub Actions enable developers to automate workflows, such as building, testing, and deploying applications, based on triggers like code pushes or pull

requests.

* **GitHub Pages:** This feature allows users to publish web content directly from a GitHub repository, making it easy to create websites and documentation for projects.

**Basic Git Commands:**

**Step 1: Cloning a Repository**

* Sign in to your GitHub account.
* Find a repository to clone (you can use a repository of your own or any public repository).
* Click the "Code" button and copy the repository URL.
* Open your terminal or command prompt.
* Navigate to the directory where you want to clone the repository.
* Run the following command:

### **git clone <repository\_url>**

* Replace <repository\_url> with the URL you copied from GitHub.
* This will clone the repository to your local machine.

## **Step 2: Making Changes and Creating a Branch**

Navigate into the cloned repository:

## **cd <repository\_name>**

* Create a new text file named "example.txt" using a text editor.
* Add some content to the "example.txt" file.
* Save the file and return to the command line.
* Check the status of the repository:

### **git status**

* Stage the changes for commit:

## **git add example.txt**

* Commit the changes with a descriptive message:

### **git commit -m "Add content to example.txt"**

* Create a new branch named "feature":

### **git branch feature**

* Switch to the "feature" branch:

***git checkout feature***

## **Step 3: Pushing Changes to GitHub**

* Add Repository URL in a variable

### **git remote add origin <repository\_url>**

* Replace <repository\_url> with the URL you copied from GitHub.
* Push the "feature" branch to GitHub:

### **git push origin feature**

* Check your GitHub repository to confirm that the new branch "feature" is available.

## **Step 4: Collaborating through Pull Requests**

* Create a pull request on GitHub:
* Go to the repository on GitHub.
* Click on "Pull Requests" and then "New Pull Request."
* Choose the base branch (usually "main" or "master") and the compare branch ("feature").
* Review the changes and click "Create Pull Request."
* Review and merge the pull request:
* Add a title and description for the pull request.
* Assign reviewers if needed.
* Once the pull request is approved, merge it into the base branch.

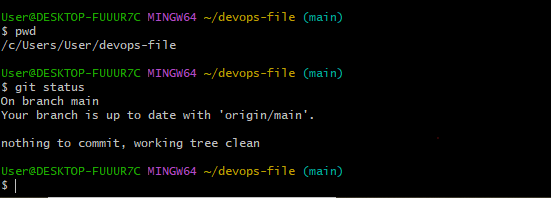
## **Step 5: Syncing Changes**

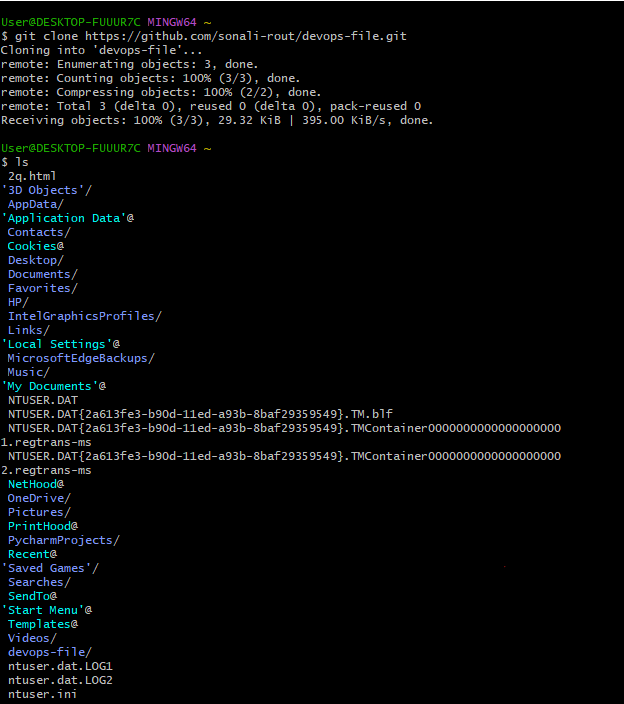
* After the pull request is merged, update your local repository:

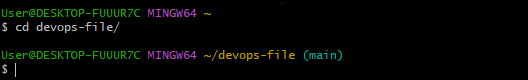
## **Software Requirements:**

* Computer with Git installed (https://git-scm.com/downloads)
* GitHub account (https://github.com/)
* Internet connection

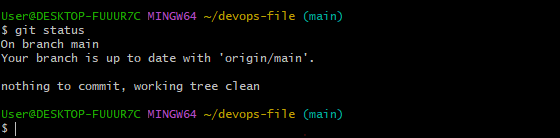
**Git Commands Screenshot:**

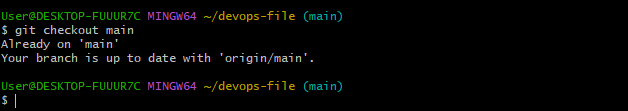
****

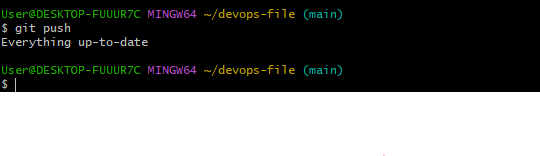
****

****

****

****

****

****

**Experiment No. 3**

**Title: Implement GitLab Operations using Git.**

**Theory:** The objective of this experiment is to guide you through the process of using Git commands to interact with GitLab, from creating a repository to collaborating with others through merge requests.

**Introduction:** GitLab is a web-based platform that offers a complete DevOps lifecycle toolset, including version control, continuous integration/continuous deployment (CI/CD), project management, code review, and collaboration features. It provides a centralized place for software development teams to work together efficiently and manage the entire development process in a single platform.

**Here some operation of GitLab-**

**Step 1:** Cloning a Repository

git clone

**Step 2:** Making Changes and Creating a Branch

Cd<repository\_name>

git status

● Stage the changes for commit:

git add example.txt

● Commit the changes with a descriptive message:

git commit -m "Add content to example.txt"

● Create a new branch named "feature":

git branch feature

● Switch to the "feature" branch:

git checkout feature

**Step 3:** Pushing Changes to GitLab

git remote add origin<repository\_url>

**Step 4:** Syncing Changes

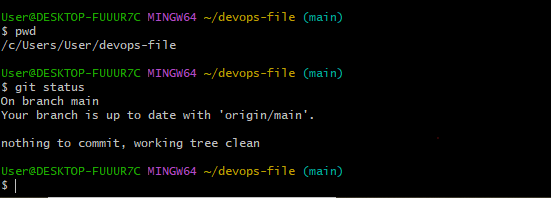
git checkout main git

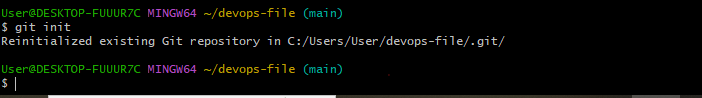
pull origin main

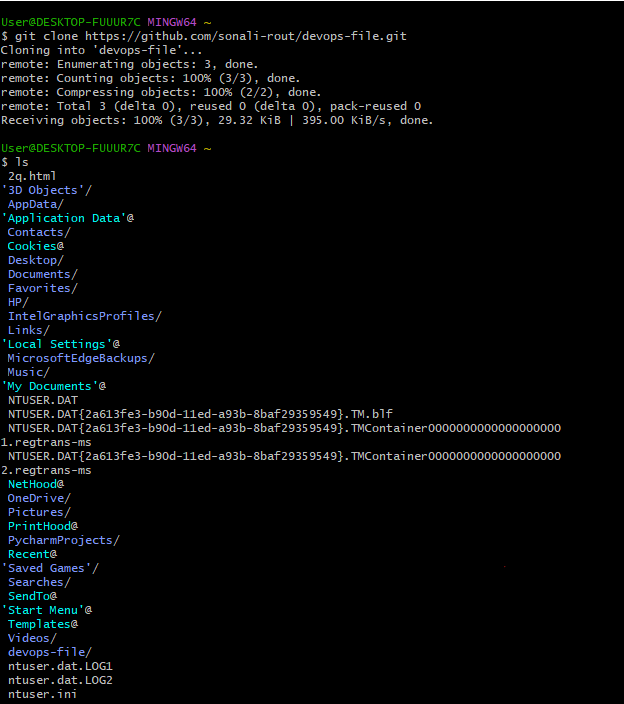
## **Software Requirements:**

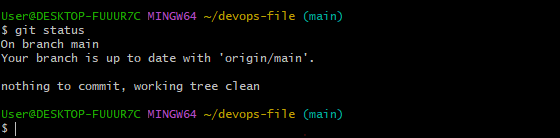
* Computer with Git installed (https://git-scm.com/downloads)
* GitLab account (https://gitlab.com/)
* Internet connection

**Screenshots:**

****

****

****

****

**Experiment No. 4**

**Title: Implement BitBucket Operations using Git.**

**Theory:** The objective of this experiment is to guide you through the process of using Git commands to interact with Bitbucket, from creating a repository to collaborating with others through pull requests.

**Introduction:** Bitbucket is a web-based platform designed to provide version control, source code management, and collaboration tools for software development projects. It is widely used by teams and individuals to track changes in code, collaborate on projects, and streamline the development process. Bitbucket offers Git and Mercurial as version control systems and provides features to support code collaboration, continuous integration/continuous deployment (CI/CD), and project management.

## **Key Features of Bitbucket:**

* **Version Control:** Bitbucket supports both Git and Mercurial version control systems, allowing developers to track changes, manage code history, and work collaboratively on projects.
* **Repositories:** In Bitbucket, a repository is a container for code, documentation, and other project assets. It houses different branches, tags, and commits that represent different versions

of the project.

* **Collaboration:** Bitbucket enables team collaboration through features like pull requests, code reviews, inline commenting, and team permissions. These tools help streamline the process of

merging code changes.

* **Pull Requests:** Pull requests in Bitbucket allow developers to propose and review code changes before they are merged into the main codebase. This process helps ensure code

quality and encourages collaboration.

* **Code Review:** Bitbucket provides tools for efficient code review, allowing team members to comment on specific lines of code and discuss changes within the context of the code itself.
* **Continuous Integration/Continuous Deployment (CI/CD):** Bitbucket integrates with CI/CD pipelines, automating processes such as building, testing, and deploying code changes to

various environments.

* **Project Management**: Bitbucket offers project boards and issue tracking to help manage tasks, track progress, and plan project milestones effectively.
* **Bitbucket Pipelines:** This feature allows teams to define and automate CI/CD pipelines directly within Bitbucket, ensuring code quality and rapid delivery.
* Access Control and Permissions: Bitbucket allows administrators to define user roles, permissions, and access control settings to ensure the security of repositories and project

assets.

**Here some Bit Bucket Operations-**

**Step 1:** Creating a Repository

**Step 2:** Cloning a Repository

git clone

**Step 3:** Making Changes and Creating a Branch

Cd<repository\_name>

**Step 4:** Pushing Changes to Bit bucket

git remote add origin

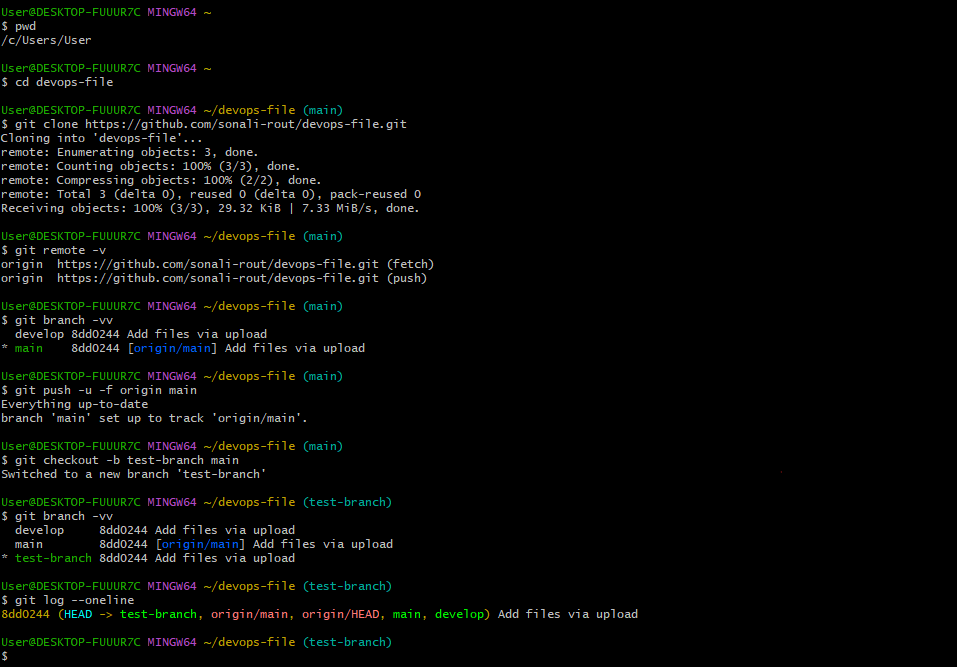
**Step 5:** Syncing Changes

● After the pull request is merged, update your local repository:

git checkout main git

pull origin main

**Screenshot:**



**Experiment No. 5**

**Title: Applying CI/CD Principles to Web Development Using Jenkins, Git, and Local HTTP Server**

**Theory:** The objective of this experiment is to set up a CI/CD pipeline for a web development project using Jenkins, Git, and web hooks, without the need for a Jenkins file. You will learn how to automatically build and deploy a web application to a local HTTP server whenever changes are pushed to the Git repository, using Jenkins' "Execute Shell" build step.

**Introduction:** Continuous Integration and Continuous Deployment (CI/CD) is a critical practice in modern software development, allowing teams to automate the building, testing, and deployment of applications. This process ensures that software updates are consistently and reliably delivered to end-users, leading to improved development efficiency and product quality.

## **Key Components:**

* **Jenkins:** Jenkins is a widely used open-source automation server that helps automate various aspects of the software development process. It is known for its flexibility and extensibility

and can be employed to create CI/CD pipelines.

* **Git:** Git is a distributed version control system used to manage and track changes in source code. It plays a crucial role in CI/CD by allowing developers to collaborate, track changes,

and trigger automation processes when code changes are pushed to a repository.

* **Local HTTP Server:** A local HTTP server is used to host and serve web applications during development. It is where your web application can be tested before being deployed to

production servers.

## **CI/CD Principles:**

* **Continuous Integration (CI):** CI focuses on automating the process of integrating code changes into a shared repository frequently. It involves building and testing the application

each time code is pushed to the repository to identify and address issues early in the development cycle.

* **Continuous Deployment (CD):** CD is the practice of automatically deploying code changes to production or staging environments after successful testing. CD aims to minimize manual intervention and reduce the time between code development and its availability to end-users.

## **The CI/CD Workflow:**

* **Code Changes:** Developers make changes to the web application's source code locally.
* **Git Repository:** Developers push their code changes to a Git repository, such as GitHub or Bitbucket.
* **Webhook:** A webhook is configured in the Git repository to notify Jenkins whenever changes are pushed.
* **Jenkins Job:** Jenkins is set up to listen for webhook triggers. When a trigger occurs, Jenkins initiates a CI/CD pipeline.
* **Build and Test:** Jenkins executes a series of predefined steps, which may include building the application, running tests, and generating artifacts.
* **Deployment:** If all previous steps are successful, Jenkins deploys the application to a local HTTP server for testing.
* **Verification:** The deployed application is tested locally to ensure it functions as expected.
* **Optional Staging:** For more complex setups, there might be a staging environment where the application undergoes further testing before reaching production.
* **Production Deployment:** If the application passes all tests, it can be deployed to the production server

**Experiment Steps:**

**Step 1:** Set Up the Web Application and Local HTTP Server

**Step 2:** Set Up a Git Repository

git init git

add .

git commit -m "Initial commit"

**Step 4:** Create a Jenkins Job

**Step 5:** Set Up the Jenkins Job (Using Execute Shell)

**Step 6:** Set Up a Web hook in Git Repository

**Step 7:** Trigger the CI/CD Pipeline

**Step 8:** Verify the CI/CD Pipeline

**Experiment No. 6**

**Title: Exploring Containerization and Application Deployment with Dockers**

**Theory:** The objective of this experiment is to provide hands-on experience with Docker containerization and application deployment by deploying an Apache web server in a Docker container. By the end of this experiment, you will understand the basics of Docker, how to create Docker containers, and how to deploy a simple web server application.

**Introduction:** Containerization is a technology that has revolutionised the way applications are developed, deployed, and managed in the modern IT landscape. It provides a standardised and efficient way to package, distribute, and run software applications and their dependencies in isolated environments called containers.

## **Key Concepts of Containerization:**

* **Containers:** Containers are lightweight, stand-alone executable packages that include everything needed to run a piece of software, including the code, runtime, system tools,

libraries, and settings. Containers ensure that an application runs consistently and reliably across different environments, from a developer's laptop to a production server.

* **Images:** Container images are the templates for creating containers. They are read-only and contain all the necessary files and configurations to run an application. Images are typically built from a set of instructions defined in a Dockerfile.
* **Docker:** Docker is a popular containerization platform that simplifies the creation, distribution,

and management of containers. It provides tools and services for building,

running, and orchestrating containers at scale.

* **Isolation:** Containers provide process and filesystem isolation, ensuring that applications and their dependencies do not interfere with each other. This isolation enhances security and

allows multiple containers to run on the same host without conflicts.

**Experiment Steps:**

**Step 1:** Install Docker

**Step 2:** Create a Simple HTML Page

**Step 3:** Create a Dockerfile

Docker file

**Step 4:** Build the Docker Image

docker build -t my-apache-server

**Step 5:** Run the Docker Container

docker run -p 8080:80 -d my-apache-server

**Step 6:** Access Your Apache Web Server

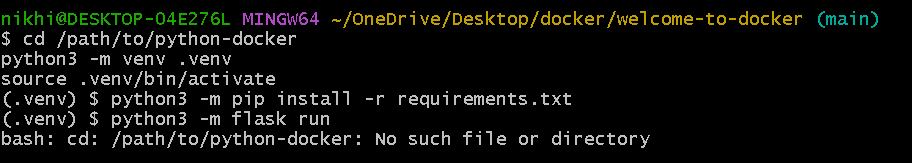
**Step 7:** Cleanup

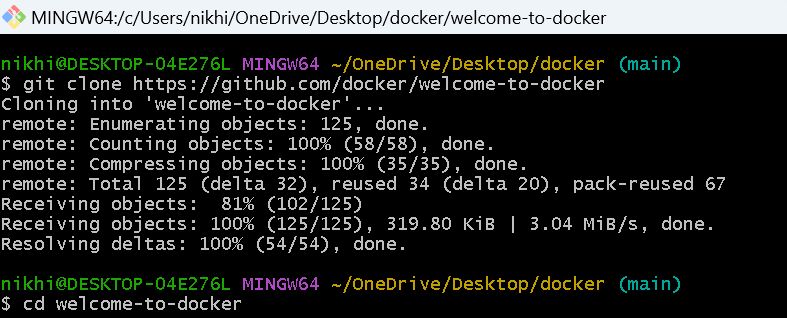
docker stop

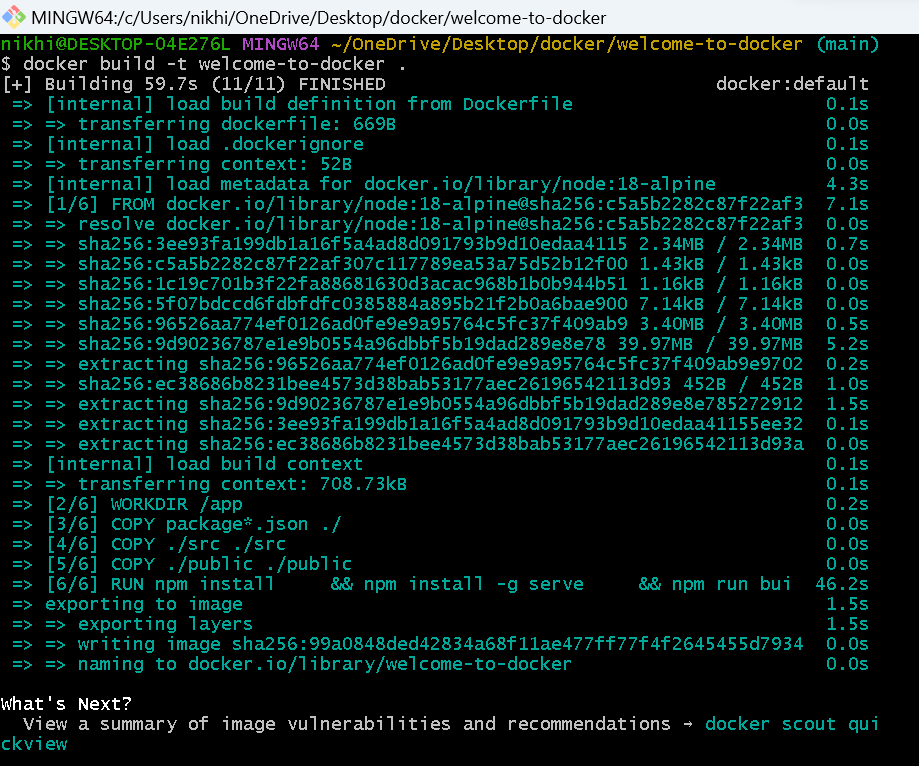
docker rm docker

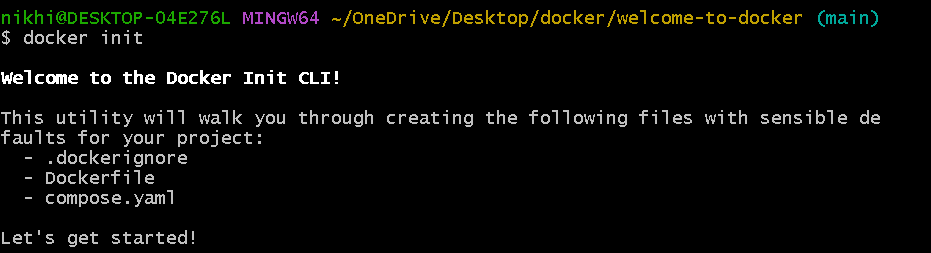
rmi my-apache-server

**Screenshot:**

****

****

****

****

**Experiment No. 7**

**Title: Applying CI/CD Principles to Web Development Using Jenkins, Git, using Docker Containers**

**Theory:** The objective of this experiment is to set up a CI/CD pipeline for a web application using Jenkins, Git, Docker containers, and GitHub webhooks. The pipeline will automatically build, test, and deploy the web application whenever changes are pushed to the Git repository, without the need for a pipeline script.

**Introduction:** Continuous Integration and Continuous Deployment (CI/CD) principles are integral to modern web development practices, allowing for the automation of code integration, testing, and deployment. This experiment demonstrates how to implement CI/CD for web development using Jenkins, Git, Docker containers, and GitHub webhooks without a pipeline script. Instead, we'll utilize Jenkins' "GitHub hook trigger for GITScm polling" feature.

**Continuous Integration (CI):**

CI is the practice of frequently and automatically integrating code changes from multiple contributors into a shared repository. The core idea is that developers regularly merge their code into a central repository, triggering automated builds and tests. Key aspects of CI include:

**Automation:** CI tools, like Jenkins, Travis CI, or CircleCI, automate the building and testing of code whenever changes are pushed to the repository.

● **Frequent Integration:** Developers commit and integrate their code changes multiple times a day, reducing integration conflicts and catching bugs early.

**Continuous Deployment (CD):**

CD is the natural extension of CI. It is the practice of automatically and continuously deploying code changes to production or staging environments after successful integration and testing. Key aspects of CD include:

● **Automation:** CD pipelines automate the deployment process, reducing the risk of human error and ensuring consistent deployments.

**● Deployment to Staging:** Code changes are deployed first to a staging environment where further testing and validation occur.

**Benefits of CI/CD in Web Development:**

● **Rapid Development:** CI/CD accelerates development cycles by automating timeconsuming tasks, allowing developers to focus on coding.

**Quality Assurance:** Automated testing ensures code quality, reducing the number of bugs and regressions.

● **Consistency:** CI/CD ensures that code is built, tested, and deployed consistently, regardless of the development environment.

**Experiment Steps:**

**Step 1:** Set Up the Web Application and Git Repository

**Step 2:** Install and Configure Jenkins

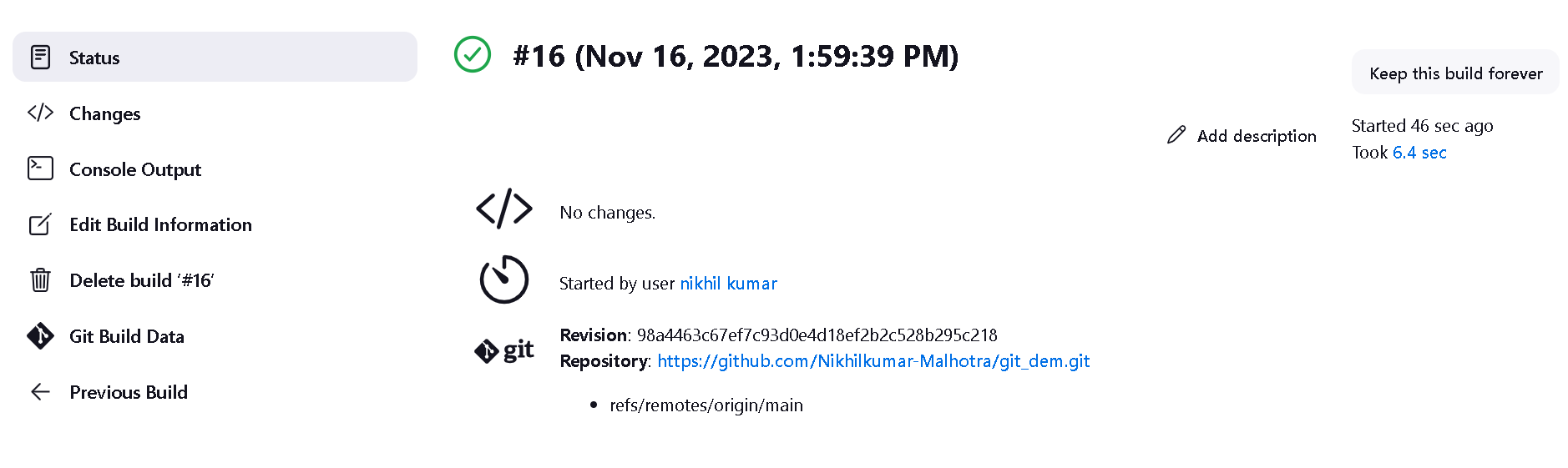
**Step 3:** Create a Jenkins Job

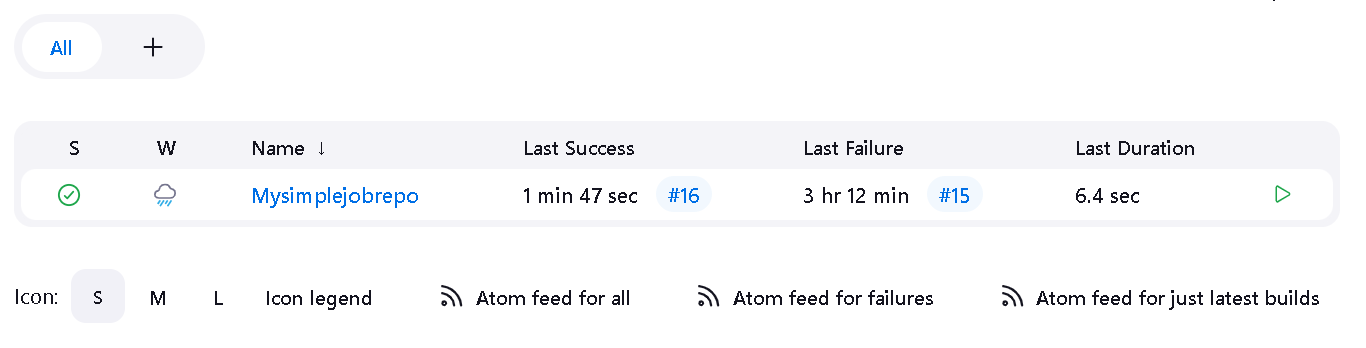
**Step 4:** Configure Build Steps

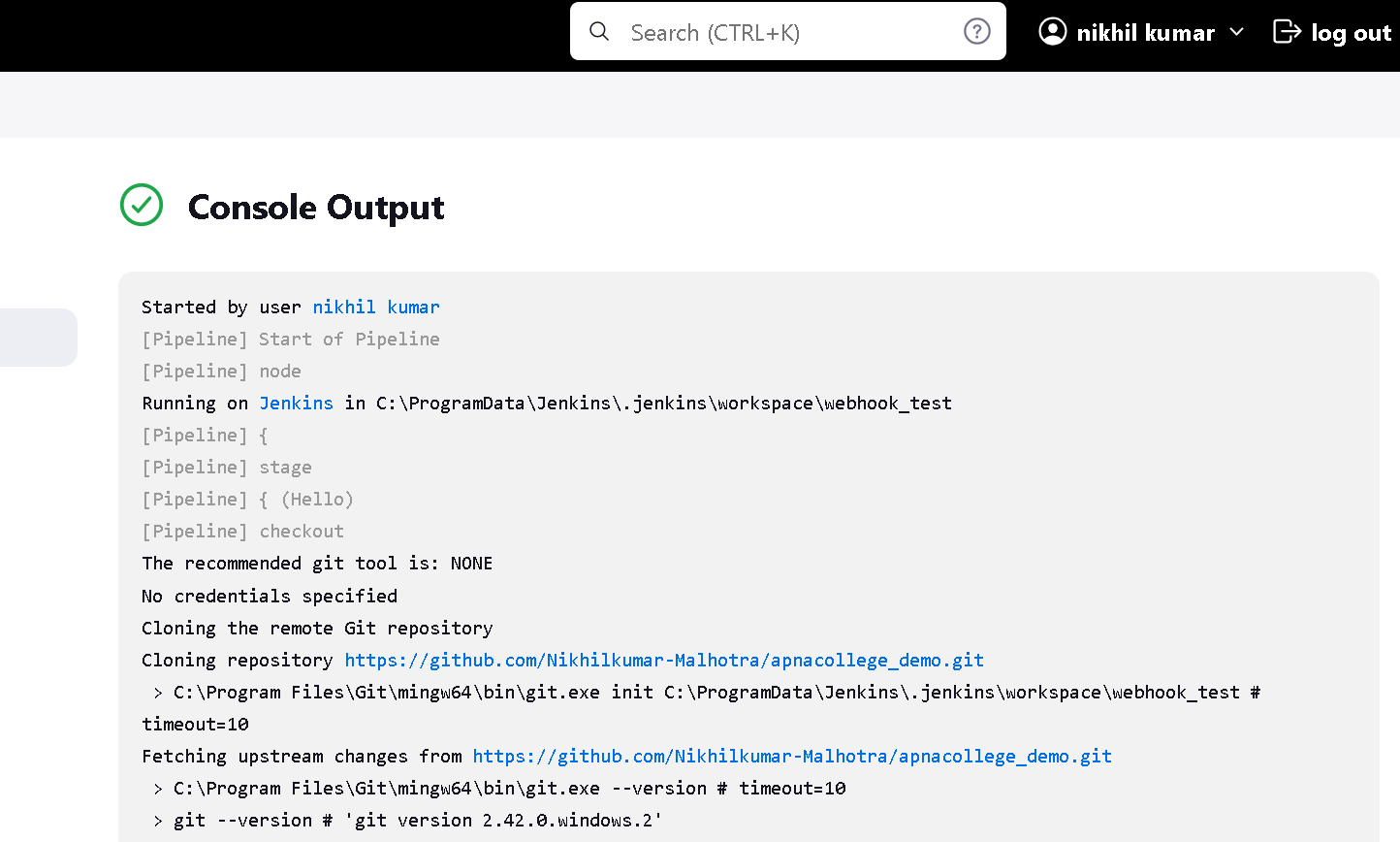
**Step 5:** Set Up a GitHub Webhook

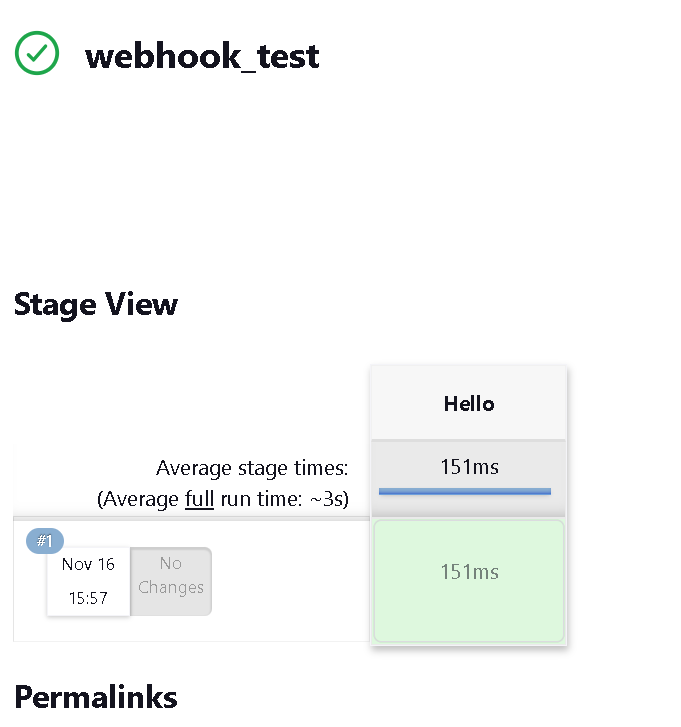
**Step 6:** Trigger the CI/CD Pipeline

**Step 7:** Verify the Deployment









**Experiment No. 8**

**Title: Demonstrate Maven Build Life Cycle**

**Theory:** The objective of this experiment is to gain hands-on experience with the Maven build lifecycle by creating a simple Java project and executing various Maven build phases.

**Introduction**: Maven is a widely-used build automation and project management tool in the Java ecosystem. It provides a clear and standardised build lifecycle for Java projects, allowing developers to perform various tasks such as compiling code, running tests, packaging applications, and deploying artefacts. This experiment aims to demonstrate the Maven build lifecycle and its different phases.

**Maven Build Life Cycle:**

The Maven build process is organised into a set of build lifecycles, each comprising a sequence of phases. Here are the key build lifecycles and their associated phases:

**Clean Lifecycle:**

● **clean:** Deletes the target directory, removing all build artifacts. Default Lifecycle:

● **validate:** Validates the project's structure.

**compile:** Compiles the project's source code.

● **test:** Runs tests using a suitable testing framework.

● **package:** Packages the compiled code into a distributable format (e.g., JAR, WAR).

● **verify:** Runs checks on the package to verify its correctness.

● **install:** Installs the package to the local repository.

● **deploy:** Copies the final package to a remote repository for sharing.

**Experiment Steps:**

**Step 1:** Setup Maven and Java

**Step 2:** Create a Maven Java Project

**Step 3:** Explore the Maven Build Phases

**Step 4:** Run the Application

**Here are some of the most important phases in the default build lifecycle:**

* **validate**: check if all information necessary for the build is available
* **compile**: compile the source code
* **test**-**compile**: compile the test source code
* **test**: run unit tests
* **package**: package compiled source code into the distributable format (jar, war, …)
* **integration-test:** process and deploy the package if needed to run integration tests
* **install:** install the package to a local repository
* **deploy:** copy the package to the remote repository

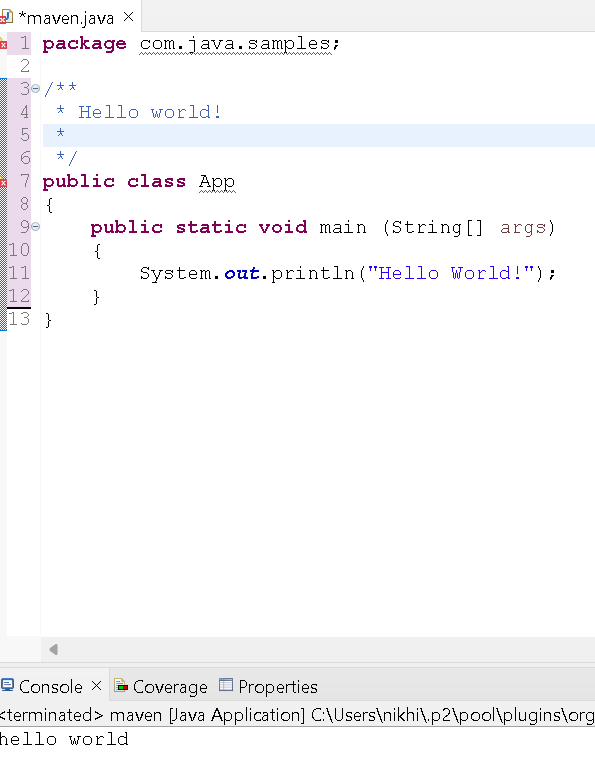
Phases are executed in a specific order. This means that if we run a specific phase using the command:

The Maven Build Phases

mvn <PHASE>

if we run the deploy phase

mvn deploy



**Experiment No. 9**

**Title: Demonstrating Container Orchestration using Kubernetes**

**Theory:** The objective of this experiment is to introduce students to container orchestration using Kubernetes and demonstrate how to deploy a containerized web application. By the end of this experiment, students will have a basic understanding of Kubernetes concepts and how to use Kubernetes to manage containers.

**Introduction:** Container orchestration is a critical component in modern application deployment, allowing you to manage, scale, and maintain containerized applications efficiently. Kubernetes is a popular container orchestration platform that automates many tasks associated with deploying, scaling, and managing containerized applications. This experiment will demonstrate basic container orchestration using Kubernetes by deploying a simple web application.

**Experiment Steps:**

**Step 1:** Create a Dockerized Web Application

**Step 2:** Deploy the Web Application with Kubernetes

**Step 3:** Deploy the Application

kubectl apply -f web-app-deployment.yaml

**Step 4:** Verify the Deployment

kubectl get pods

## **Key Concepts in Kubernetes:**

**Containerization:** Kubernetes relies on containers as the fundamental unit for packaging and running applications. Containers encapsulate an application and its dependencies, ensuring consistency across various environments.

**Cluster:** A Kubernetes cluster is a set of machines, known as nodes, that collectively run containerized applications. A cluster typically consists of a master node (for control and management) and multiple worker nodes (for running containers).

**Nodes:** Nodes are individual machines (virtual or physical) that form part of a Kubernetes cluster. Nodes run containerized workloads and communicate with the master node to manage and orchestrate containers.

**Pod:** A pod is the smallest deployable unit in Kubernetes. It can contain one or more tightly coupled containers that share the same network and storage namespace. Containers within a pod are typically used to run closely related processes.

**Deployment:** A Deployment is a Kubernetes resource that defines how to create, update, and scale instances of an application. It ensures that a specified number of replicas are running at all times.

**Service:** A Service is an abstraction that exposes a set of pods as a network service. It provides a stable IP address and DNS name for accessing the pods, enabling load balancing and discovery.

**Namespace:** Kubernetes supports multiple virtual clusters within the same physical cluster, called namespaces. Namespaces help isolate resources and provide a scope for organizing and managing workloads.

## **Experiment Steps:**

**Step 1: Create a Dockerized Web Application**

* + Create a simple web application (e.g., a static HTML page) or use an existing one.
  + Create a Dockerfile to package the web application into a Docker container. Here's an example Dockerfile for a simple web server:

### # Use an official Nginx base image FROM nginx:latest

***#*** Copy the web application files to the Nginx document root COPY

### ./webapp /usr/share/nginx/html

* + Build the Docker image:

docker build -t my-web-app .

# Step 2: Deploy the Web Application with Kubernetes

Create a Kubernetes Deployment YAML file (web-app-deployment.yaml) to deploy the web application:

### apiVersion: apps/v1 kind:

Deployment metadata:

### name: my-web-app-deployment spec:

replicas***: 3*** # Number of pods to create selector:

### matchLabels:

app: my-web-app # Label to match pods template:

### metadata:

labels***:***

### app: my-web-app # Label assigned to pods spec:

containers***:***

### - name: my-web-app-container

image: my-web-app:latest # Docker image to use ports:

- containerPort: 80 # Port to expose

## **Step 3: Deploy the Application**

* + Apply the deployment configuration to your Kubernetes cluster:

kubectl apply -f web-app-deployment.yaml

## **Step 4: Verify the Deployment**

* + Check the status of your pods:

kubectl get pods

**Experiment No. 10**

**Title: Create the GitHub Account to demonstrate CI/CD pipeline using Cloud Platform.**

**Theory:** The objective of this experiment is to help you create a GitHub account and set up a basic CI/CD pipeline on GCP. You will learn how to connect your GitHub repository to GCP, configure CI/CD using Cloud Build, and automatically deploy web pages to an Apache web server when code is pushed to your repository.

**Introduction:** Continuous Integration and Continuous Deployment (CI/CD) pipelines are essential for automating the deployment of web applications. In this experiment, we will guide you through creating a GitHub account and setting up a basic CI/CD pipeline using Google Cloud Platform (GCP) to copy web pages for an Apache HTTP web application.

## **Basic CI/CD Workflow:**

A basic CI/CD workflow using GitHub, GCP, and AWS typically includes the following steps:

Code Development**:** Developers work on code changes and commit them to a GitHub repository.

## Continuous Integration (CI):

* 1. GitHub Actions or a CI tool like Jenkins is used to automatically build, test, and package the application whenever code changes are pushed to the repository.
  2. Automated tests are executed to ensure code quality.

## Continuous Deployment (CD):

* 1. Once code changes pass CI, the application can be automatically deployed to a staging environment.
  2. Integration and acceptance tests are performed in the staging environment.

## Deployment to Production:

* 1. If all tests in the staging environment pass, the application can be automatically deployed to the production environment in GCP or AWS.

## Monitoring and Logging:

* 1. Monitoring tools are used to track the application's performance and detect issues.
  2. Logging and analytics tools are used to gain insights into application behavior.

## Feedback Loop:

* 1. Any issues or failures detected in production are reported back to the development team for further improvements.
  2. The cycle repeats as new code changes are developed and deployed.

Experiment Steps:

**Step 1:** Create a GitHub Account

**Step 2:** Create a Sample GitHub Repository

**Step 3:** Set Up a Google Cloud Platform Project

**Step 4:** Connect GitHub to Google Cloud Build

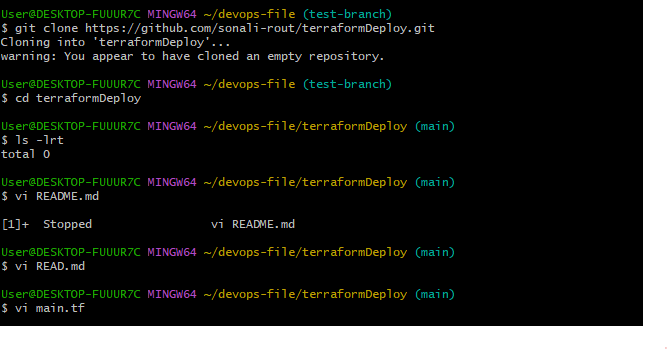
**Step 5:** Create a CI/CD Configuration File

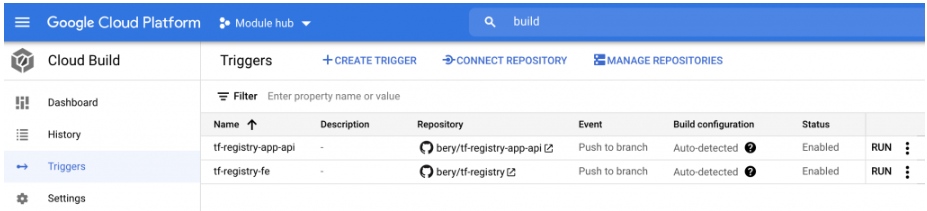
**Step 6:** Trigger the CI/CD Pipeline

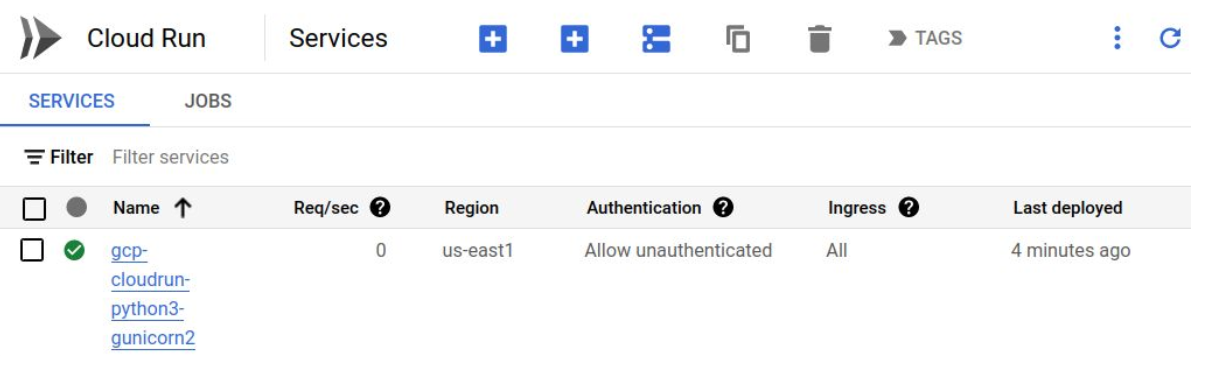
**Step 7:** Monitor the CI/CD Pipeline

**Step 8:** Access Your Deployed Web Pages

Screenshot:



****

****