**JENKINS**

Jenkins is an open-source automation server that is primarily used to automate tasks related to software development, such as building, testing, and deploying code. It is widely used for Continuous Integration (CI) and Continuous Delivery (CD), which are practices that help improve the quality and speed of software development.

**Key Concepts of Jenkins:**

**Continuous Integration (CI):**

Jenkins allows developers to automatically integrate their code changes into a shared repository multiple times a day. This helps detect bugs and issues early in the development cycle.

**Continuous Delivery (CD):**

Jenkins automates the delivery pipeline, meaning it can automatically deploy new versions of your software to various environments (e.g., staging, production).

**Automation:**

Jenkins automates various tasks like building projects, running tests, and deploying applications. This reduces the need for manual intervention and speeds up the development cycle.

**Plugins:**

Jenkins supports a wide variety of plugins that extend its functionality. These plugins can integrate Jenkins with version control systems (like Git), build tools (like Maven or Gradle), testing frameworks (like JUnit), deployment platforms, and more.

**Pipelines:**

Jenkins allows you to define your CI/CD workflows in code using Jenkinsfiles. A Jenkinsfile is a text file that defines the steps and actions Jenkins will perform when running a job, such as pulling the latest code from a Git repository, running tests, and deploying to a server.

**Distributed Builds:**

Jenkins can distribute build tasks across multiple machines, allowing for faster execution of jobs in larger teams or on more complex systems.

**Benefits of Jenkins:**

**Automates repetitive tasks:** Build, test, and deploy processes can be automated, reducing the potential for human error.

**Improves code quality:** With automated testing, Jenkins ensures that new code integrates smoothly with the existing codebase.

**Faster delivery:** Continuous delivery with Jenkins enables faster and more reliable delivery of software updates.

**Flexibility:** It supports numerous plugins and integrates with almost any tool in the software development lifecycle.

**Jenkins in DevOps**

In the context of DevOps, Jenkins plays a pivotal role by enabling Continuous Integration (CI) and Continuous Delivery (CD), two key principles of DevOps. Here's how Jenkins fits into the DevOps lifecycle:

**Continuous Integration (CI):**

CI is the practice of frequently integrating code changes into a shared repository. Jenkins helps automate the process by continuously building, testing, and integrating code after every change.

Developers commit their code to a version control system (e.g., Git), and Jenkins automatically triggers a build and tests the code.

By automating builds and tests, Jenkins ensures that bugs are detected early, improving code quality.

**Continuous Delivery (CD):**

CD extends CI by automatically deploying code to production or other environments (staging, testing) after a successful build and test cycle.

Jenkins can automate the entire deployment process, pushing code changes through multiple stages (e.g., build, test, deploy) and ensuring that the software is always in a deployable state.

This ensures faster, more reliable releases with minimal manual intervention, a core goal of DevOps.

**Automation of Repetitive Tasks:**

Jenkins automates tasks like code compilation, running unit tests, security scans, and deploying to staging or production environments.

This automation eliminates the need for manual processes, reduces human error, and speeds up software delivery cycles.

**Collaboration:**

DevOps emphasizes collaboration between development and operations teams. Jenkins helps by providing a central automation tool that both teams can use to ensure smooth, consistent, and repeatable processes.

Jenkins' flexible configuration, job monitoring, and notifications allow for collaboration in real-time across teams, making it easier for developers, testers, and operations staff to work together.

**Infrastructure as Code (IaC):**

Jenkins integrates with Infrastructure as Code (IaC) tools like Terraform, Ansible, Puppet, or Chef. This allows DevOps teams to automate and version-control their infrastructure.

Jenkins can trigger infrastructure changes, manage deployment configurations, and ensure that the environment setup matches the code, improving consistency across environments.

**Pipeline as Code:**

Jenkins allows you to define CI/CD pipelines in code using Jenkinsfile. This allows teams to version and store their build and deployment processes in the same way they manage source code.

This concept is referred to as Pipeline as Code, which is a core part of the DevOps culture, enabling teams to codify their entire delivery workflow.

**Feedback Loop:**

Jenkins fosters a rapid feedback loop, which is crucial for DevOps. Automated testing and reporting can immediately inform developers about issues with code, helping them quickly fix problems before they get into the production environment.

**How Jenkins Supports DevOps Principles:**

**Collaboration:** Jenkins promotes teamwork by enabling both development and operations teams to integrate and deploy code in a smooth, automated way.

**Automation:** Automates the entire lifecycle from code commit to build, test, and deployment, reducing manual effort and speeding up software delivery.

**Continuous Improvement:** Jenkins' role in CI/CD ensures that developers continuously deliver improvements to software, helping organizations quickly respond to feedback and customer needs.

**Monitoring and Feedback**: Jenkins provides real-time feedback on the state of the code, builds, tests, and deployments. This enables proactive issue resolution and quick reactions to changes.

**Consistency:** Through automation, Jenkins ensures that software is built, tested, and deployed the same way every time, regardless of environment or team.

**Jenkins and DevOps Tools Integration:**

Jenkins integrates well with a variety of DevOps tools to create a seamless DevOps pipeline. Some common integrations include:

* Version Control Systems: Git, SVN
* Build Tools: Maven, Gradle
* Testing Frameworks: JUnit, Selenium
* Deployment Tools: Kubernetes, Docker, Ansible
* Monitoring: Prometheus, Grafana
* ChatOps: Slack, Microsoft Teams for notifications

**Example DevOps Pipeline with Jenkins:**

**Code Commit:** Developers commit code to a version control system (e.g., Git).

**Jenkins Build Trigger:** Jenkins detects the change and triggers a build.

**Automated Testing:** Jenkins runs unit tests, integration tests, and code quality checks.

**Build and Package:** If tests pass, Jenkins builds the application and packages it for deployment.

**Deployment to Staging:** Jenkins deploys the application to a staging environment for further testing.

**Approval for Production:** If staging tests are successful, Jenkins deploys the application to production (or triggers further approval processes).

**Monitoring:** Once the app is live, Jenkins can trigger monitoring tools to track performance, and logs can be sent to chat platforms for the team to monitor.

**In DevOps (automation of repetitive tasks) we consist of below stages:**

* + Plan
  + Code
  + Build
  + Test
  + Release
  + Deploy
  + Operate
  + Monitor

In Release stage (Pipeline) the DevOps stages Code, Build, Test, Release and Deploy will integrate with CI/CD tools.

Mainly there are three types pf pipelines, they are

1. Freestyle
2. Pipeline
3. Multibranch

**CI/CD:** Continuous Integration and Continuous Delivery/Deployment

**Tools used for the CI/CD are:** Jenkins (it is open-source tool), Gitops, travisCI, bamboocity, ...

**CI (Continuous Integration):** which comes under the development portion.

: which integrates the DevOps stages Plan, Code, Deploy and Test.

**Continuous Delivery:** In which deployment will be done in non-production regions.

**Continuous Deployment:** In which deployment will be done in production regions.

In real time environment we have three regions, they are

* Dev region
* Testing region: In this we have UAT (User Acceptance Testing) and SIT (System Integration Testing)
* Production region

**The deployment process will be done in following steps:**

* Developer will develop the code and send the code to UAT region.
* In UAT region, testing will be done and send to SIT region.
* In SIT region, testing will be done and send to Pre-Production region.
* In Pre-Production region, continuous delivery (which is the higher version of release code) will be done and send to Production region.
* In production region continuous deployment (which is current running version (live) version) will be done.

**Deployments in Jenkins:**

In Jenkins, deployment refers to the process of automating the delivery of an application or software component to a production or staging environment. There are several ways to handle deployments depending on your workflow, infrastructure, and tools in use.

**Here are the types of deployments in Jenkins:**

**1. Manual Deployment**

**Description:** In a manual deployment, Jenkins is configured to trigger a deployment, but the process itself requires human intervention or approval.

**Use Case:** Used when deployments require confirmation or validation before going to production.

**Example:** After Jenkins builds and tests the code, an operator manually triggers the deployment to the production environment after reviewing the output.

**Common Tools:** Jenkins' built-in features like input steps for approvals.

**2. Automated Deployment (CI/CD Pipelines)**

**Description:** In an automated deployment, Jenkins automatically deploys code to various environments (e.g., staging, production) after successful builds and tests.

**Use Case:** This type is common in CI/CD workflows where after each successful build or merge, code is automatically deployed to the next environment.

**Example:** When code passes all unit tests, Jenkins deploys the build to a staging environment for further testing, and if everything goes well, it is then deployed to production.

**Common Tools:** Jenkins Pipelines, Docker, Kubernetes, and cloud deployment tools like AWS, Azure, or Google Cloud.

**3. Blue-Green Deployment**

**Description:** This type of deployment involves running two identical production environments: Blue and Green. At any time, one environment is live, while the other one is idle or used for testing. Once the new version of the application is tested and ready in the idle environment, traffic is switched to the updated environment.

**Use Case:** This ensures zero downtime during deployment and reduces the risk of failure by allowing quick rollbacks.

**Example:**

The Blue environment is currently serving live traffic.

A new version is deployed to the Green environment and tested.

After verification, traffic is routed to the Green environment, making it live.

The Blue environment becomes idle and can be used for the next update.

Common Tools: Jenkins Pipelines, Docker, Kubernetes, AWS Elastic Beanstalk, or any platform that supports load balancing.

**4. Canary Deployment**

**Description:** In a canary deployment, a new version of the application is rolled out to a small subset of users or servers (the "canaries") to test the new release before rolling it out to the entire production environment.

**Use Case:** This is useful when you want to test new features in a production-like environment but with a limited impact, so you can detect any issues early.

**Example:** A new version of the application is deployed to only 5% of the servers or users, and once it's proven stable, it’s gradually rolled out to the remaining 95%.

**Common Tools:** Jenkins Pipelines, Docker, Kubernetes, and services like AWS Elastic Load Balancing, Azure App Service.

**5. Rolling Deployment**

**Description:** A rolling deployment gradually replaces the old version of the application with the new version across multiple servers or containers. The new version is deployed in chunks, and traffic is shifted incrementally to the updated servers.

**Use Case:** Often used for applications with multiple instances to ensure zero downtime by updating servers one by one or in small batches.

**Example:** The old application is replaced with the new version across servers or containers in batches. Each batch is tested before moving on to the next.

**Common Tools:** Jenkins Pipelines, Docker, Kubernetes, AWS Elastic Beanstalk, Azure App Service.

**6. Serverless Deployment**

**Description:** In serverless deployment, the code is deployed to serverless platforms (e.g., AWS Lambda, Google Cloud Functions). These platforms automatically scale the application, and users don’t need to manage infrastructure.

**Use Case:** Ideal for microservices or event-driven architectures where managing servers is unnecessary.

**Example:** Jenkins can deploy code to AWS Lambda functions, automatically creating and updating serverless functions that trigger based on specific events (e.g., an API request or a file upload).

**Common Tools:** Jenkins Pipelines, AWS Lambda, Google Cloud Functions, Serverless Framework.

**7. Hybrid Deployment**

**Description:** A hybrid deployment involves a combination of the above deployment strategies, for example, using a rolling deployment for the backend servers while using blue-green deployment for the frontend.

**Use Case:** Suitable for complex applications that have multiple components requiring different deployment strategies.

**Example:** A microservice backend may use rolling deployment while the front-end application uses blue-green deployment for fast and seamless transitions.

**Common Tools:** Jenkins Pipelines, Kubernetes, Docker, AWS, Azure, and custom deployment scripts.

**8. Docker Container Deployment**

**Description:** With Docker, Jenkins can automate the creation, testing, and deployment of containerized applications. Jenkins builds the Docker image, runs tests, and deploys it to container orchestration platforms like Kubernetes, Docker Swarm, or a cloud platform.

**Use Case:** Ideal for microservices, cloud-native applications, or when portability and consistency across environments are needed.

**Example:** Jenkins builds a Docker image and pushes it to a container registry (e.g., Docker Hub). The image is then deployed to Kubernetes or Docker Swarm for scaling.

**Common Tools:** Jenkins Pipelines, Docker, Kubernetes, AWS ECS.

**9. Infrastructure as Code (IaC) Deployment**

**Description:** Jenkins can be used to automate the deployment of infrastructure itself by leveraging tools like Terraform, Ansible, Chef, or Puppet. This approach allows infrastructure to be managed as code, which is version-controlled and can be easily deployed and replicated.

**Use Case:** Used for provisioning, updating, and managing infrastructure in a consistent, repeatable manner.

**Example:** Jenkins triggers a Terraform script to provision infrastructure on AWS or deploy configurations using Ansible.

**Common Tools:** Jenkins Pipelines, Terraform, Ansible, Puppet, Chef.

**Summary of Common Deployment Strategies:**

| **Deployment Type** | **Description** | **Use Case** |
| --- | --- | --- |
| **Manual Deployment** | Requires human intervention to trigger deployment. | Used when approval or validation is needed before production. |
| **Automated Deployment** | Automatically deploys after build and tests. | Common in CI/CD workflows for fast and continuous releases. |
| **Blue-Green Deployment** | Two environments; one live, one idle, switch traffic after deployment. | Zero downtime, safe deployments with rollback option. |
| **Canary Deployment** | Deploy to a small subset of users/servers first, then expand. | Gradual rollout with early detection of issues. |
| **Rolling Deployment** | Gradually replaces old version with the new one across servers. | Ensures zero downtime with minimal risk. |
| **Serverless Deployment** | Deploy to serverless environments like AWS Lambda. | Ideal for event-driven applications with no server management. |
| **Hybrid Deployment** | Combines multiple deployment strategies. | Complex applications requiring different strategies. |
| **Docker Container Deployment** | Automates the deployment of containerized applications. | Portability and consistency in microservices. |
| **Infrastructure as Code Deployment** | Automates infrastructure provisioning and management. | Infrastructure management as code, often in cloud environments. |

**Jenkins Installation in AWS Server:**

-First, we have to create an instance in AWS.

-For running the jenkins, it requires JDK (Java Development Kit) or JRE (Java Runtime Engine).

-Then search for jenkins.io in browser -> then click the download option -> and select the required OS -> then copy the required script for JDK installation and Jenkins installation.

-Now create a file with the name jenkins.sh in the server and paste the copied script from the browser.

: vi jenkins.sh

**The copied script to the jenkins.sh file:**

#!/bin/bash

sudo apt update

sudo apt install fontconfig openjdk-17-jre

sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \

https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key

echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc]" \

https://pkg.jenkins.io/debian-stable binary/ | sudo tee \

/etc/apt/sources.list.d/jenkins.list > /dev/null

sudo apt-get update

sudo apt-get install jenkins

-Then run the jenkins file to install the Jenkins

: sh jenkins.sh

-After that check the installed version of the Jenkins, by using below command.

: jenkins –version

-To check the status of the Jenkins, we can use below command.

: systemctl status jenkins

-The default directory of Jenkins in the server is: /var/lib/jenkins

-And the Installed plugins will store in: /var/lib/jenkins/plugins

-Newly created users will be stored in: /var/lib/jenkins/users

-Jobs run in the Jenkins will be stored in: /var/lib/jenkins/jobs

-To access the Jenkins application, open the browser and search with the public ip address of the server and port number.

-To install plugins in Jenkins:

-> Dashboard -> Manage Jenkins -> plugins -> available plugins -> search for the required plugin and install

-To create a user:

-> Dashboard -> Manage Jenkins -> Users -> Create User -> Provide the details and create

**Jobs in Jenkins:**

In Jenkins, a job refers to a task or a unit of work that Jenkins will execute. Jobs can be configured to perform various tasks such as building, testing, and deploying applications. There are different types of Jenkins jobs, and each one serves a different purpose. Here’s an overview of common types of Jenkins jobs:

**1. Freestyle Project**

**Description:** This is the most basic type of job in Jenkins. It allows you to configure a job through a simple user interface, providing options to define various build steps, triggers, and post-build actions.

**Common Uses:** Simple tasks like building code, running tests, or deploying applications.

**2. Pipeline**

**Description:** A pipeline is a more advanced type of job that defines a series of steps (stages) to be executed in a pipeline for building, testing, and deploying your application. The pipeline is defined as code (using Jenkinsfile) and offers more flexibility than a Freestyle Project.

**Common Uses:** Continuous Integration and Continuous Delivery (CI/CD) workflows, complex multi-stage builds.

**3. Multi-Branch Pipeline**

**Description:** This job type automatically creates a pipeline for each branch in a Git repository. It detects new branches and automatically creates a corresponding job for them.

**Common Uses:** Managing pipelines for projects with multiple branches, like feature branches, without having to manually configure jobs for each branch.

**4. GitHub Organization**

Description: This type of job integrates with GitHub and automatically creates jobs for each repository in a GitHub organization.

**Common Uses:** Automating CI/CD for many repositories within a GitHub organization.

**5. Maven Project**

**Description:** A special type of job that integrates with Maven, a build automation tool for Java projects. This job type is pre-configured for handling Maven builds, testing, and deployments.

**Common Uses:** Running Maven builds, managing dependencies, and deploying Java applications.

**6. External Job**

**Description:** This job type allows Jenkins to track the status of an external process, such as a build or test run, that is managed outside of Jenkins.

**Common Uses:** Monitoring builds and jobs running in external systems like remote build servers.

**7. Docker Pipeline**

**Description:** A specialized version of a pipeline job that uses Docker containers for building, testing, and deploying applications.

**Common Uses:** Running builds and tests within isolated Docker containers.

**8. Matrix Project**

**Description:** A job type used for running builds on multiple configurations, such as different versions of a language or different OS platforms. Each combination of configurations will be run as a separate job.

**Common Uses:** Running tests in multiple environments, cross-platform testing.

**9. Promoted Build**

**Description:** These jobs are related to marking builds as "promotions" when certain conditions are met. For example, once a build passes all tests, it can be "promoted" to a staging or production environment.

**Common Uses:** Approving or marking builds for deployment after passing quality checks.

**How Jenkins Jobs Work:**

**Source Code Management:** Jenkins can integrate with version control systems like Git, SVN, etc., to fetch the latest source code for the job.

**Build Triggers:** Jobs can be triggered by various events such as code commits, manual execution, scheduled times, or other jobs.

**Build Steps:** Jobs can include one or more steps such as compiling code, running tests, or deploying applications.

**Post-Build Actions:** After the job completes, actions like sending notifications, archiving artifacts, or deploying the application can be configured.

**Common Jenkins Job Uses:**

**Continuous Integration (CI):** Regularly building and testing code whenever changes are committed.

**Continuous Delivery (CD):** Automating the deployment process, ensuring code gets deployed to production or staging automatically after passing tests.

**Automation:** Automating repetitive tasks such as testing, building, packaging, and deployment.

**DevOps:** automates the software delivery using CI/CD tools like Jenkins with build triggers.

**Build Triggers:** In Jenkins, build triggers are mechanisms that automatically start the execution of a pipeline or job based on certain events or conditions.

These triggers help automate the build process and eliminate the need for manual intervention to start builds.

-In a Jenkins pipeline, there are different ways to trigger a build. Below are some of the most common triggers used in Jenkins pipelines:

**1. Poll SCM (pollSCM):** This trigger polls the source code management (SCM) system (e.g., Git, SVN) for changes and triggers a build if changes are detected.

**2. GitHub Webhook (GitHub trigger):** With a GitHub webhook, Jenkins can be configured to listen for specific events (e.g., push, pull request) and trigger builds automatically.

**3. Build After Other Projects are Built (Build Trigger):** This trigger allows you to set up a build dependency, where the pipeline triggers after another project/job finishes building successfully.

**4. Timer Trigger (cron):** A timer trigger uses cron-like syntax to schedule builds at specific times or intervals (e.g., run every day at midnight).

**5. Build on SCM Change (SCM Trigger):** This trigger is similar to pollSCM, but more focused on detecting changes and triggering based on them.

**6. Manual Trigger (Input Step):** In some cases, you may want to have the build pipeline paused and require manual intervention before proceeding. This can be done using the input step.

**7. Parameterized Trigger:** You can trigger a build based on parameters, allowing you to pass specific parameters to the build. This is often used to trigger a downstream job with specific inputs.

**8. Custom Triggers (Using Jenkinsfile hooks):** You can also write custom triggers or actions using hooks. This allows you to execute a script or some other command to trigger the pipeline.

**-> Build periodically:**

In Jenkins, setting up a build periodically trigger is a way to automatically trigger a build on a regular schedule.

This is done using cron syntax in the Jenkins pipeline configuration.

If you want to run a Jenkins pipeline periodically (for example, every day at midnight or every hour), you can use the cron trigger within the triggers block of your pipeline.

-create a pipeline with name: build periodically

-then in the configuration section we have the triggers option, in that select the "build periodically" option

\*It is used with cron expression:

\* \* \* \* \*

minutes (0-59) |hours (0-23) |day of month (1-31) |month (1-12) |day of the week (0-6)

**Example** : \* \* \* \* \* : Every minute

: 0 \* \* \* \* : Every hour

: 0 0 \* \* \* : Every day at 12:00 AM

: 0 0 \* \* FRI : At 12:00 AM, only on Friday

: 0 0 1 \* \* : At 12:00 AM, on the day 1 of the month

-provide the time period according to requirement

-then save and apply the job

-now the pipeline can run automatically, at your requested time

**-> Upstream and Downstream jobs:**

In Jenkins, upstream and downstream jobs are terms used to describe the relationship between different jobs or pipelines, especially when one job triggers another. This is a fundamental concept in Jenkins’ continuous integration and delivery (CI/CD) systems, allowing you to chain jobs together based on their execution order.

-now we have to configure two pipelines, in that second pipeline need to run automatically after the completion of first pipeline

-create a pipeline with name: upstream job and configure with hello world program

-then create another job with name: downstream job

-then in the configuration section we have the triggers option, in that select the "Build after other projects are build" option

-then provide the job name, that should complete before this job and select the required trigger option

-then configure the pipeline script with a sample hello world program

-then save and apply the job

-now, this pipeline is ready to run after the completion of the first job

**-> pollSCM:**

pollSCM is a feature that allows you to trigger a build when changes are detected in your source code repository (typically a Git repository or similar source control management system). It's often used in Jenkins pipelines to automatically build and test your code when new commits are pushed to the repository.

-create a pipeline with name pollSCM

-and configure with the hello world program in the pipeline script

-then open "configuration" and select "pipeline syntax" in the "pipeline" tab

-and select the checkout option in sample steps

-then provide the required details

github url:

credentials:

branch:

-then click on "Generate Pipeline Script" option

-now copy and paste that script in the pipeline script

-then in the configuration section we have the triggers option, in that select the "pollSCM" and provide the cron timing period to check the code changes

-then save and apply the job

-then pipeline will run only when there are changes in the github code in scheduled time interval

**-> Webhook:**

In Jenkins, a webhook is a way to trigger a Jenkins job (or pipeline) automatically when an event happens in an external system, such as a commit pushed to a GitHub repository, a merge request, or other events. Webhooks are commonly used to integrate Jenkins with version control systems (VCS) like GitHub, GitLab, Bitbucket, etc.

When a webhook is triggered, Jenkins automatically starts a job without needing manual intervention.

-create a pipeline with name Webhook

-then in the configuration section we have the triggers option, in that select the "GitHub hook trigger for GITScm polling" option

-and configure with the hello world program in the pipeline script

-then open "configuration" and select "pipeline Syntex" in the "pipeline" tab

-and select the checkout option in sample steps

-then provide the required details

github url:

credentials:

branch:

-then click on "Generate Pipeline Script" option

-now copy and paste that script in the pipeline script

-then copy the jenkins ip address with port number

-then open the github repository and open settings option and slect the webhooks option

-then click on "Add Webhook" and provide the password for webhook creation

-now paste the copied jenkins url with github-webhook/ in the "Payload URL"

-and select the "application/json" option in content type

-and select the "Send me everything" option in Which events would you like to trigger this webhook?

-and click on Add webhook

-now the pipeline is set to run with webhook, when the changes are done in the github code

**->Jenkins Master Slave Configuration:**

In Jenkins, a master-slave (agent) configuration allows you to distribute the build workload across multiple machines, where the master is the central Jenkins server that manages the overall Jenkins environment and the slaves (or agents) are additional machines that execute jobs on behalf of the master.

This configuration is useful when you have resource-intensive jobs, need to scale Jenkins, or want to run builds on different environments (e.g., different OS or configurations).

-The machine(server) in which you have installed the Jenkins is called as the Jenkins Master machine

-sometimes we won’t run the jobs in master machine for safety reasons, at that time the jobs will run in agent(slave) machines

-when multiple teams work on master machine then the load will increase on that machine and server will get down, for that each team will work with separate agent machine

-Reasons for configuring Agents:

i. Build distribution

ii. Security

iii. To maintain master machine always up and run

-Agent configuration

-Create a server and install Jenkins in it, then it will become the master machine

-then install the jdk in master machine: sudo apt update && apt install openjdk-17-jre -y

-then create another server for the slave machine

-then install the jdk in slave machine: sudo apt update && apt install openjdk-17-jre -y

-then user to root user

-then change the directory to /opt and make a directory with name Jenkins\_server in it: cd /opt and mkdir jenkins\_server (: mkdir -p /opt/jenkins\_server)

-then change the user owner and group owner to non-root user: chown -R ubuntu:ubuntu jenkins\_server

-now open the jenkins installed in the master machine and go to the "Manage Jenkins" and click on "Nodes" and click on "New Node"

-give the name for the node as "myagentmachine" and type as "Permanent Agent" and click on "create"

-then provide the required details for the Node

Description: for build distribution

Remote root directory: /opt/Jenkins\_slave/

Labels: slave "slave1" "slave2" "slave3"

Usage: Use this node as much as possible

Lauch method: Launch agents via SSH

Host: agent machine ip address

Credentials: Add credentials:

Domain: Global credentials

Kind: SSH user name with private key

scope: Global

ID: ubuntu

Description: agent user name and pemfile

Username: ubuntu

Private key: enter directly

key: copy and paste the pemfile of the keypair using for the agent machine and click on add

then select the added credentials

Host Key Verifying Strategy: Non verifying Verification strategy

Availability: Keep this agent as much as possible

and click on "Save"

-then open the agent from Nodes and click on launch agent

-after agent is in online

-create a job and configure with sample Hello world program

-in that update the agent to created agent with label

agent {

label "slave1"

}

-then save and apply the job

-then the job will run in the Agent(slave) machine

**Deploying the Java project in Docker with Jenkins Pipeline**

-First create the server

-then install the openjdk-17-jre, maven

-and also install and run the Jenkins and Docker

-configure the port numbers 22, 8080 and 8081 in security group for the server

now open the Jenkins with public ip and port number (8080)

-then login and configure jenkins

plugins: install required plugins, like: Docker, Eclipse

tools: configure the required tools, like: jdk, docker

-create a new pipeline and configure with sample hello world project

like: pipeline {

agent any

stages {

stage('Hello') {

steps {

echo 'Hello World'

}

}

}

}

-update the script with required tools

like: pipeline {

agent any

tools {

jdk 'jdk-11'

}

}

-then open "configuration" and select "pipeline syntax" in the "pipeline" tab

-and select the checkout option in sample steps

-then provide the required details

github url:

credentials:

branch:

-then click on "Generate pipeline Script" option

-now copy and paste that script in the pipeline script

like: pipeline {

agent any

tools {

jdk 'jdk-17'

}

stages {

stage('Git-Checkout') {

steps {

checkout scmGit(branches: [[name: '\*/main']], extensions: [], userRemoteConfigs: [[credentialsId: 'my-github', url: 'https://github.com/dinesh-4136/JavaWebCalculator.git']])

}

}

}

}

-then save and run the pipeline, then checkout stage will run

-then update the pipeline script to produce the artifact

like: pipeline {

agent any

tools {

jdk 'jdk-17'

}

stages {

stage('Git-Checkout') {

steps {

checkout scmGit(branches: [[name: '\*/main']], extensions: [], userRemoteConfigs: [[credentialsId: 'my-github', url: 'https://github.com/dinesh-4136/JavaWebCalculator.git']])

}

}

stage('Package') {

steps {

sh 'mvn package'

}

}

}

}

-then update the script to build the docker image

like: pipeline {

agent any

tools {

jdk 'jdk-17'

}

stages {

stage('Git-Checkout') {

steps {

checkout scmGit(branches: [[name: '\*/main']], extensions: [], userRemoteConfigs: [[credentialsId: 'my-github', url: 'https://github.com/dinesh-4136/JavaWebCalculator.git']])

}

}

stage('Package') {

steps {

sh 'mvn package'

}

}

stage('Build and Run Docker Image') {

steps {

sh 'docker build -t myapp1:v1 .'

sh 'docker run --name myapp1 -d -p 8081:8080 myapp1:v1'

}

}

}

}

-then check whether the application is running or not: 0.000.000.00(ip):8081(port)/webapp-0.1

**Jenkins Questions**

1. Explain about Jenkins pipeline/Jenkins stages?

2. Have you use declarative pipeline and scripted pipeline?

3. What is the difference between declarative and scripted pipeline?

4. Why do we need configure master and slave configuration?

5. How did you automate the build tools or CI/CD pipelines?

6. Suppose your pipeline got aborted (or) build is failed what are the steps you are going to take?

7. What is the difference between freestyle job and pipeline job?

8. Suppose your Jenkins master server is down what will you do?

9. Suppose your Jenkins jobs are taking too much time to build to run what will you do?

10. What is the difference between CI/CD-CD

11. Why we are using Jenkins?

12. Do you know any other method using CI/CD?

13. What are the different plugins you have used in Jenkins?

14. What is para-metered build in Jenkins?

15. How to troubleshoot when job fails?

16. What is upstream, downstream?

**Here are detailed answers to your Jenkins interview questions:**

**1. Explain Jenkins Pipeline / Jenkins Stages**

A Jenkins pipeline is an automated sequence of steps used to build, test, and deploy software.

Pipeline Structure

A pipeline consists of multiple stages, each representing a phase in the CI/CD process:

Example of Jenkins Pipeline with Stages

pipeline {

agent any

stages {

stage('Build') {

steps {

echo 'Building the application...'

sh 'mvn clean package'

}

}

stage('Test') {

steps {

echo 'Running tests...'

sh 'mvn test'

}

}

stage('Deploy') {

steps {

echo 'Deploying the application...'

sh 'scp target/\*.war user@server:/deploy/'

}

}

}

}

**2. Have you used Declarative and Scripted Pipelines?**

Yes. Jenkins supports two types of pipelines:

| Type | Description | Syntax |
| --- | --- | --- |
| Declarative Pipeline | Structured, easier to read and maintain | Uses pipeline {} block |
| Scripted Pipeline | More flexible, written in Groovy code | Uses node {} block |

**3. Difference Between Declarative and Scripted Pipeline**

| Feature | Declarative Pipeline | Scripted Pipeline |
| --- | --- | --- |
| Ease of Use | Simple, structured | Complex, full control |
| Syntax | pipeline {} | node {} |
| Error Handling | Limited | More control |
| Recommended For | Standard CI/CD | Advanced workflows |

Example of a Scripted Pipeline:

node {

stage('Checkout') {

git 'https://github.com/user/repo.git'

}

stage('Build') {

sh 'mvn clean package'

}

stage('Deploy') {

sh 'scp target/\*.war user@server:/deploy/'

}

}

**4. Why Do We Need Master-Slave Configuration?**

Jenkins Master-Slave architecture is used for load balancing and parallel execution.

Use Cases:

* Master handles job scheduling, UI, and plugin management.
* Slaves (Agents) execute jobs based on availability.

Setting Up Master-Slave in Jenkins

1. Go to Manage Jenkins → Manage Nodes
2. Add a new node (slave) and configure it.
3. Install Java and start the slave with:
4. java -jar agent.jar -jnlpUrl <Jenkins\_URL>

**5. How Did You Automate Build Tools or CI/CD Pipelines?**

* Used Jenkins Pipelines to automate build, test, and deployment.
* Integrated GitHub, Maven, Docker, and Kubernetes for full automation.
* Scheduled Builds using cron expressions:
* triggers {
* cron ('H/5 \* \* \* \*') // Runs every 5 minutes
* }

**6. What Steps Do You Take When a Build Fails or a Pipeline is Aborted?**

1. Check logs (Console Output) for error messages.
2. Verify the source code (broken commits, syntax errors).
3. Check dependencies (missing libraries, network issues).
4. Rebuild the project and verify tests.
5. Retry the pipeline manually or use:
6. retry(3) { sh 'mvn test' } // Retry 3 times if test fails

**7. Difference Between Freestyle Job and Pipeline Job**

| Feature | Freestyle Job | Pipeline Job |
| --- | --- | --- |
| UI-Based | Yes | No |
| Scripting Required | No | Yes (Groovy) |
| Flexibility | Limited | Highly Flexible |
| Usage | Basic builds | CI/CD automation |

**8. What If Jenkins Master Server is Down?**

1. Check Jenkins Logs:
2. tail -f /var/log/jenkins/jenkins.log
3. Restart Jenkins:
4. systemctl restart jenkins
5. Use Backup Server if High Availability (HA) is configured.

**9. What If Jenkins Jobs Take Too Long?**

1. Optimize Code & Dependencies (Reduce unnecessary steps).
2. Use Parallel Execution:
3. parallel {
4. stage('Build') { steps { sh 'mvn package' } }
5. stage('Test') { steps { sh 'mvn test' } }
6. }
7. Use Jenkins Slave Nodes for distributed builds**.**

**10. Difference Between CI/CD and CD**

| Concept | CI/CD | CD |
| --- | --- | --- |
| Continuous Integration (CI) | Automates code integration & testing | Ensures fast feedback |
| Continuous Delivery (CD) | Deploys code to a test/staging environment | Manual approval for production |
| Continuous Deployment (CD) | Deploys automatically to production | Fully automated |

**11. Why Do We Use Jenkins?**

* Automates CI/CD
* Integrates with Git, Docker, Kubernetes
* Supports parallel and distributed builds
* Large plugin ecosystem

**12. Other Methods for CI/CD?**

* GitHub Actions
* GitLab CI/CD
* CircleCI
* Azure DevOps
* TeamCity

**13. Plugins Used in Jenkins**

* Git Plugin (Integrates GitHub)
* Pipeline Plugin (For CI/CD automation)
* SonarQube Plugin (Code quality analysis)
* Docker Plugin (Containerized builds)
* Kubernetes Plugin (Deploy to K8s clusters)

**14. What Is a Parameterized Build in Jenkins?**

A parameterized build allows users to pass input values before running a job.

Example:

1. Go to Job Configuration → Check This project is parameterized.
2. Add a String Parameter (ENVIRONMENT).
3. Modify the pipeline:
4. pipeline {
5. parameters {
6. string (name: 'ENVIRONMENT', defaultValue: 'dev', description: 'Enter environment')
7. }
8. stages {
9. stage('Deploy') {
10. steps {
11. sh "echo Deploying to ${ENVIRONMENT}"
12. }
13. }
14. }
15. }

**15. How to Troubleshoot When a Job Fails?**

1. Check Console Output for errors.
2. Verify Git Branch & Merge Conflicts.
3. Check Logs & System Resources:
4. df -h # Disk usage
5. free -m # Memory usage
6. Rebuild with Debug Mode:
7. sh 'mvn clean install -X'

**16. What Are Upstream and Downstream Jobs?**

* Upstream Job: A job that triggers another job.
* Downstream Job: A job that gets triggered by another job.

Example:

post {

success {

build job: 'Deploy-Job'

}

}