CI/CD with Jenkins

Siddhesh Prabhugaonkar
Microsoft Certified Trainer
siddheshpg@azureauthority.in

Module Overview

1. Fundamentals of CI/CD
2. Jenkins basics
3. Jenkins + GitHub integration
4. Freestyle Jobs vs Pipeline Jobs
5. Writing Jenkins pipelines using Groovy
6. CI pipeline for NestJS 11 + oracledb
7. CI pipeline for React app
8. Handling public vs private GitHub repos
9. Preparing apps for Podman containerization in the next module
10. Lab setup and exercises for both Windows and macOS

SECTION 1 — Introduction to CI/CD

1.1 What is CI/CD?

Continuous Integration (CI)

A development practice where:

- Developers push code frequently to a shared repository (GitHub).
- Every push triggers **automated build + test** using Jenkins.
- Ensures early detection of defects.

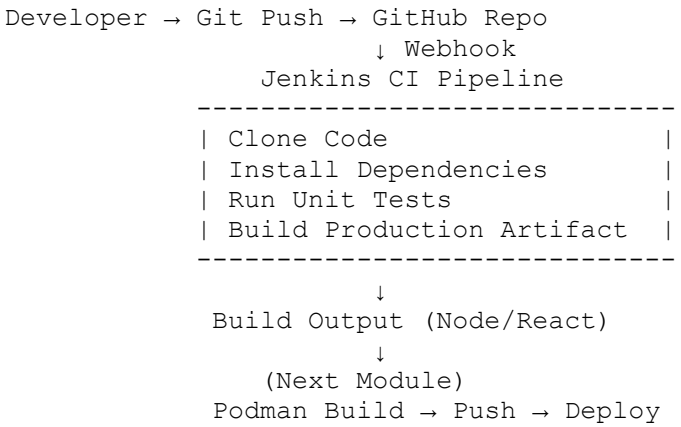
Continuous Delivery (CD)

- After CI completes, the system automatically packages the application (build artifacts).
- Deployment may be manual or automated.

Continuous Deployment

- Code goes **automatically to production** after tests pass.
- No manual approval.

1.2 CI/CD Workflow

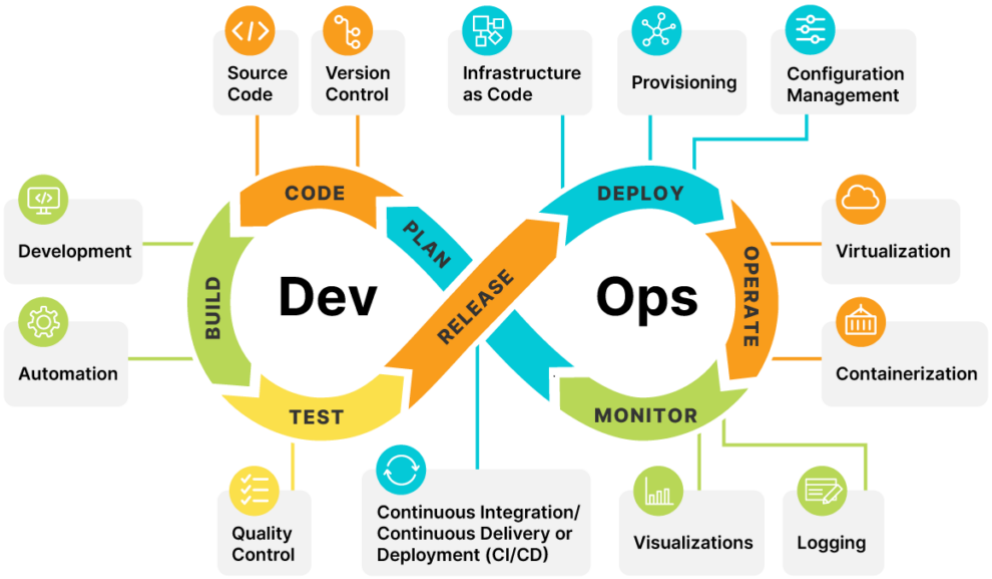


1.3 Benefits in Real-World Development

- Faster delivery
- Better quality through automated testing
- Consistency across environments
- Easy team collaboration
- Immediate feedback
- No manual repetitive tasks

1.4 CI vs CD vs DevOps

Term	Meaning	Scope
CI	Automating build + tests	Developer workflow
CD	Automated delivery to deployment-ready state	Release engineering
DevOps	Culture + tools enabling automation	Organization-wide



SECTION 2 — Jenkins Basics

2.1 Installing Jenkins (Windows & macOS)

Windows

1. Install Java 17 (Temurin recommended).
2. Download Jenkins LTS `.msi` installer.
3. Install service → Jenkins runs on `http://localhost:8080`.
4. Unlock Jenkins using initial admin password.
5. Install recommended plugins.

macOS

1. Install Java 17 using Homebrew:
2. `brew install --cask temurin@17`
3. Install Jenkins:
4. `brew install jenkins-lts`
5. `brew services start jenkins-lts`
6. Open browser:
   `http://localhost:8080`
7. Unlock and complete setup wizard.

Jenkins is an open-source automation server that facilitates CI/CD.

Key Benefits:

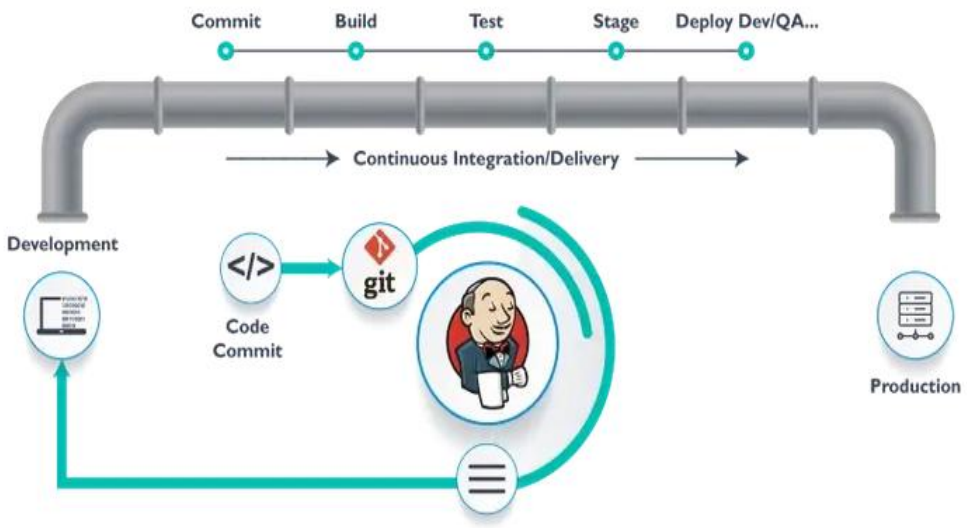
- - Automates repetitive tasks
- - Provides continuous feedback
- - Supports a vast ecosystem of plugins
- - Easily scalable

Jenkins automates:

- CI build
- Unit testing
- Image creation (via Podman)
- Deployment steps

Before Jenkins:
❌ Manual builds → "It works on my machine!" syndrome.
❌ No standardized testing → Bugs in production.
❌ Slow release cycles → Weeks to deploy.

After Jenkins:
✅ **Automated builds** → Code integration in minutes.
✅ **Instant feedback** → Tests run on every commit.
✅ **Faster deployments** → CI/CD pipelines.

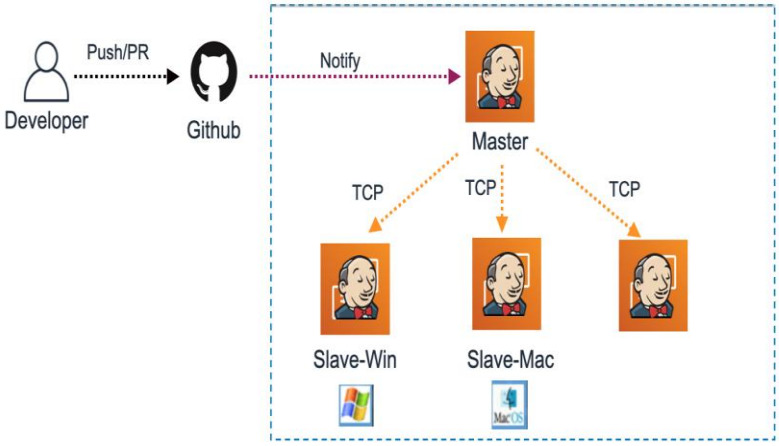


Jenkins Architecture

Jenkins follows a Master-Slave architecture

- Jenkins Master:
  - - Manages build jobs
  - - Provides a web-based dashboard
  - - Distributes build tasks
- Jenkins Slave:
  - - Executes build jobs

- - Supports parallel execution
- - Can run on different OS environments



2.2 Jenkins UI Overview

- Dashboard
- Manage Jenkins
- Credentials
- New Item (to create jobs)
- Build Queue
- Console Output
- Plugins

2.3 Jenkins Plugins Needed

Mandatory

Plugin	Purpose
Git	Clone GitHub repositories
Pipeline	Enables Jenkinsfile support
GitHub Integration	Webhooks
NodeJS Plugin	(Optional) Install Node within Jenkins agents

SECTION 3 — Jenkins with GitHub

3.1 Public vs Private Repositories

Public Repo

- Jenkins can clone without credentials (read-only).
- Webhook still requires Jenkins server to be reachable (if local, use ngrok).

Private Repo

- Personal Access Token (PAT) must be added in Jenkins:
- `Credentials → Global → Add credentials → Username + PAT`
- Jenkinsfile must reference the credential ID.

3.2 GitHub Webhooks Setup

What is a GitHub Webhook?

A GitHub Webhook is an automated notification mechanism that **sends real-time events** from GitHub to an external service (in this case, Jenkins).

Whenever an event occurs in a repository—such as:

- Push to a branch
- Pull request activity
- Tag creation
- Release creation

GitHub sends an **HTTP POST request** to a specified URL along with a JSON payload describing the event.

Purpose in CI/CD

- Automatically trigger Jenkins pipelines whenever code changes.
- Remove the need for manual clicking of "Build Now".
- Ensures CI runs **instantly** after every push.

How it works

```
Developer Pushes Code → GitHub → Sends Webhook → Jenkins Receives Event
                                        ↓                  ↓
                                  Webhook URL        Starts Pipeline
```

Key components

- **Payload URL:** URL of your Jenkins webhook listener
- `http://<jenkins-host>/github-webhook/`
- **Content-Type:** `application/json`

- **Trigger Type:** Usually "Just the push event"
- **Secret (optional):** Token to validate authenticity

GitHub Personal Access Token (PAT)

What is a GitHub PAT?

A GitHub Personal Access Token (PAT) is a **secure, revocable alternative to using passwords** when authenticating GitHub from tools like:

- Jenkins
- VS Code
- CLI
- Scripts
- CI/CD systems

A PAT is required when Jenkins needs to access a **private GitHub repository**.

Why is PAT needed?

- GitHub disabled password-based authentication for Git operations.
- PATs allow secure **read/write access** to repositories.
- Jenkins cannot clone private repos without PAT stored in credentials.

How to Create a GitHub PAT

Step-by-step

1. Log into GitHub
2. Go to:
   Profile → Settings → Developer settings → Personal access tokens → Tokens (classic)
3. Click **Generate new token**
4. Choose token type: **Classic token** (commonly used in Jenkins)
5. Provide a name, e.g., `jenkins-access-token`
6. Set expiration (recommended: 90 days or custom)
7. Select scopes:

For Jenkins cloning private repos, the minimum is:

- `repo` (full repo access)
- `read:packages` (optional)

8. Click **Generate token**
9. Copy the token once — GitHub will not show it again.
10. Add PAT to Jenkins:

```
Manage Jenkins → Credentials → System → Global → Add Credentials
Kind: Username with password
Username: <your github username>
Password: <PAT token>
ID: github-pat
```

GitHub Webhooks Setup

1. Open repository → Settings → Webhooks
2. Payload URL:
3. http://<jenkins-url>/github-webhook/
4. Content type: application/json
5. Trigger: "Just the push event"
6. Save

Jenkins Jobs

Jenkins Jobs define automation tasks such as builds, tests, and deployments.

Types of Jenkins Jobs:

- **Freestyle Jobs:** Simple jobs configured via the Jenkins UI.
- **Pipeline Jobs:** Uses Jenkinsfile to define CI/CD workflows.
- **Multi-Configuration Jobs:** Supports parameterized builds with different configurations.
- **Multibranch Pipeline Jobs:** Automatically creates jobs for different Git branches.

Freestyle Project vs Pipeline

Feature	Freestyle Project	Pipeline
Configuration	UI-based	Code-based (Jenkinsfile)
Flexibility	Limited	Highly customizable
Scalability	Moderate	High
Use Case	Simple builds & jobs	Complex CI/CD workflows
Example	Basic build job for compiling & testing	CI/CD pipeline automating build, test, deploy

When to Use Which?

- **Freestyle Job** is suitable for small projects requiring minimal customization.
- **Pipeline Job** is ideal for complex workflows, automation, and scalability.

3.3 Creating a Freestyle Job

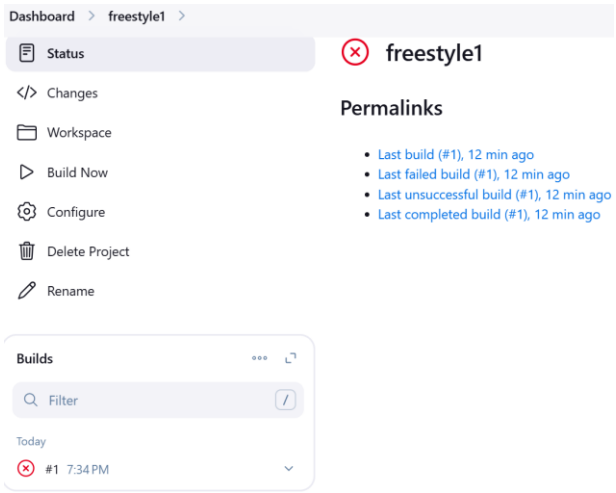
Steps:

1. Jenkins Dashboard → New Item → Freestyle
2. Source Code Management → Git
3. Enter Repo URL
4. If private, add credentials
5. Build Steps:
    - Execute shell (macOS/Linux)
    - Execute batch (Windows)

Example Build Command

```
npm install
npm run test
npm run build
```

Lab: Create a Freestyle Job



- Dashboard => New Item
- Enter an item name = ***freestyle1***
- Item type = **Freestyle project**
- scroll down to the **Build Steps**
- click **Add build step**
- select **Execute Shell**
- Command = echo “Hello Jenkins”
- click **Save**
- Click **Build Now**
- Verify
    - Click “build Number”
    - Console Output

3.4 Creating a Pipeline Job

1. New Item → Pipeline
2. Choose:
    - Pipeline Script (inline)
    - Pipeline Script from SCM (Jenkinsfile stored in Git)

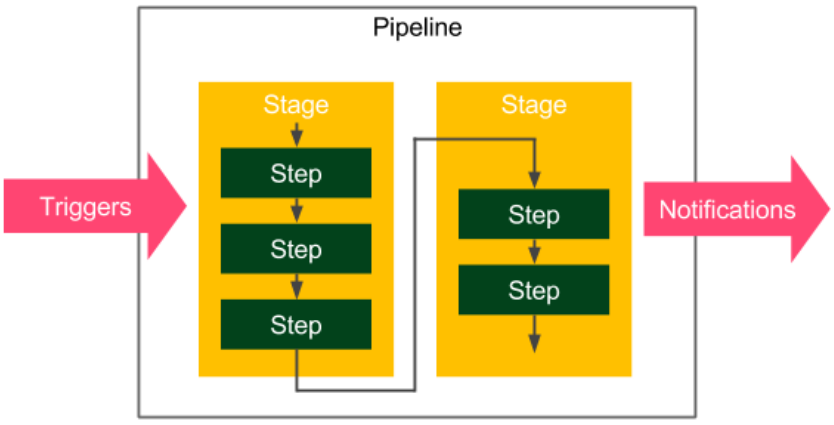
Recommended: always store Jenkinsfile in repo.

SECTION 4 — Freestyle vs Pipeline

Feature	Freestyle Job	Pipeline Job
Tooling	UI-driven	Code-driven
Scalability	Poor	Excellent
Reusability	Low	High
Complex workflows	Hard	Easy
Version-controlled	No	Yes (Jenkinsfile)
Parallel stages	No	Yes
Environment mgmt	Basic	Advanced

Conclusion:
Pipeline is the recommended approach for NestJS + React CI.

SECTION 5 —Jenkins Pipeline



- **Agent:** Slave on which jenkins pipeline is executed.
- **Stage:** A stage is a block in which tasks are performed through a jenkins pipeline.
- **Stages:** Block that includes multiple stage in a pipeline.
- **Steps:** All the tasks performed in any stage.

Jenkins pipelines use **Declarative** or **Scripted** Groovy.

Declarative Pipeline Example

```
pipeline {
    agent any
    stages {
        stage('Install') {
            steps {
                sh 'npm install'
            }
        }
        stage('Test') {
            steps {
                sh 'npm test'
            }
        }
    }
}
```

Scripted Pipeline Example

```
node {
    stage('Install') {
        sh 'npm install'
    }
}
```

Groovy basics used in Jenkins:

- Variables
- Functions
- Closures
- Maps / Lists
- String interpolation: `"Hello ${name}"`
- Conditions:
- `if (env.BRANCH_NAME == "main") { ... }`

Lab: create pipeline job

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building the application...'
            }
        }
        stage('Test') {
            steps {
                echo 'Running tests...'
            }
        }
```

```
        stage('Deploy') {
            steps {
                echo 'Deploying application...'
            }
        }
    }
}
```

SECTION 6 — CI for Node Backend (NestJS 11 + oracledb)

What Jenkins should do:

1. Clone GitHub repo
2. Install Node modules
3. Install `oracledb` dependencies
4. Run unit tests
5. Build production build
6. Archive artifacts (for next module)

6.1 Sample Jenkinsfile (Declarative)

For Public GitHub Repo

```
pipeline {
    agent any

    tools {
        nodejs "NodeJS17"
    }

    stages {
        stage('Checkout') {
            steps {
                git url: 'https://github.com/my-org/nestjs-app.git',
branch: 'main'
            }
        }

        stage('Install Dependencies') {
            steps {
                sh 'npm install'
            }
        }

        stage('Test') {
            steps {
                sh 'npm run test'
```

```
            }
        }

        stage('Build') {
            steps {
                sh 'npm run build'
            }
        }

        stage('Archive Build') {
            steps {
                archiveArtifacts artifacts: 'dist/**', fingerprint: true
            }
        }
    }
}
```

Since Jenkins is running locally on your machine, all artifacts generated during a pipeline are stored **inside Jenkins' workspace** on your system.

Location of Jenkins Workspace (Hands-on Important)

Default Path (Windows)

```
C:\ProgramData\Jenkins\.jenkins\workspace\<job-name>\
```

Default Path (macOS – Homebrew installation)

```
/Users/<username>/.jenkins/workspace/<job-name>/
```

If installed as a service or manually relocated:

```
/usr/local/var/jenkins/workspace/<job-name>/
```

Where Are Build Artifacts Stored?

When using Jenkinsfile stage:

```
archiveArtifacts artifacts: 'dist/**'
```

Jenkins copies the artifacts to:

Artifact Storage in Jenkins UI

```
Jenkins Dashboard → Job → Specific Build # → Artifacts
```

These are stored on disk under:

Windows

```
C:\ProgramData\Jenkins\.jenkins\jobs\<job-name>\builds\<build-number>\archive\
```

macOS

```
/Users/<username>/.jenkins/jobs/<job-name>/builds/<build-number>/archive/
```

What the Learner Must Understand

- Jenkins runs **locally**, so artifacts remain on the **same local machine**.
- Archived artifacts are visible both:
    1. In the Jenkins UI
    2. On the filesystem inside Jenkins directories
- Next module (Podman) will **consume these artifacts** for creating images.

Private Repo Version

```
pipeline {
    agent any

    tools {
        nodejs "NodeJS17"
    }

    stages {
        stage('Checkout') {
            steps {
                git branch: 'main',
                    credentialsId: 'github-pat',
                    url: 'https://github.com/my-org/private-nestjs-app.git'
            }
        }

        stage('Install Dependencies') {
            steps {
                sh 'npm install'
            }
        }

        stage('Test') {
            steps {
                sh 'npm run test'
            }
        }

        stage('Build') {
            steps {
                sh 'npm run build'
            }
        }
```

```
        stage('Archive Build') {
            steps {
                archiveArtifacts artifacts: 'dist/**', fingerprint: true
            }
        }
    }
}
```

Windows Batch Equivalent

Replace `sh` with `bat`:

```
bat 'npm install'
bat 'npm run test'
bat 'npm run build'
```

SECTION 7 — CI for React App

Jenkins Pipeline Requirements:

1. Clone code
2. Install npm packages
3. Run tests (`npm test --watchAll=false`)
4. Build production bundle
5. Archive `/build` folder

7.1 Sample Jenkinsfile

```
pipeline {
    agent any

    tools {
        nodejs "NodeJS17"
    }

    stages {
        stage('Checkout') {
            steps {
                git url: 'https://github.com/my-org/react-app.git', branch:
'main'
            }
        }

        stage('Install') {
            steps {
                sh 'npm install'
            }
```

```
        }

        stage('Test') {
            steps {
                sh 'npm test --watchAll=false'
            }
        }

        stage('Build') {
            steps {
                sh 'npm run build'
            }
        }

        stage('Archive Build') {
            steps {
                archiveArtifacts artifacts: 'build/**', fingerprint: true
            }
        }
    }
}
```

SECTION 8 — Combined Pipeline: NestJS + React (Multi-Stage)

Often the repo contains:

```
/backend
/frontend
```

Combined Jenkinsfile

```
pipeline {
    agent any
    tools { nodejs "NodeJS17" }

    stages {

        stage('Checkout') {
            steps {
                git url: 'https://github.com/my-org/fullstack-app.git',
branch: 'main'
            }
        }

        stage('Backend CI') {
            steps {
                dir('backend') {
                    sh 'npm install'
                    sh 'npm test'
                    sh 'npm run build'
                }
            }
```

```
        }

        stage('Frontend CI') {
            steps {
                dir('frontend') {
                    sh 'npm install'
                    sh 'npm test --watchAll=false'
                    sh 'npm run build'
                }
            }
        }

        stage('Archive Artifacts') {
            steps {
                archiveArtifacts artifacts: '**/dist/**, **/build/**'
            }
        }
    }
}
```

SECTION 9 — What Learners Must Know BEFORE Podman Module

Although Podman + images will be taught next, they need to know:

1. Build output folders

- NestJS output: `/dist`
- React output: `/build`

2. These become context for Podman images

3. Jenkins will later call `podman build`, but NOT now

4. Folder structure must be clean

```
backend/Dockerfile
frontend/Dockerfile
podman-compose.yml
```

5. Jenkinsfile must archive artifacts to use later

SECTION 10 — LABS / DEMOS

LAB 1 — Install Jenkins

Windows

1. Install Java 17
2. Install Jenkins MSI
3. Open `http://localhost:8080`

macOS

1. Install Java 17
2. Install via Homebrew
3. Start Jenkins service

LAB 2 — Create GitHub Webhook

1. Create public repo
2. Add Jenkins webhook
3. Commit a file and trigger Jenkins

LAB 3 — Create a Freestyle Job

1. Clone NestJS repo
2. Run build commands
3. Observe build output

LAB 4 — Create a Pipeline Job (Declarative)

1. Create a repo with Jenkinsfile
2. Setup pipeline from SCM
3. Run pipeline
4. Observe stages in Blue Ocean

LAB 5 — Combined Backend + Frontend CI

1. Push folder structure
2. Run multi-stage pipeline

3. Validate archived artifacts