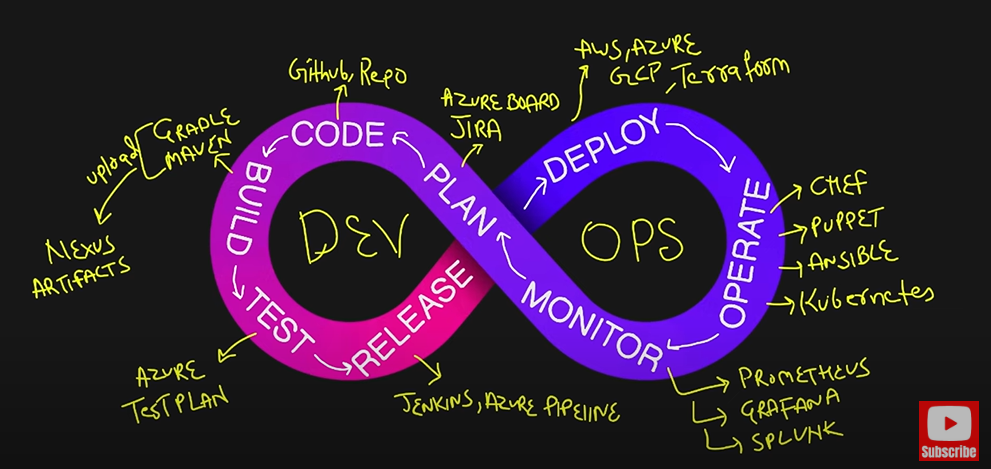
1. **Devops, CI/CD Pipeline, Jenkins**



**DevOps**

**What is DevOps?**

**DevOps** is a set of practices, principles, and tools that combine software development (**Dev**) and IT operations (**Ops**) to improve the speed, efficiency, and quality of delivering software. It fosters collaboration between development and operations teams, breaking down traditional differences to enable faster delivery, more reliable releases, and enhanced performance in the software lifecycle.

At its core, DevOps is about:

1. **Automation**: Automating repetitive tasks like testing, deployment, and monitoring.
2. **Collaboration**: Bridging the gap between development, operations, and other stakeholders.
3. **Continuous Processes**: Incorporating practices like Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment.

**The Need for DevOps**

In traditional software development, there were distinct differences between development teams (who wrote code) and operations teams (who deployed and maintained software). This led to several problems:

**1. Long Development Cycles**

* Developers would spend months building features before handing them off to operations for deployment.
* Testing and integration happened late, causing delays when issues were found.

**2. Lack of Collaboration**

* Developers focused on creating features, while operations focused on system stability.
* This led to conflicts, especially when rapid changes disrupted the production environment.

**3. High Failure Rates**

* Code written without considering the production environment often caused failures during deployment.

**4. Slow Recovery**

* When something broke in production, it took a long time to diagnose and fix, leading to downtime and frustrated users.

**Why DevOps is Needed**

DevOps addresses these challenges by creating a culture of collaboration and leveraging automation tools. Here's why organizations need it:

**1. Faster Time to Market**

* DevOps enables **frequent and reliable releases**, allowing businesses to quickly adapt to market changes or customer needs.
* For example, companies like Amazon deploy changes **thousands of times a day** thanks to DevOps practices.

**2. Improved Quality and Reliability**

* Automated testing and CI/CD pipelines ensure that bugs are caught early.
* Rollbacks and monitoring systems make releases more reliable.

**3. Enhanced Collaboration**

* Shared responsibility between development and operations eliminates the "us vs. them" mindset.
* Tools like Slack, Jira, and collaboration platforms enhance communication.

**4. Scalability**

* With tools like Kubernetes and Terraform, DevOps practices help applications scale seamlessly to handle increased traffic.

**Tools Used in DevOps**

* **Version Control**: GitHub, GitLab, Bitbucket
* **CI/CD**: Jenkins, GitLab CI/CD, CircleCI, Travis CI
* **Configuration Management**: Ansible, Puppet, Chef
* **Containerization and Orchestration**: Docker, Kubernetes
* **Monitoring and Logging**: Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana)
* **Infrastructure Automation**: Terraform, AWS CloudFormation

**Example of DevOps in Action**

**Scenario**: A company develops an online food delivery app.

**Before DevOps**:

1. Developers write code for new features like "Add to Favorites."
2. Code is handed to the operations team, who deploy it manually after weeks.
3. A bug is discovered post-release, causing the app to crash during peak hours.
4. Operations spends hours debugging without knowing the code changes.

**With DevOps**:

1. Developers commit code to a shared repository (e.g., GitHub).
2. Automated CI/CD pipelines test the code, build it, and deploy it to a staging environment.
3. Monitoring tools catch performance issues early.
4. If the code passes, it's deployed to production with minimal downtime.

**CI/CD pipeline**

A **CI/CD pipeline** in DevOps automates the process of software development, testing, and deployment, ensuring that applications are delivered faster and with higher quality. Let’s break it down into its two core components:

**1. Continuous Integration (CI)**

Continuous Integration is the practice of regularly merging all developers' code changes into a shared repository several times a day. Automated tools then build and test the new code to detect bugs early.

**Steps in CI:**

1. **Code Commit**: Developers write code and push it to a shared repository like GitHub, GitLab, or Bitbucket.
2. **Build Process**:
   * The CI system pulls the latest code.
   * It compiles the code into a usable application (if needed).
   * Dependencies (like libraries) are installed.
3. **Automated Testing**:
   * Tests are run automatically to ensure the new code doesn’t break the application.
4. **Feedback**:
   * If the build or tests fail, the system notifies the developers immediately (via email, Slack, or dashboards).
   * Developers fix the issues and repeat the process.

**Example of CI:** Imagine a team working on an e-commerce app. One developer adds a feature for a discount coupon while another works on a payment integration. Without CI, merging this code could break the app, and bugs may only be found later. With CI, automated tests immediately catch errors, ensuring the app remains stable.

**Tools for CI:**

* **Jenkins**: A widely-used open-source automation server.
* **CircleCI**: A cloud-based CI service.
* **Travis CI**: Ideal for open-source projects.
* **GitLab CI/CD**: Integrated directly into GitLab repositories.
* **Azure DevOps Pipelines**: Part of Microsoft Azure services.

**2. Continuous Delivery (CD)**

Continuous Delivery ensures that code changes that pass CI are automatically prepared for release to production. This involves automating further stages of testing and creating deployable artifacts (e.g., Docker containers, executables).

**Steps in CD:**

1. **Integration with CI**:
   * CD begins after CI has successfully built and tested the code.
2. **More Testing**:
   * Performance testing, security testing, and environment-specific testing (e.g., staging) may be included.
3. **Artifact Creation**:
   * Create ready-to-deploy packages (e.g., a Docker image).
4. **Manual Approval (Optional)**:
   * Teams can set up manual approval steps for critical environments like production.
5. **Deploy to Staging/Production**:
   * Deploy the code to a staging environment first and then to production.

**Example of CD:** Once the discount coupon feature passes CI, it’s automatically deployed to a staging environment. The product team tests the feature in a staging environment. If approved, the system deploys it to the production server.

**Tools for CD:**

* **AWS CodePipeline**: Automates the deployment of applications on AWS.
* **Azure DevOps Pipelines**: Manages builds, releases, and deployments.
* **Spinnaker**: An open-source tool for continuous delivery.
* **ArgoCD**: GitOps-based deployment for Kubernetes.
* **Octopus Deploy**: Focused on deployment and release management.

**3. Continuous Deployment (Optional)**

Continuous Deployment goes a step further than Continuous Delivery. Here, every code change that passes CI/CD tests is automatically deployed to production without manual intervention.

**Key Difference:**

* **Continuous Delivery**: Requires manual approval for production deployment.
* **Continuous Deployment**: Fully automated, no manual approval.

**Complete CI/CD Pipeline Example**

**Scenario: A Blogging Application**

1. **Development**:
   * A developer writes a feature to add image uploads.
   * The code is committed to a GitHub repository.
2. **CI Process**:
   * Jenkins detects the commit, pulls the code, and starts the pipeline.
   * It builds the application and runs automated tests (e.g., image upload functionality test).
   * If the tests pass, Jenkins creates a "build artifact" (e.g., a Docker container).
3. **CD Process**:
   * Jenkins pushes the artifact to a container registry (e.g., Docker Hub).
   * The code is deployed to a staging server for further testing (using Kubernetes or another deployment tool).
   * After validation, the artifact is either:
     + Manually approved for deployment to production (Continuous Delivery).
     + Automatically deployed to production (Continuous Deployment).
4. **Monitoring and Feedback**:
   * Tools like Prometheus and Grafana monitor the application’s performance in production.
   * Alerts notify the team if something goes wrong.

**Tools in CI/CD Pipelines**

**Version Control:**

* GitHub, GitLab, Bitbucket

**Build and CI Tools:**

* Jenkins, CircleCI, Travis CI, GitLab CI/CD

**Testing Tools:**

* Selenium (UI testing), JUnit (Java unit testing), PyTest (Python)

**Containerization and Deployment:**

* Docker (containerization), Kubernetes (orchestration), Terraform (infrastructure as code)

**Monitoring:**

* Prometheus, Grafana, Datadog

**Jenkins**

**What is Jenkins?**

Jenkins is an **open-source automation server** widely used in DevOps. It helps automate tasks like building, testing, and deploying software, enabling Continuous Integration (CI) and Continuous Delivery (CD). Jenkins allows developers to focus on writing code while it handles the repetitive tasks required to integrate and deploy the software.

Think of Jenkins as the "manager" in your software development process. When you push your code, Jenkins takes it, runs tests, builds it into an application, and can even deploy it automatically.

**Key Features of Jenkins**

1. **Pipeline Automation**: Automates the software development lifecycle, including building, testing, and deploying code.
2. **Extensibility with Plugins**: Jenkins supports over 1,800 plugins to integrate with tools like GitHub, Maven, Docker, and Kubernetes.
3. **Distributed Builds**: Jenkins can distribute work across multiple servers to speed up tasks.
4. **Continuous Integration and Delivery (CI/CD)**: It’s the backbone of CI/CD pipelines.
5. **Cross-Platform**: Runs on Windows, macOS, and Linux.
6. **Web Interface**: Provides an easy-to-use dashboard to manage jobs and pipelines.

**How Jenkins Works**

**1. Core Process**

* **Code Commit**: A developer pushes code changes to a repository (e.g., GitHub).
* **Trigger**: Jenkins detects the changes and triggers a pipeline.
* **Build**: Jenkins compiles the code, resolving dependencies (e.g., using Maven for Java).
* **Test**: Automated tests are executed (e.g., unit tests, integration tests).
* **Deploy**: Jenkins packages and deploys the application (e.g., on a server or Kubernetes cluster).

**2. Key Components in Jenkins**

* **Jobs**:
  + A job in Jenkins is a task or process like building an application or running tests.
  + Example: Building a Java app every time code changes are pushed to GitHub.
* **Build Pipeline**:
  + A series of steps (build → test → deploy) that Jenkins executes.
  + Example: A CI/CD pipeline for an e-commerce website.
* **Nodes/Agents**:
  + Jenkins can run tasks on different servers or machines (agents) to distribute work.
  + Example: Running builds on a Linux server and tests on a Windows server.
* **Plugins**:
  + Extend Jenkins functionality to integrate with version control systems (e.g., Git), build tools (e.g., Gradle), and cloud platforms (e.g., AWS).

**Example Jenkins Pipeline**

**Scenario: Building and Deploying a Python Application**

1. **Step 1: Install Jenkins and Plugins**
   * Install the Jenkins Git and Python plugins.
2. **Step 2: Configure the Pipeline**
   * Create a pipeline job and add the pipeline script.
3. **Step 3: Run the Pipeline**
   * Jenkins will:
     + Clone the code from GitHub.
     + Install dependencies.
     + Run automated tests.
     + Deploy the app to a remote server.

**Tools Integrated with Jenkins**

* **Version Control**: Git, GitHub, Bitbucket
* **Build Tools**: Maven, Gradle, Ant
* **Testing**: JUnit, Selenium, PyTest
* **Deployment**: Docker, Kubernetes, AWS
* **Notification**: Slack, Email

**Architecture of Jenkins**

Jenkins follows a master-slave architecture:

* **Jenkins Master**: The central server that manages the build process, schedules jobs, and monitors the slaves.
* **Jenkins Slaves**: Worker nodes that execute the build tasks assigned by the master. This setup allows for distributed builds, improving efficiency.

Sure! Let’s go through each of these topics in detail.

(Youtube Video: MPrashant)

Video job workflow:

1. Create a job for only printing hello world in execute shell

2. Create a file on local machine and run it in execute shell

3. Use Parameters: String parameters (FName, LName) and also choices (Male, Female)

4. Scheduling cron jobs using build periodically

5. Taking code from github by copying repo path in SCM and executing the code in execute shell

6. Automatically create a build job if any changes made to the code in github using Build triggers in poll SCM.

7. Sending email if any failure in code from github using post-build actions (remember the initial mail settings before creating a job)

8. Working with remote server using ssh password or ssh keygen for authentication, after authentication install plugin for ssh and then transfer a file from local server to remote server and we can also exclude some files that we don’t want to transfer, like from 5 files we don’t want to transfer one file then.

9. Working with ansible playbook

10. Project of pulling code from github on local machine and transferring it to the host machine and running the website using apache webserver

**Passwordless Authentication**

**Detailed Step-by-Step Guide: SSH Setup and File Transfer Using Jenkins**

This guide walks you through setting up SSH authentication between a local machine and a remote server, integrating it with Jenkins using the **Publish Over SSH plugin**, and configuring a Jenkins job to transfer files and execute commands.

**Step 1: Generate SSH Key on the Local Machine**

1. **Run the Command to Generate SSH Keys**: ssh-keygen
   * **Purpose**: Creates a public-private key pair for secure authentication.
   * **Prompts**: File Location: Press **Enter** to save the keys in the default path (~/.ssh/id\_rsa).
   * **Output**:
     + Two files:
       - id\_rsa (Private Key): Keep this secure; don’t share it.
       - id\_rsa.pub (Public Key): Share this with the remote machine.

**Step 2: Copy the Public Key to the Remote Machine**

1. **Run the Command**: ssh-copy-id username@ipaddress
   * Replace username with the remote server's username (e.g., ubuntu).
   * Replace ipaddress with the remote server's IP address (e.g., 192.168.1.100).
2. **Purpose**: This command appends your public key (id\_rsa.pub) to the remote machine’s ~/.ssh/authorized\_keys file.
3. **Authentication**: You may be prompted for the remote machine's password during this step.

**Step 3: Test the SSH Connection**

1. **Run the Command**: ssh username@ipaddress
   * This opens an SSH session with the remote machine.
2. **Expected Result**: If the setup is correct, you’ll log in without entering a password.
3. **Verify Files on the Remote Machine**: On the remote machine, check the ~/.ssh/authorized\_keys file to confirm the key was added:

cat ~/.ssh/authorized\_keys

**Step 4: Install the Publish Over SSH Plugin in Jenkins**

1. **Go to Jenkins**:
   * Open your Jenkins dashboard in a browser (http://localhost:8080).
2. **Install the Plugin**:
   * Navigate to **Manage Jenkins** → **Manage Plugins** → **Available**.
   * Search for **Publish Over SSH**.
   * Install the plugin and restart Jenkins if prompted.

**Step 5: Add SSH Credentials in Jenkins**

1. **Go to Publish Over SSH Configuration**:
   * Navigate to **Manage Jenkins** → **Configure System**.
   * Scroll to the **Publish Over SSH** section.
2. **Find Your Keys**:
   * On your local machine, navigate to the .ssh directory:
   * cd ~/.ssh
   * Files to locate:
     + id\_rsa (Private Key): Used for authentication.
     + id\_rsa.pub (Public Key): Already copied to the remote server.
3. **Add SSH Server Details**:
   * Add the following:
     + **Name**: A descriptive name for the server (e.g., "RemoteServer").
     + **Hostname**: The remote server’s IP address or domain (e.g., 192.168.1.100).
     + **Username**: The username used to log in (e.g., ubuntu).
     + **Private Key**: Either:
       - Upload the id\_rsa file.
       - Open the id\_rsa file in a text editor and copy its content, then paste it into the private key field.
4. **Save the Configuration**:
   * Test the connection by clicking **Test Configuration**. If successful, it confirms Jenkins can communicate with the remote machine.

**Step 6: Create a Freestyle Project in Jenkins**

1. **Create a New Job**:
   * Navigate to the Jenkins dashboard and click **New Item**.
   * Select **Freestyle Project**, and name it (e.g., "RemoteFileTransferJob").
2. **Configure Build Steps**:
   * Go to the **Build** section.
   * Add the step **Send files or execute commands over SSH**.

**Step 7: Create a Source File**

1. **Create a File in Jenkins Workspace**:
   * Navigate to Jenkins’ workspace directory: cd /var/lib/jenkins/workspace/RemoteFileTransferJob
   * Create a file (e.g., testfile.txt): echo "This is a test file" > testfile.txt

**Step 8: Configure File Transfer in Jenkins**

1. **Add Transfer Details**:
   * In the build step **Send files or execute commands over SSH**:
     + **Source Files**: Enter the path to the file to transfer (e.g., testfile.txt).
     + **Remote Directory**: Specify the target directory on the remote machine (e.g., /home/username/).
2. **Optional: Add Remote Commands**:
   * After transferring the file, you can execute commands on the remote machine. For example:

ls -l /home/username/

**Step 9: Build the Job**

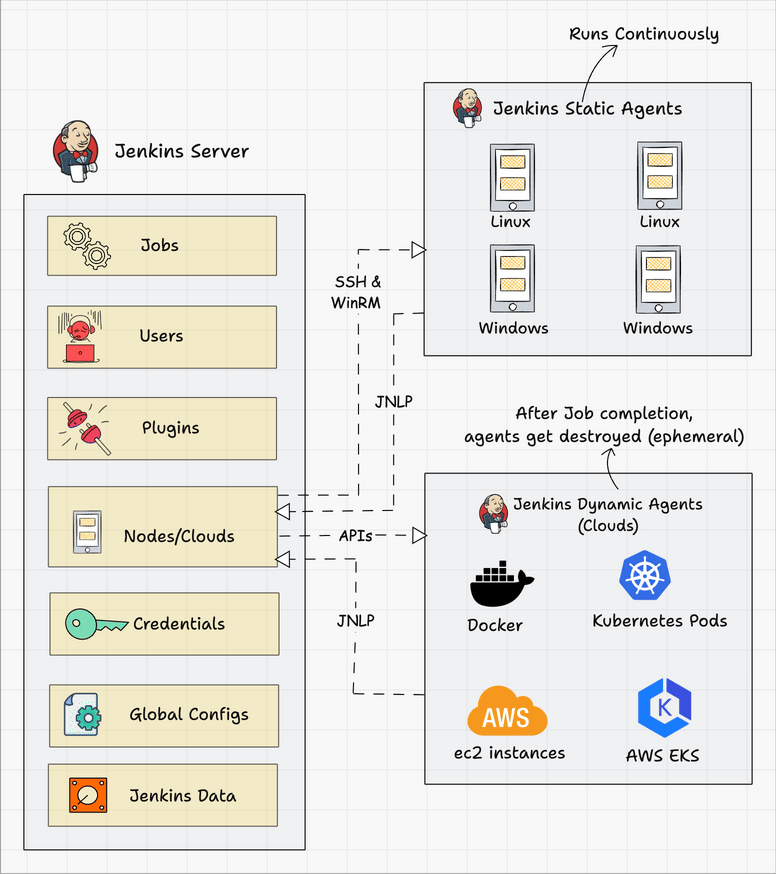
1. **Run the Job**: Click **Build Now** from the project dashboard.
2. **Verify on the Remote Machine**:
   * SSH into the remote machine: ssh username@ipaddress
   * Check the directory for the transferred file: ls -l /home/username/

**Summary of Commands**

1. **Generate SSH Key**:
2. ssh-keygen
3. **Copy Public Key to Remote Machine**:
4. ssh-copy-id username@ipaddress
5. **Test SSH Connection**:
6. ssh username@ipaddress
7. **File Transfer via Jenkins**:
   * Set up the **Publish Over SSH plugin**.
   * Configure the Jenkins job to transfer files and execute commands.
   1. **Jenkins Job:**

A **Jenkins job** is a task or a set of tasks that Jenkins runs. Jobs can be used for various purposes such as building code, running tests, deploying applications, and more. There are different types of Jenkins jobs:

* **Freestyle Project**: A flexible and simple job type that allows you to configure almost any build process.
* **Pipeline**: A job that defines the entire build process, including multiple stages, in a script (usually written in Groovy).
* **Multibranch Pipeline**: A pipeline job that can automatically create sub-jobs for each branch in a source control repository.
* **Maven Project**: A job specifically designed for building Maven projects.



* 1. **Running Bash Scripts with a Jenkins Job:**

To run a bash script in a Jenkins job, you can use the “Execute shell” build step. Here’s how you can do it:

1. **Create a New Job**:
   * Go to the Jenkins dashboard.
   * Click on “New Item”.
   * Enter a name for your job and select “Freestyle project”, then click “OK”.
2. **Configure the Job**:
   * In the job configuration page, scroll down to the “Build” section.
   * Click on “Add build step” and select “Execute shell”.
3. **Add Your Bash Script**:
   * In the “Command” text box, enter your bash script. For example: echo "Hello, World!"
   * You can also reference a script file stored in your repository:

./path/to/your-script.sh

1. **Save and Run the Job**:
   * Click “Save” to save the job configuration.
   * Click “Build Now” to run the job.
   1. **Scheduling Jobs in Jenkins:**

You can schedule Jenkins jobs to run at specific times using cron syntax. Here’s how to schedule a job:

1. **Configure Build Triggers**: In the job configuration page, scroll down to the “Build Triggers” section. Check the “Build periodically” option.
2. **Set the Schedule**:
   * In the “Schedule” text box, enter a cron expression. For example, to run the job every day at midnight:
   * \* \* \* \* \*
   * The syntax is similar to Linux cron syntax:
     + \*: Every Minute of the Hour.
     + \*: Every Hour of the day.
     + \*: Every day of the month.
     + \*: Every month.
     + \*: Every day of the week.
3. **Save the Configuration**: Click “Save” to apply the schedule.
4. **Working with GitHub in Jenkins**

To integrate GitHub with Jenkins, follow these steps:

1. **Install GitHub Plugin**:
   * Go to **Manage Jenkins** > **Manage Plugins**.
   * In the **Available** tab, search for “GitHub Plugin” and install it.
2. **Add GitHub Credentials**:
   * Go to **Manage Jenkins** > **Manage Credentials**.
   * Add your GitHub credentials (username and personal access token).
3. **Create a New Job**:
   * Go to the Jenkins dashboard and click on **New Item**.
   * Enter a name for your job and select **Freestyle project** or **Pipeline**, then click **OK**.
4. **Configure Source Code Management**:
   * In the job configuration, under **Source Code Management**, select **Git**.
   * Enter the repository URL and select the credentials you added earlier.
5. **Build Steps**:
   * Under **Build Steps**, click execute shell to run the particular code you are pulling form github (like python).

**5. Auto GitHub Repo Pulling**

To automatically pull changes from a GitHub repository:

1. **Configure Poll SCM**:
   * In the job configuration, under **Build Triggers**, check **Poll SCM**.
   * Enter a cron expression to specify how often Jenkins should check for changes (e.g., H/5 \* \* \* \* to poll every 5 minutes).
   * If any changes occur Jenkins will create a build.
2. **Email Notification from Jenkins**

Sending email if any failure in code from github using post-build actions (remember the initial mail settings before creating a job)

To set up email notifications in Jenkins:

1. **Configure Email Settings**:
   * Go to **Manage Jenkins** > **Configure System**.
   * Scroll down to **E-mail Notification** and enter your SMTP server details (e.g., for Gmail: smtp.gmail.com, port 587).
2. **Configure Email Notifications in Jobs**:
   * In the job configuration, under **Post-build Actions**, select **Email Notification**.
   * Tick the option of sending e-mail for every unstable build
3. **Make Changes in Code:** Make some changes in code to fail the running and then the job will send the mail to the developer notifying the mistake
4. **Working with Remote Server in Jenkins copy data from asus laptop**

To execute commands on a remote server from Jenkins:

1. **Install SSH Plugin**:
   * Go to **Manage Jenkins** > **Manage Plugins**.
   * In the **Available** tab, search for “SSH” and install the **SSH plugin**.
2. **Configure SSH Servers**:
   * Go to **Manage Jenkins** > **Configure System**.
   * Scroll down to **SSH remote hosts** and add your remote server details (hostname, username, password, or SSH key).
3. **Create a Job to Execute Remote Commands**:
   * In the job configuration, under **Build** section, add a **Build step** and select **Execute shell script on remote host using ssh**.
   * Enter the commands you want to run on the remote server.
4. **Test the Configuration**:
   * Save the job configuration and run the job to ensure it connects to the remote server and executes the commands successfully.

**Example**

Let’s say you want to set up a Jenkins job that pulls code from a GitHub repository, runs a build, and sends an email notification on success or failure, while also executing a script on a remote server.

1. **Create a New Job**:
   * Name: BuildAndDeployJob
   * Source Code Management: Git (GitHub repository URL)
   * Build Triggers: GitHub hook trigger for GITScm polling
2. **Add Build Steps**:
   * Execute shell: ./build.sh (your build script)
   * Execute shell script on remote host using ssh: ./deploy.sh (your deployment script on the remote server)
3. **Post-build Actions**: Editable Email Notification: Configure to send emails on build success or failure.

**8. Ansible Playbook with Job**

An **Ansible playbook** is a YAML file that defines a series of tasks to be executed on remote hosts. Here’s a simple example of an Ansible playbook that installs Apache on a web server:

---

- name: Install and start Apache

hosts: webservers

become: yes

tasks:

- name: Install Apache

apt:

name: apache2

state: present

- name: Start Apache

service:

name: apache2

state: started

To run this playbook as a job in Jenkins:

1. **Install ansible plugin**
2. **Create a Jenkins Job**:
   * Go to Jenkins dashboard and click on “New Item”.
   * Enter a name for your job and select “Freestyle project”, then click “OK”.
3. **Install Ansible of the local machine and create a basic playbook for pinging a website**
4. **Pass the path of the playbook file in the build step of invoke ansible playbook.**
5. **Configure the Job**:
   1. **Website update on Remote Server using Ansible (MPrashant Video)**

* Install Apache Webserver on remote server
* Create a Job in Jenkins (Freestyle Project)
* Pulled the code from github using SCM (Basic code taken from chatgpt)
* Created a Ansible Playbook for uploading a new code and restarting the service (website), where the host was the ip-address of the remote server and the tasks where
  1. Copy the file from the Jenkins location on the local server (the file saved after pulling it from github) and transfer it to remote machine
  2. After transferring, restart the service
  3. Become: true is given to provide sudo access to restart the service, as it would not get restarted if no sudo access
* Then in build steps, I invoked ansible playbook and provided the path of the playbook file
* Then I choosed the poll SCM option in build triggers to check every minute if any changes occur and if do it will get transferred directly.
* Then comes the post-build action, I choosed the email notification by which if any error occurs I will get notify about it.

**10. User Management (Role-Based) in Jenkins**

To manage users and roles in Jenkins, you can use the **Role Strategy Plugin**:

1. **Install the Plugin**:
   * Go to **Manage Jenkins** > **Manage Plugins**.
   * In the **Available** tab, search for “Role-based Authorization Strategy” and install it.
2. **Configure Roles**:
   * Go to **Manage Jenkins** > **Manage and Assign Roles** > **Manage Roles**.
   * Define global roles (e.g., admin, developer) and assign permissions.
3. **Assign Roles to Users**:
   * Go to **Manage Jenkins** > **Manage and Assign Roles** > **Assign Roles**.
   * Assign the defined roles to users or groups.

**10. Jenkins + Maven**

**Maven** is another build automation tool, but it focuses on project management and comprehension. It uses an XML file called pom.xml (Project Object Model) to manage project dependencies, build processes, and documentation. **Maven** is similar to Gradle but uses a different approach. Imagine you’re assembling a piece of furniture with a detailed instruction manual. Maven uses a file called pom.xml to list all the parts (dependencies) and steps (build processes) needed to complete your project. For example, if you’re building a Java application, Maven will download the necessary libraries and compile your code according to the instructions in the pom.xml file. To integrate Maven with Jenkins:

1. **Install Maven Plugin**:
   * Go to **Manage Jenkins** > **Manage Plugins**.
   * In the **Available** tab, search for “Maven Integration” and install it.
2. **Configure Maven**:
   * Go to **Manage Jenkins** > **Global Tool Configuration**.
   * Add Maven installations by specifying the name and installation path.
3. **Create a Maven Job**:
   * Go to Jenkins dashboard and click on “New Item”.
   * Enter a name for your job and select “Maven project”, then click “OK”.
   * Configure the job by specifying the POM file location and goals (e.g., clean install).

**11. Pipeline**

A Jenkins Pipeline automates the process of building, testing, and deploying applications. Here’s how to create a CI/CD pipeline:

1. **Create a Pipeline Job**:
   * Go to Jenkins dashboard and click on “New Item”.
   * Enter a name for your job and select “Pipeline”, then click “OK”.
2. **Define the Pipeline**:
   * In the job configuration, go to the “Pipeline” section.

pipeline {  
    agent any

    stages {  
        stage('build') {  
            steps {  
                echo 'build successful'  
            }  
        }  
        stage('test') {  
            steps {  
                echo 'test case runs successful'  
            }  
        }  
        stage('deploy') {  
            steps {  
                echo 'successfully deployed'  
            }  
        }  
    }  
}

**Example of a Real-World Pipeline**

1. **Git Source Job**: Pulls the source code from a Git repository.
2. **Analysis Jobs**: Runs code analysis tools like SonarQube.
3. **Deploy Jobs**: Deploys the application to different environments (e.g., customer and developer environments).
4. **Artifactory**

**Artifactory** is an open-source repository manager created by JFrog. It stores and manages your software artifacts, which are the files produced during the build process of your software projects. These artifacts can include compiled code, configuration files, and other necessary components.

**Why Use Artifactory?**

* **Centralized Storage**: Keeps all your build artifacts in one place.
* **Version Control**: Manages different versions of your artifacts.
* **Accessibility**: Team members can access and download artifacts from anywhere.
* **Integration**: Works seamlessly with CI/CD tools like Jenkins.

**Real-World Scenario**

Imagine a large software development team working on a project. They use a CI/CD pipeline to build and deploy their software. After the build process, artifacts are produced. Instead of storing these artifacts on a cluttered network share, they use Artifactory to manage them efficiently. This makes it easy to track versions, implement security, and share artifacts with team members, partners, and customers.

**Overview of the Project:** We will create a CI/CD pipeline for a simple Java-based REST API project. This project will expose one endpoint. The main goal is to integrate this project with Jenkins to automate the build and deployment process.

**Components Used**

1. **Source Control**: **Bitbucket**: **Bitbucket** is a Git-based source code repository that allows teams to collaborate on code, manage repositories, and integrate with other Atlassian tools like Jira and Confluence. [Bitbucket supports both Git and Mercurial repositories and offers features like pull requests, code reviews, and built-in CI/CD pipelines through Bitbucket Pipelines](https://bitbucket.org/). **Bitbucket** is like a digital locker for your code. Imagine you and your friends are working on a school project. Bitbucket lets you all store your project files in one place online, so everyone can access and update them. It also helps you keep track of changes and review each other’s work. For example, if you write a new piece of code, you can ask your friend to review it before it gets added to the main project.
2. **Build Framework**: **Gradle**: An open-source build automation tool that uses a Groovy-based DSL (Domain Specific Language) for project configuration. It simplifies the build process compared to XML-based tools like Apache Maven. **Gradle** is a tool that helps you build your software projects. Think of it like a chef following a recipe. Gradle takes your code and all its ingredients (like libraries and dependencies) and combines them to create the final product. For instance, if you’re making an Android app, Gradle will compile your code, package it, and prepare it for testing or release.
3. **Artifact Repository Manager**: **Jfrog Artifactory**: A universal repository manager that stores and manages binary artifacts (e.g., compiled code, configuration files). It supports multiple languages and integrates with various CI/CD tools. **Artifactory** is like a warehouse for storing all the different parts (artifacts) of your software. These parts can be anything from libraries to full applications. Artifactory helps you manage these parts, keep track of different versions, and share them with your team. It supports many types of packages, such as those used by Maven, Gradle, Docker, and npm, making it a versatile tool for managing software components.
4. **Building CI/CD pipeline with Jenkins and Maven**

Sure, let’s simplify the process of setting up a CI/CD pipeline with Jenkins and Maven:

**What You Need**

1. **Jenkins**: A tool to automate building and testing your code.
2. **Maven**: A tool to manage and build your Java projects.
3. **Git**: A place to store your code.

**Steps to Set Up**

**1. Install Jenkins and Maven**

* **Jenkins**: Download and install it from the Jenkins website.
* **Maven**: Download and install it from the Maven website.

**2. Configure Jenkins**

* **Install Plugins**: In Jenkins, install the Maven and Git plugins.
* **Set Up Maven**: Go to Manage Jenkins > Global Tool Configuration and add Maven by specifying its installation path.

**3. Create a Jenkins Job**

* **New Job**: Click New Item in Jenkins, name your job, and choose Freestyle project.
* **Source Code**: Under Source Code Management, select Git and enter your repository URL.
* **Build Triggers**: Set up automatic build triggers, like checking for new code periodically.

**4. Add Build Steps**

* **Build Step**: In the Build section, add a step to run Maven with goals like clean install (which cleans and builds your project).

**5. Post-Build Actions**

* **Test Results**: Add an action to publish test results if you have tests.
* **Deploy**: Add an action to deploy your build artifacts if needed.

**6. Run the Job**

* **Build Now**: Click Build Now to start the job.
* **Monitor**: Watch the build process in the Console Output.

**Using a Jenkins Pipeline**

For more complex workflows, you can use a Jenkinsfile in your project. Here’s a simple example:

pipeline {

agent any

tools {

maven 'Maven' // Name of the Maven installation in Jenkins

}

stages {

stage('Checkout') {

steps {

git 'https://github.com/your-repo/your-project.git'

}

}

stage('Build') {

steps {

sh 'mvn clean install'

}

}

stage('Test') {

steps {

junit '\*\*/target/surefire-reports/\*.xml'

}

}

stage('Deploy') {

steps {

// Add your deployment steps here

}

}

}

}

This file tells Jenkins to:

1. **Checkout**: Get the code from Git.
2. **Build**: Use Maven to build the project.
3. **Test**: Run tests and publish results.
4. **Deploy**: Deploy the build artifacts (if needed).

This setup helps automate the process of building, testing, and deploying your code, making it easier and faster to release new features.

1. **Build database schema deployment pipeline with Jenkins and Sqitch**

**What is Sqitch?**

**Sqitch** is a database change management tool that helps developers manage database schema changes in a reliable and consistent way. **Sqitch** is a tool that helps you manage changes to your database. Think of it as a version control system (like Git) but specifically for your database schema. It ensures that changes to your database are applied in a consistent and reliable way.

**Key Features**

1. **Framework-Free**: Sqitch doesn’t depend on any specific programming framework. You can use it with any project.
2. **Supports Multiple Databases**: It works with many database systems like PostgreSQL, MySQL, SQLite, Oracle, and more.
3. **Native Scripting**: You write your database changes in the native language of your database (like SQL for PostgreSQL).
4. **Dependency Management**: Sqitch keeps track of the order in which changes should be applied, ensuring everything happens in the right sequence.
5. **Integrity**: It uses a system similar to Git to ensure that changes are applied correctly and consistently.

**How Sqitch Works**

1. **Plan File**: This file lists all the changes you want to make to your database, in the order they should be applied.
2. **Change Scripts**: Each change has three scripts:
   * **Deploy Script**: Applies the change.
   * **Revert Script**: Undoes the change.
   * **Verify Script**: Checks that the change was applied correctly.
3. **Commands**: Sqitch provides commands to deploy changes, revert them if needed, and verify that everything is correct.

**Step-by-Step Guide for Build database schema deployment pipeline with Jenkins and Sqitch**

**1. Install Jenkins and other plugins**

**2. Install and Configure Sqitch**

* **Install Sqitch**: Follow the installation instructions for your operating system from the Sqitch website.
* **Initialize Sqitch Project**: Create a new Sqitch project in your repository.

sqitch init my\_project --engine pg

**3. Set Up Jenkins Job**

* **Create a New Job**
* **Configure Source Code Management**: Under the Pipeline section, configure your Git repository.

pipeline {

agent any

stages {

stage('Checkout') {

steps {

git 'https://github.com/your-repo/your-project.git'

}

}

stage('Deploy') {

steps {

sh 'sqitch deploy db:pg://user:password@localhost/mydb'

}

}

stage('Verify') {

steps {

sh 'sqitch verify db:pg://user:password@localhost/mydb'

}

}

}

}

**4. Create Sqitch Change Scripts**

* **Add a Change**: Create a new change for your database schema.

sqitch add create\_users\_table -n "Create users table"

* **Deploy Script**: Write the SQL to create the users table in deploy/create\_users\_table.sql.

**SQL**

CREATE TABLE users (

user\_id SERIAL PRIMARY KEY,

username VARCHAR(50) NOT NULL,

email VARCHAR(100) NOT NULL,

created\_at TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP);

* **Revert Script**: Write the SQL to drop the users table in revert/create\_users\_table.sql. DROP TABLE users;
* **Verify Script**: Write the SQL to check if the users table exists in verify/create\_users\_table.sql.

SELECT 1 FROM pg\_tables WHERE tablename = 'users';

**5. Configure Jenkins Pipeline**

* **Pipeline Script**: Use the following script in the Jenkins pipeline configuration.

pipeline {

agent any

tools {

sqitch 'Sqitch' // Name of the Sqitch installation in Jenkins

}

stages {

stage('Checkout') {

steps {

git 'https://github.com/your-repo/your-project.git'

}

}

stage('Deploy') {

steps {

sh 'sqitch deploy db:pg://user:password@localhost/mydb'

}

}

stage('Verify') {

steps {

sh 'sqitch verify db:pg://user:password@localhost/mydb'

}

}

}

}

**6. Run the Pipeline**

* **Build Now**: Click Build Now to start the pipeline.
* **Monitor**: Watch the build process in the Console Output.