**JENKINS**

1. **What is jenkins**
2. **Key Components of Jenkins**
3. **How Jenkins Works**
4. **Jenkins Architecture**
5. **Why we need Jenkins**
6. **Why Jenkins is Popular**
7. **What is CI/CD**
8. **Need for CI/CD**
9. **CI/CD Pipeline**
10. **Stages of CI/Cd pipeline**
11. **Benifits of CI/CD Pipeline**
12. **Types of Pipeline**
13. **Jenkins Environments**
14. **What we need Environemnts in Jenkins**

**What is JENKINS:**

* Jenkins is a tool that is used for automation.
* Jenkins is an open-source automation server
* Jenkins supports continuous integration (CI) and continuous delivery (CD).
* It allows developers to automatically build, test, and deploy their applications, making the development process more efficient and reliable.
* It is written in Java and runs on java only.
* Port number is 8080

**Key Components of Jenkins**

1. **Jenkins Server:** The core of Jenkins, responsible for managing jobs, plugins, security, credentials, nodes, and global settings.
2. **Jenkins Jobs:** Tasks that Jenkins performs, such as building code, running tests, and deploying applications.
3. **Jenkins Plugins:** Extensions that add new features and functionalities to Jenkins. There are thousands of plugins available for various purposes.
4. **Jenkins Nodes/Agents:** Machines that run jobs. The Jenkins server can distribute jobs across multiple nodes to balance the load and speed up the process.
5. **Jenkins Web Interface:** A user-friendly interface that allows users to configure and manage Jenkins, view job statuses, and access logs.

**How Jenkins Works**

1. **Source Code Management:** Jenkins integrates with various version control systems like Git, SVN, and Mercurial to fetch the latest code
2. **Build Triggers:** Jenkins can be configured to trigger builds automatically based on events like code commits, scheduled times, or manual triggers.
3. **Build Execution:** Jenkins executes the build process, which can include compiling code, running tests, and packaging the application.
4. **Post-Build Actions:** After the build, Jenkins can perform additional tasks like deploying the application, sending notifications, or archiving artifacts.

**Jenkins Architecture**

Jenkins follows a master-agent architecture:

* **Master:** The Jenkins server that manages jobs, schedules builds, and distributes tasks to agents.
* **Agent:** Machines that execute the jobs assigned by the master. Agents can be on the same machine as the master or on different machines.

**Why we need Jenkins:**

1. **Automation:** Automates repetitive tasks, reducing manual effort and errors.
2. **Scalability:** Can handle large projects and distribute workloads across multiple agents.
3. **Extensibility:** Thousands of plugins available to extend Jenkins' capabilities.
4. **Community Support:** A large and active community that contributes to the development and improvement of Jenkins.
5. **Continuous Integration:** Jenkins enables continuous integration by automatically building and testing code changes as they are committed to the repository. This ensures that code is always in a deployable state.
6. **Continuous Deployment/Delivery:** Jenkins can automate the deployment process, allowing you to continuously deliver updates to your application. You can configure it to deploy to different environments, such as development, staging, and production.

**Why Jenkins is Popular**

* **Open Source:** Jenkins is free to use and has a vibrant open-source community that continuously contributes to its development and improvement.
* **Extensibility:** Jenkins has a vast ecosystem of plugins (over 1,800) that extend its functionality to support various tools, languages, and workflows. This makes it highly customizable and adaptable to different project needs.
* **Automation:** Jenkins automates repetitive tasks such as building, testing, and deploying code, which significantly reduces manual effort and errors. This makes the software development process more efficient and reliable.
* **Ease of Use:** Jenkins provides a user-friendly web interface that simplifies the configuration and management of jobs and pipelines. It also supports various triggers like code commits, scheduled builds, and manual builds.
* **Scalability:** Jenkins can handle projects of all sizes, from small projects to large-scale enterprise applications. Its master-agent architecture allows for load balancing and distributed builds, making it highly scalable.
* **Continuous Integration and Continuous Delivery (CI/CD):** Jenkins is widely used for implementing CI/CD pipelines, which helps in integrating code changes frequently, testing them automatically, and deploying them seamlessly. This accelerates the software delivery process and ensures higher code quality.
* I**ntegration with Version Control Systems:** Jenkins integrates with popular version control systems like Git, SVN, and Mercurial, allowing it to fetch and build code from repositories automatically.
* **Active Community and Support:** Jenkins has a large and active community of users and contributors who provide support, share knowledge, and create plugins. This ensures that any issues are quickly addressed and new features are continuously added.
* **Flexibility:** Jenkins supports a wide range of environments, including different operating systems (Windows, Linux, macOS), cloud services (AWS, Azure, GCP), and containerization technologies (Docker, Kubernetes).

**Types of Jenkins:**

**1. Freestyle Jenkins Jobs:**

* Definition: The simplest form of Jenkins job, where each build is defined by a series of steps.
* Use Case: Ideal for straightforward tasks that do not require complex configurations.

**Parameterised Jobs**

* Parameterised jobs in Jenkins allow you to pass parameters to the job when it is triggered. These parameters can control the behavior of the job, making it more flexible and dynamic

Example:

1. String Parameter: Specify a branch name or version number.
2. Boolean Parameter: Enable or disable certain features or steps.
3. Choice Parameter: Select from a predefined list of options.

**Sequential Jobs**

* Sequential jobs in Jenkins are jobs that run one after the other in a predefined order. Each job waits for the previous job to complete before starting.
* This is useful when you have a series of tasks that need to be executed in a specific order. Sequential jobs can be implemented using pipelines or by setting up build triggers that start the next job after the current one finishes.

**Parallel Jobs**

* Parallel jobs in Jenkins are jobs that run simultaneously. This is useful for speeding up the build process by executing independent tasks at the same time.
* In a pipeline, you can use the parallel directive to define stages that should run in parallel. For example, you might run different types of tests (unit, integration, performance) in parallel to save time.

**2. Pipeline Jenkins Jobs:**

* Definition: Allows users to define the entire build process as code using a domain-specific language called Groovy.
* Use Case: Suitable for complex workflows, enabling continuous integration and continuous delivery pipelines. Provides better control over build processes and versioning of pipeline code.

**CI/CD**

**Continuous Integration (CI)**

* Continuous Integration (CI) is a software development where developers regularly merge their code changes into a shared repository, typically multiple times a day.
* Each merge triggers an automated build and testing process to detect integration issues early.
* CI aims to improve software quality and reduce the time it takes to deliver new features.
* Popular CI tools include Jenkins, GitLab CI, Travis CI, and CircleCI.

**Continuous Delivery (CD)**

* Continuous Delivery (CD) is an extension of CI that focuses on automating the deployment process so that code changes can be released to production at any time.
* It ensures that the codebase is always in a deployable state.
* CD aims to make software deployment a routine and reliable process.
* Popular CD tools include Jenkins, GitLab CI/CD, Spinnaker, and Bamboo.

**Need for CI/CD:**

* CI/CD stands for Continuous Integration and Continuous Deployment/Delivery. It's a set of practices in software development that aims to improve the process of integrating code changes and deploying them to production.

**why CI/CD is essential:**

1. **Faster Development Cycles:** By integrating code changes frequently, teams can identify and fix issues early, reducing the time between development and deployment.
2. **Improved Code Quality:** Automated testing is a key component of CI/CD. It ensures that code changes are thoroughly tested before they are integrated, resulting in higher-quality software.
3. **Reduced Manual Work:** Automation in CI/CD reduces the need for manual intervention in the build, test, and deployment processes, freeing up developers to focus on writing code.
4. **Consistent Deployments:** Automation ensures that deployments are consistent and repeatable, reducing the likelihood of errors that can occur with manual deployments.
5. **Greater Collaboration:** CI/CD encourages collaboration among team members by providing a shared platform for integrating and testing code changes.
6. **Early Issue Detection:** By continuously integrating and testing code, teams can detect and address issues early in the development process, reducing the risk of bugs reaching production.
7. **Scalability:** CI/CD pipelines can scale with the needs of the project, allowing teams to handle larger and more complex codebases efficiently.

**CI/CD PIPELINES**

**Pipeline:**

* It is a code & it has a set of stages & stages contains steps will execute entire pipeline

**Continuous Integration (CI) Pipeline:**

* A Continuous Integration (CI) pipeline is an automated process that facilitates the integration of code changes from multiple developers into a shared repository, ensuring that new code is built, tested, and validated before being integrated into the main codebase.
* The CI pipeline streamlines the process of integrating code changes by automating various stages of the software development lifecycle.
* It helps catch integration issues early, maintain code quality, and ensure that the codebase remains stable and deployable.

**Benefits of a CI Pipeline**

* **Early Detection of Issues**: By running automated tests and analysis early in the development process, the CI pipeline helps catch and fix issues before they reach production.
* **Improved Code Quality:** Continuous integration and testing ensure that code changes are thoroughly validated, leading to higher-quality software.
* **Faster Development Cycles:** Automation reduces manual intervention, speeding up the development process and enabling more frequent code releases.
* **Consistent Builds:** The CI pipeline ensures that builds are consistent and reproducible, reducing the risk of errors and inconsistencies.

**Continuous Deployment (CD) Pipeline:**

* A Continuous Deployment (CD) pipeline is an automated process that takes the code changes that have passed through the Continuous Integration (CI) pipeline and deploys them to production or other environments without manual intervention.
* It ensures that software is always in a deployable state and can be released to users at any time.
* The CD pipeline extends the CI process by automating the deployment and delivery of software.
* It focuses on ensuring that the code is always ready for production and that deployments are reliable, consistent, and repeatable. By automating the deployment process, the CD pipeline reduces the risk of human error and speeds up the release cycle.

**Benefits of a CD Pipeline**

* **Faster Releases:** Automation enables quick and frequent releases, reducing the time to market for new features and updates.
* **Improved Quality:** Continuous testing and monitoring ensure that only high-quality code is deployed to production.
* **Consistency:** Automated deployments provide consistent results, reducing the risk of deployment failures and configuration drift.
* **Reduced Manual Intervention:** By automating the deployment process, the CD pipeline minimizes the potential for human error and increases efficiency

**Stages of a Pipeline:(Stages wil be multiple)**

1. Git clone url (git checkout)
2. Build --> Generate war files
3. Deploy --> We need to copy files to server(Tomcat)

**Steps in pipeline:(Same steps applied)**

1. Git clone url(git checkout)
2. Build -->Commands mvn clean package
3. Deploy -->war file needs to be copied to server(Tomcat)

**Steps or Algorithnm for jenkins pipeline:**

1. Start jenkins
2. Install "pipeline" plugins
3. Create new item --> pipeline1 --> type pipeline --> ok
4. Create or get jenkinsfile in pipeline section --> pipeline scriot --> select the script

**Types of pipeline:**

**Groovy:** Programming language that writes scripts to automate tasks.

**Declarative Pipeline**

* It is defined using a human-readable ,specify what steps to execute rather than how to execute them.
* Consists of a single pipeline block with stages and steps,starts with pipeline keyword.

**Example:**

Jenkinsfile1

pipeline {

agent any

stages {

stage('Build') {

steps {

echo 'Building...'

}

}

stage('Test') {

steps {

echo 'Testing...'

}

}

stage('Deploy') {

steps {

echo 'Deploying...'

}

}

}

}

**Example:**

pipeline {

agent any

stages {

stage('Build') {

steps {

echo 'Building...'

}

}

}

}

**Scripted Pipeline:**

* Jenkins pipeline defined using a Groovy-based scripting language in a jenkinsfile,specify exact steps and execution details
* Uses a node block with arbitrary Groovy code(starts with node keyword)

**Example:**

node {

stage('Build') {

echo 'Building...'

}

stage('Test') {

echo 'Testing...'

}

stage('Deploy') {

echo 'Deploying...'

**Jenkins Environments :**

* In real-time Jenkins environments, you typically have multiple stages or environments to ensure smooth deployment and testing. Here are some common environments used in Jenkins pipelines:

1. **Development (DEV):** This is where developers integrate and test their code. It's the first stage where initial builds and unit tests are run.
2. **Testing/Quality Assurance (QA):** After the development stage, the code moves to the QA environment. Here, more extensive testing is performed, including integration tests, functional tests, and sometimes performance tests.
3. **User Acceptance Testing (UAT):** This environment is used for user acceptance testing. It's a pre-production environment where end-users or stakeholders validate the functionality and performance of the application.
4. **Staging:** The staging environment is a replica of the production environment. It's used for final testing before the code is deployed to production. This stage ensures that everything works as expected in an environment that closely mimics the live environment.
5. **Production (PROD):** This is the live environment where the application is accessible to end-users. The code that passes all previous stages is deployed here.

**What we need Environemnts in Jenkins :**

* It seems like you want to understand why we need different environments in Jenkins. Let's explore that in more detail:

1. **Quality Assurance:** Different environments allow for different levels of testing and quality assurance. For instance, the development environment is used for unit tests, while the QA environment is used for integration tests and other in-depth testing. This helps in catching bugs early and ensuring the quality of the code before it reaches production.
2. **Consistency:** Environments ensure that the code runs consistently across various stages. By having development, staging, and production environments, you can be sure that what works in development will also work in production.
3. **Isolation:** Each environment acts as an isolated sandbox where changes can be made and tested without affecting other environments. This isolation is crucial for debugging and testing new features without risking the stability of the live application.
4. **Configuration Management**: Different environments allow you to manage different configurations needed for various stages. For example, you might have a different database for the development environment compared to the production environment. This helps in maintaining proper configurations and reducing errors during deployment.
5. **Automated Deployment:** Environments in Jenkins allow for automated deployments through the CI/CD pipeline. This automation speeds up the deployment process and ensures that code changes are deployed in a controlled and systematic manner.
6. **Rollback Strategy:** In case of issues in the production environment, having different environments allows for a rollback strategy. You can easily revert to a previous stable version without impacting the entire application.
7. **Collaboration:** Different environments facilitate better collaboration among development, QA, and operations teams. Each team can work in their respective environments, ensuring smooth and efficient workflow.

**Plugins:**

* Additional/External features added to our project called plugins.
* In order to install plugins we need to utilize dashboard
* If already exists then go to

jenkins dashboard-->plugins-->type or search plugins

**Steps to install plugins:**

Access Jenkins Dashboard-->Manage Jenkins-->Manage Plugins-->Available Tab-->Search and Select Plugins-->Install-->-->Wait for Installation-->Restart Jenkins

**Build Triggers:**

Build triggers determine when to start a new build of a Jenkins job. They help automate the CI/CD pipeline by defining conditions under which builds are initiated.

**Poll SCM (Source Code Management):**

Polling SCM allows Jenkins to check the version control system for changes at regular intervals. If changes are detected, it triggers a build.

* **Configuration:** In the job configuration, select "Poll SCM" and specify the schedule using a cron-like syntax.

**Build Periodically:**

This trigger allows you to schedule builds at specific intervals, irrespective of whether there are changes in the source code.

**Configuration**:

* In the job configuration, select "Build periodically" and use the cron-like syntax to define the schedule.

**Webhooks:**

Webhooks are used to trigger builds in Jenkins when an event occurs in another service, such as a push to a GitHub repository.

**Configuration:**

* In GitHub, go to the repository settings, select "Webhooks," and add a new webhook with the Jenkins URL.
* In Jenkins, configure the job to use the appropriate trigger (e.g., "GitHub hook trigger for GITScm polling").

**Cron Syntax:**

Jenkins uses a cron-like syntax to define schedules for polling SCM and periodic builds. Here’s a quick overview of the format:

\* \* \* \* \*

| | | | |-------- Day of the week (0 - 7) (0 and 7 are Sunday)

| | | |---------Month (1 - 12)

| | |------------Day of the month (1 - 31)

| |------------Hour (0 - 23)

|--------------Minute (0 - 59)

**Examples:**

* H/15 \* \* \* \* : Poll every 15 minutes.
* 0 2 \* \* 1-5 : Build at 2 AM from Monday to Friday.