



DevOps Professional

Guide - Jenkins

Advanced Pipelines, Docker CI/CD & GitHub Integration

Complete Tutorial with Git & Git Bash

Generated: 2026-02-04

Table of Contents

Right-click the TOC and select "Update Field" to refresh page numbers

Jenkins Advanced Pipeline (Production Style)	3
Key Features Explained	3
Jenkins + Docker Secure CI/CD	5
Security Best Practices	5
Jenkins + SonarQube + Quality Gate	6
Quality Gate Benefits	6
Jenkins + Nexus (Artifact Management)	7
GitHub Integration with Git Bash	8
Step 1: Setting Up Git	8
Install Git	8
Configure Git Identity	8
Step 2: Creating a Repository	8
Initialize a New Repository	8
Create Repository on GitHub	8
Step 3: Pushing Files to GitHub	9
First Time Push	9
Subsequent Pushes	9
Step 4: Common Git Commands	9
Step 5: Working with Branches	10
Step 6: Handling Common Issues	10
Authentication Errors	10
Merge Conflicts	10
Undo Changes	11

Jenkins Advanced Pipeline (Production Style)

This pipeline demonstrates a real-world Jenkins setup with stability, safety, and maintainability in mind.

```
pipeline {
  agent any
  options {
    timeout(time: 30, unit: 'MINUTES')
    disableConcurrentBuilds()
    buildDiscarder(logRotator(numToKeepStr: '15'))
  }
  stages {
    stage('Checkout') {
      steps {
        git branch: 'main',
            url: 'https://github.com/USERNAME/REPO.git'
      }
    }
    stage('Build') {
      steps {
        sh 'mvn clean package -DskipTests'
      }
    }
    stage('Test') {
      steps {
        sh 'mvn test'
      }
    }
    stage('SonarQube Analysis') {
      steps {
        sh 'mvn sonar:sonar'
      }
    }
  }
  post {
    success {
      archiveArtifacts artifacts: 'target/*.jar'
    }
    always {
      cleanWs()
    }
  }
}
```

Key Features Explained

- **Timeout Control:** Prevents builds from running indefinitely (30 minutes max)
- **Concurrent Build Prevention:** Ensures only one build runs at a time per pipeline
- **Log Rotation:** Keeps only the last 15 builds to save disk space

- **Workspace Cleanup:** Automatically cleans workspace after each build

Jenkins + Docker Secure CI/CD

This pipeline builds and pushes Docker images securely using Jenkins Credentials.

```
pipeline {
  agent any
  environment {
    IMAGE = "dockerhubuser/myapp"
    TAG = "${BUILD_NUMBER}"
  }
  stages {
    stage('Checkout') {
      steps {
        git 'https://github.com/USERNAME/REPO.git'
      }
    }
    stage('Docker Build') {
      steps {
        sh 'docker build -t $IMAGE:$TAG .'
      }
    }
    stage('Docker Push') {
      steps {
        withCredentials([usernamePassword(
          credentialsId: 'dockerhub-creds',
          usernameVariable: 'USER',
          passwordVariable: 'PASS'
        )]) {
          sh '''
            echo $PASS | docker login -u $USER --password-stdin
            docker push $IMAGE:$TAG
          '''
        }
      }
    }
  }
  post {
    always {
      sh 'docker logout'
    }
  }
}
```

Security Best Practices

- **Credential Management:** Uses Jenkins Credentials plugin to store secrets securely
- **Environment Variables:** Build number used as image tag for traceability
- **Secure Login:** Password passed via stdin to avoid shell history exposure
- **Logout Cleanup:** Always logs out from Docker registry after push

Jenkins + SonarQube + Quality Gate

SonarQube ensures code quality and security. Quality Gates prevent bad code from reaching production.

```
stage('Quality Gate') {  
    steps {  
        timeout(time: 2, unit: 'MINUTES') {  
            waitForQualityGate abortPipeline: true  
        }  
    }  
}
```

Quality Gate Benefits

- **Code Coverage:** Ensures minimum test coverage thresholds are met
- **Bug Prevention:** Blocks code with critical or blocker issues
- **Security Scanning:** Detects vulnerabilities before deployment
- **Technical Debt:** Prevents accumulation of code smells

Jenkins + Nexus (Artifact Management)

Nexus stores versioned build artifacts, enabling rollback and stable deployments.

```
stage('Publish Artifact') {  
    steps {  
        sh 'mvn deploy'  
    }  
}
```

Artifacts are stored in Nexus with version numbers and can be retrieved later for deployment.

- **Version Control:** Every build is versioned and traceable
- **Rollback Capability:** Quickly revert to previous stable versions
- **Dependency Management:** Centralized storage for all dependencies
- **Release Promotion:** Promote artifacts through environments

GitHub Integration with Git Bash

This section provides a comprehensive guide for uploading files to GitHub using Git Bash.

Step 1: Setting Up Git

Install Git

Download and install Git from <https://git-scm.com/downloads>. During installation, select "Git Bash Here" option.

Configure Git Identity

Set your name and email (used in commit history):

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"

# Verify configuration
git config --list
```

Step 2: Creating a Repository

Initialize a New Repository

```
# Navigate to your project folder
cd /path/to/your/project

# Initialize Git repository
git init

# Check repository status
git status
```

Create Repository on GitHub

1. Go to GitHub.com and sign in
2. Click the '+' icon and select "New repository"
3. Enter repository name and description
4. Choose public or private visibility
5. Do NOT initialize with README (we'll push existing code)

6. Copy the repository URL (HTTPS or SSH)

Step 3: Pushing Files to GitHub

First Time Push

```
# Add all files to staging area
git add .

# Or add specific files
git add filename.txt

# Commit with a message
git commit -m "Initial commit"

# Add remote repository (replace with your URL)
git remote add origin https://github.com/USERNAME/REPO.git

# Push to GitHub
git push -u origin main

# If your default branch is 'master':
git push -u origin master
```

Subsequent Pushes

```
# Check status
git status

# Add modified files
git add .

# Commit changes
git commit -m "Description of changes"

# Push to GitHub
git push
```

Step 4: Common Git Commands

Command	Description
<code>git status</code>	Show working tree status
<code>git add .</code>	Stage all changes

Command	Description
<code>git commit -m 'msg'</code>	Commit staged changes
<code>git push</code>	Push commits to remote
<code>git pull</code>	Fetch and merge from remote
<code>git clone <url></code>	Clone a repository
<code>git log --oneline</code>	View commit history
<code>git branch</code>	List branches

Step 5: Working with Branches

```
# Create and switch to new branch
git checkout -b feature-branch

# Switch between branches
git checkout main

# List all branches
git branch -a

# Merge branch into current
git merge feature-branch

# Delete branch after merge
git branch -d feature-branch
```

Step 6: Handling Common Issues

Authentication Errors

If you get authentication errors, use a Personal Access Token instead of password:

```
# Generate token on GitHub: Settings > Developer settings > Personal access tokens
git remote set-url origin https://TOKEN@github.com/USERNAME/REPO.git
```

Merge Conflicts

When conflicts occur during merge or pull:

```
# Pull latest changes first
git pull origin main
```

```
# If conflicts exist, edit files to resolve
# Then mark as resolved
git add .
git commit -m "Resolved merge conflicts"
git push
```

Undo Changes

```
# Undo uncommitted changes
git checkout -- filename

# Undo last commit (keep changes)
git reset --soft HEAD~1

# Undo last commit (discard changes)
git reset --hard HEAD~1

# View commit history
git log --oneline
```



DevOps Professional Guide

Jenkins & GitHub Integration

Complete CI/CD Pipeline Tutorial

© 2026 - All Rights Reserved