Week 6 - Continuous Integration and Continuous Delivery

- Understanding aggregate functions (SUM, AVG, COUNT)
- Continuous Integration/Continuous Deployment (CICD) Workflow
- Overview
- Understanding JenkinsFile: Exploring Jenkins Jobs, Jenkins
- Pipeline, and Jenkins File
- Build Triggers in Jenkins: SCM Polling, GITScm Polling, Build
- Periodically - Uncovering the What, Why, and How
- Groovy File Creation: Conceptualizing and Crafting Groovy Scripts
- for Jenkins
- Integrating GitHub with Jenkins: Establishing Webhooks for
- Seamless Collaboration
- Grasping Merge Request Concepts in the CI/CD Process
- Jenkins Master-Slave Configuration: Optimizing Resource
- Utilization in CI/CD
- Triggering Pipelines Directly from JenkinsFile: Streamlining
- Automation
- GitLab Branching Strategy: Best Practices for Code Collaboration
- and Version Control
- Hands-On Experience: Building a Jenkins CI Pipeline with Groovy,
- Incorporating Various Stages
- Configuring Docker Engineer as the Jenkins Slave - Jenkins
- Dynamic Slave Configuration
- Jenkins Plugins - Docker, Git, Maven and other common used
- plugins
- Integrating Kubernetes with jenkins

—----------

Continuous Integration and Continuous Delivery

Continuous Integration (CI):

Continuous Integration is a software development practice where developers frequently integrate code changes into a shared repository. Each integration is verified by automated build and test processes, allowing teams to detect errors early in the development cycle. Key aspects of CI include:

- Frequent Integration: Developers integrate code changes into a shared repository multiple times a day, ensuring that changes are integrated and tested continuously.

- Automated Build and Test: CI systems automatically build the application from the latest codebase and run tests to validate its functionality. This ensures that changes do not introduce regressions or bugs.

- Early Feedback: CI provides early feedback to developers about the quality of their code changes, allowing them to fix issues promptly and avoid integration conflicts with other team members' changes.

- Integration Pipeline: CI systems typically implement an integration pipeline, which consists of various stages such as build, test, and deployment. Each stage verifies different aspects of the codebase, ensuring its integrity.

Continuous Delivery (CD):

Continuous Delivery is an extension of Continuous Integration where code changes are automatically deployed to production-like environments after passing through the CI pipeline. The goal of CD is to ensure that software can be released reliably at any time. Key aspects of CD include:

- Automated Deployment: CD automates the deployment process, allowing teams to release software quickly and reliably. Deployment pipelines are defined to promote changes through various environments, such as development, testing, staging, and production.

- Infrastructure as Code (IaC): CD often involves the use of Infrastructure as Code (IaC) tools to automate the provisioning and configuration of infrastructure environments. This ensures consistency and reproducibility across different environments.

- Release Orchestration: CD tools provide release orchestration capabilities to manage the release process, coordinate deployments, and rollback changes if necessary. This helps reduce the risk associated with releasing new features or updates.

- Continuous Feedback: CD provides continuous feedback to stakeholders by monitoring application performance, user feedback, and other metrics. This feedback loop helps teams identify areas for improvement and prioritize future development efforts.

Jenkins in CI/CD:

Jenkins is a popular open-source automation server used for implementing CI/CD pipelines. It provides a wide range of plugins and integrations with other tools and technologies, making it highly flexible and customizable. With Jenkins, teams can automate build, test, and deployment processes, enabling Continuous Integration and Continuous Delivery practices.

Jenkins Installation

- Create Jenkins container from jenkins official image - *#docker run -p 8080:8080 -p 50000:50000 -dit --name jenkins --restart=on-failure -v jenkins_home:/var/jenkins_home jenkins/jenkins:lts-jdk17*

- Read the initial admin password for Jenkins - *#docker exec jenkins cat /var/lib/jenkins/secrets/initialAdminPassword*

Jenkins Jobs

A Jenkins job defines a specific task or process within a CI/CD pipeline.
A Jenkins job is a unit of work or a task that Jenkins performs as part of a Continuous Integration (CI) or Continuous Delivery (CD) pipeline. It encapsulates a set of instructions or steps that Jenkins executes to automate various aspects of the software development lifecycle

- Configuration: A Jenkins job is configured with parameters such as source code repository URL, build triggers, build steps, post-build actions, and notification settings.

- Build Steps: Jobs consist of a series of build steps or tasks that Jenkins executes sequentially. These steps typically include compiling code, running tests, packaging artifacts, and deploying applications.

- Integration with Tools: Jenkins jobs integrate with a wide range of tools and technologies through plugins. This allows for seamless integration with version control systems, build tools, testing frameworks, deployment platforms, and other DevOps tools.

- Triggers: Jobs can be triggered manually by users or automatically based on predefined triggers such as code commits, scheduled intervals, or external events.

- Parameters: Jenkins jobs can accept parameters, enabling users to customize job behavior at runtime. Parameters may include branch names, build versions, configuration options, and more.

- Environment: Jenkins provides an isolated environment for each job execution, ensuring consistency and reproducibility of build dependencies and configurations.

- Monitoring and Reporting: Jobs generate build logs, test reports, and other artifacts that provide insights into the build process. These artifacts are essential for monitoring build progress, troubleshooting issues, and analyzing build results.

Jenkins Triggers:

Jenkins triggers are mechanisms that initiate the execution of Jenkins jobs or pipelines based on specific events or conditions. These triggers allow automation of the build, test, and deployment processes, ensuring that tasks are executed in response to relevant events.

- Poll SCM: This trigger checks the source code repository for changes at regular intervals (polling), such as every few minutes. If changes are detected in the repository since the last build, the Jenkins job is triggered to start a new build.

- Build after other projects are built: This trigger starts a build of a Jenkins job when one or more specified upstream projects have completed their builds successfully.
  It allows for defining dependencies between different Jenkins jobs, ensuring that downstream jobs are triggered automatically after the successful completion of upstream jobs.

- Build periodically: This trigger schedules builds to run at specific time intervals using cron-like syntax. Users can specify the frequency of builds (e.g., every hour, daily at a specific time) to automate regular builds at predefined intervals.

- GitHub hook trigger for GITScm polling: This trigger listens for webhook notifications from GitHub repositories. When changes are pushed to the GitHub repository, GitHub sends a webhook notification to Jenkins, triggering a build of the associated Jenkins job. To create a GitHub webhook, please click on settings -> webhooks -> and then add the webhook(http://JENKINS_IP:8080/github-webhook/)

Jenkins Pipeline:

Creating a pipeline in Jenkins involves defining the steps and stages of your software delivery process as code. Jenkins Pipeline is a suite of plugins that allows you to express your pipeline in a declarative or scripted format, stored alongside your project's source code. We can create our pipeline with Jenkins Groovy DSL language.

Steps to Create a Pipeline:

- Choose Pipeline Type: Decide whether you want to create a Declarative Pipeline or a Scripted Pipeline. Declarative Pipeline offers a simpler, more opinionated syntax, while Scripted Pipeline provides more flexibility with Groovy scripting.

- Create a Jenkinsfile: In your project's repository, create a file named Jenkinsfile.
  This file will contain the definition of your pipeline, including stages, steps, and other pipeline configurations.

- Define Pipeline Stages: Define the stages of your pipeline, representing different phases of the software delivery process such as build, test, and deploy. Each stage can consist of one or more steps, which are individual tasks or actions performed as part of that stage.
- Configure Pipeline Steps: Define the steps to be executed within each stage of your pipeline. Steps can include actions like compiling code, running tests, packaging artifacts, deploying applications, and sending notifications.

- Define Pipeline Triggers: Optionally, define triggers for your pipeline to specify when it should be executed. Triggers can be based on events such as code commits, pull requests, scheduled intervals, or external webhook notifications.

- Commit Jenkinsfile to Repository: Commit your Jenkinsfile to your project's repository alongside your source code. Jenkins will automatically detect the Jenkinsfile in your repository and use it to create and execute your pipeline.

- Configure Jenkins Job: In the Jenkins web interface, create a new pipeline job.
  Specify the location of your project's repository and the branch containing the Jenkinsfile. Jenkins will use the Jenkinsfile to define and execute your pipeline when the job is triggered.

Create A Sample Jenkins Pipeline From Jenkinsfile

- Download Stage: This stage downloads the Python code from a GitHub repository.
- Test Stage: Installs necessary dependencies, runs tests on the Python code using pytest, and confirms successful testing.
- Build Docker Image Stage: Builds a Docker image named "mywebimg" using the Dockerfile in the downloaded code repository.
- Deploy in PROD Stage: Checks if a Docker container named "webos" is already running. If it is, the existing container is stopped and removed. Then, a new container named "webos" is launched using the previously built Docker image, exposed on port 81.

Jenkinsfile -

```
pipeline {
   agent{
      label "ec2"
   }

   stages {
      stage('Download') {
         steps {
            git branch: 'main', url:
https://github.com/sudhanshuvlog/PythonAppCI-CD.git'
            echo "Code downloaded"
         }
      }
      stage('Test') {
         steps {
            sh "yum install python3-pip -y"
            sh "pip3 install -r requirements.txt"
            sh "pytest"
            echo "Code have been testes succesfully!"
         }
      }
      stage("Build Docker Image")
         {
            steps {
               sh "docker build -t mywebimg ."
```

```
            }
        }
    stage("Deploy in PROD") {
        steps {
            script {
                def isContainerRunning = sh(script: "docker ps -q -f name=webos",
returnStatus: true)
                if (isContainerRunning == 0) {
                    sh "docker rm -f webos"
                }
                sh "docker run -dit --name webos -p 81:80 mywebimg"
            }
        }
    }

}
```

Creating Credentials in Jenkins:

- Navigate to Credentials: Log in to your Jenkins dashboard and go to "Manage Jenkins" > "Manage Credentials".
- Add Credentials: Click on "Global credentials (unrestricted)" or a domain where you want to add credentials, then click on "Add Credentials".
- Select Credential Choose the credential type as “username password” for storing sensitive data.
- Fill in Details: Fill in the required details such as username, password, SSH key, or secret text based on the selected credential type.
- Save: Click "OK" or "Save" to create the credentials.

Using Credentials in Pipeline:

In the below pipeline, We are authentication with Dockerhub, and then we are building and pushing a docker image from jenkins.

```
pipeline {
    agent {
        label 'ec2'
    }
```

```
    stages {
        stage('Build & Tag Docker Image') {
            steps {
                script {
                    dir('src') {

                    withDockerRegistry(credentialsId: 'docker-cred', toolName: 'docker') {
                        sh "docker build -t jinny1/cartservice:latest ."
                    }
                        }
                }
            }
        }

        stage('Push Docker Image') {
            steps {
                script {
                    withDockerRegistry(credentialsId: 'docker-cred', toolName: 'docker') {
                        sh "docker push jinny1/cartservice:latest "
                    }
                }
            }
        }
    }
}
```

Jenkins-Master Slave Architecture:

Jenkins Master-Slave architecture is a distributed setup where a central Jenkins Master node manages and delegates tasks to multiple Jenkins Slave nodes. This architecture allows for scalability, improved performance, and the ability to execute jobs concurrently on multiple nodes. Here's a brief overview:

- Jenkins Master: The Jenkins Master node is the central server responsible for managing and coordinating the build and deployment process. It stores configuration information, manages jobs, and schedules builds. The Master node communicates with Slave nodes to distribute workload and execute jobs.

- Jenkins Slave: Jenkins Slave nodes are worker machines that execute build jobs delegated by the Master node. Slaves can run on different platforms and environments, allowing for parallel execution of builds across various

configurations. Slaves may be physical machines or virtual instances, and they can be located on-premises or in the cloud.

Setting up an EC2 Instance as a Jenkins Slave Node:

- Launch EC2 Instance: Launch a new EC2 instance in the AWS console, choosing an appropriate instance type and Amazon Machine Image (AMI) based on your requirements. Ensure that the security group associated with the instance allows inbound connections on the port used by Jenkins for communication (typically port 22 for SSH).

- Install Java and Jenkins Agent: Install Java Runtime Environment (JRE) on the instance, as Jenkins requires Java to run.
    - *#wget https://download.oracle.com/java/17/archive/jdk-17.0.10_linux-x64_bin.rpm*
    - *#yum install jdk-17.0.10_linux-x64_bin.rpm -y*

- Configure Jenkins Master: In the Jenkins web interface, navigate to the "Manage Jenkins" section and then to "Manage Nodes and Clouds". Add a new node and configure it as a "Permanent Agent". Provide details such as the node name, remote root directory, and launch method (typically "Launch agent via execution of command on the Master").

Jenkins Dynamic Slave Configuration With Docker

Using Jenkins with dynamic Docker slaves enhances build efficiency, scalability, and consistency.

Follow the below steps to setup Dynamic Slave:

- Create a Docker Image: Build a Docker image with the necessary dependencies and environment configurations for your Jenkins jobs.
  Push this image to a centralized registry like Docker Hub, Use the Dockerfile mentioned below for building the Docker image

- Configure Jenkins Master:

- Install Docker Plugin: Ensure the Docker plugin is installed on your Jenkins master.

- Go to the Docker Engine - *#vim /usr/lib/systemd/system/docker.service* And add -> -**H tcp://0.0.0.0:4243** in the row which starts with "ExecStart"

- Configure Docker Cloud:
  Go to Manage Jenkins > Configure System.
  Scroll down to Cloud and click Add a new cloud > Docker.
  Provide the Docker host URL (e.g., tcp://localhost:2376) and configure security settings as needed.

- Set Up Docker Agent Templates:

  - Add Docker Template: In the Docker section, add a Docker template.

  - Specify Docker Image: Enter the name of the Docker image from your registry

- Configure Jenkins Job:

  - Assign Labels: Ensure your Jenkins job is configured to use the label assigned to the Docker template.
  - When a job is triggered, Jenkins will: Automatically create a Docker container using the specified image. Run the build inside the dynamically created container.

- Run and Monitor Jobs:
  - Trigger a Jenkins job and observe that a new Docker container is instantiated for the job.
  - The job runs in isolation within this container, ensuring a clean and consistent environment.

Why do we need a dynamic agent/slave?

Using Jenkins dynamic slaves via Docker offers several advantages:

- Scalability: Docker allows Jenkins to dynamically provision and de-provision build environments (slaves) based on the current workload. This helps in efficiently utilizing resources.

- Isolation: Each build runs in its isolated Docker container, ensuring that builds do not interfere with each other. This is particularly useful when dealing with different dependencies or configurations.

- Consistency: Docker containers ensure that the build environment is consistent across different builds and developers' machines, reducing the "works on my machine" problem.

- Efficiency: Containers are lightweight and start quickly compared to virtual machines, leading to faster provisioning of build environments and reduced overhead.

- Clean Environments: Using Docker, each build can start with a clean slate, preventing issues caused by leftover artifacts from previous builds.

- Flexibility: Jenkins can use different Docker images for different projects or stages of the CI/CD pipeline, allowing for tailored environments suited to specific needs.

Use the below Dockerfile to build an Jenkins slave image.

```
FROM centos:7
#if it takes time to build docker image or it got stuck while building, then use "amazonlinux:latest" as the base image
# Install necessary packages
RUN yum update -y
RUN yum install -y git wget openssh-server
#shadow-utils will provide us the "chpasswd" command, which we will use to change root password
RUN yum install -y shadow-utils-2:4.9-12.amzn2023.0.2.x86_64
RUN yum install python3 -y
# Download and install Oracle JDK
RUN wget https://download.oracle.com/java/17/archive/jdk-17.0.8_linux-x64_bin.rpm
RUN yum install -y jdk-17.0.8_linux-x64_bin.rpm

# Set root password
RUN echo 'root:redhat' | chpasswd

# Configure SSH
RUN ssh-keygen -A

# Define the default command to run when the container starts
CMD ["/usr/sbin/sshd", "-D"]
```

- Run a Jenkins Pipeline on Dynamic Slave Agent, Use below Jenkinsfile

```
pipeline {
   agent {dockerContainer 'jinny1/jenkinsslave'}

   stages {
      stage('Hello') {
         steps {
            echo 'Hello World'
         }
      }
   }
}
```

Maven

The Jenkins Maven Plugin integrates Jenkins with Apache Maven, a widely-used build automation tool primarily for Java projects. This plugin enables Jenkins to build, test, and deploy Maven-based projects.

Key Features:

- Automatic Installation: The plugin can automatically install Apache Maven on the Jenkins agent if it's not already available.

- Maven Project Configuration: Jenkins can be configured to recognize Maven projects by specifying the location of the project's pom.xml file.

- Build Triggers: Builds can be triggered automatically based on SCM changes (e.g., Git commits) or scheduled at specific intervals.

- Build Environment: Maven-specific environment variables can be configured for each build, ensuring consistency across different executions.

- Build Goals and Options: Jenkins can execute Maven goals and pass additional options such as profiles or properties during the build process.

- Test Reporting: Jenkins can parse and display test reports generated by Maven, providing insights into test results and coverage.

Checkout a sample project Below

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                // Checkout the source code from your version control system
git branch: 'main', url: 'https://github.com/Trainersudhanshu/sample-java-app.git'
            }
        }

        stage('Build') {
            steps {
                // Use Maven to build the project
                sh 'mvn clean install'
            }
        }

        stage('Test') {
            steps {
                // Run tests using Maven
                sh 'mvn test'
            }
        }
    }

    post {
        // Define post-build actions, such as notifications or cleanup tasks
        success {
            // Actions to perform if the pipeline succeeds
            echo 'Pipeline succeeded! Project built and tested successfully.'
        }
        failure {
            // Actions to perform if the pipeline fails
            echo 'Pipeline failed! Check build logs for errors.'
        }
    }
}
```

In this pipeline:

- We define three stages: Checkout, Build, Test.
- In the Checkout stage, we clone the source code from the repository.
- In the Build stage, we use Maven (mvn clean install) to build the project.

- In the Test stage, we run tests using Maven (mvn test).
- Finally, we define post-build actions to execute based on the pipeline result (success or failure).

GitHub Actions

Seamless Integration: GitHub Actions seamlessly integrates with your GitHub repositories, allowing you to define workflows directly within your codebase.

No Infrastructure Management: There's no need to manage infrastructure like EC2 instances or Jenkins servers. GitHub handles the underlying infrastructure, simplifying the setup process.

Easy Configuration: Workflows are defined using YAML files within your repository, making it easy to version control and collaborate on CI/CD configurations.

Event-Driven Triggers: GitHub Actions triggers workflows based on various events such as pushes, pull requests, issue comments, and more, ensuring your CI/CD pipeline responds dynamically to repository changes.

GitHub Actions Workflow

- Workflow A GitHub Actions Workflow is an automated process designed to handle tasks such as building, testing, packaging, releasing, or deploying projects within your repository.

- We have to create .github/workflows directory in our repository and we have to write the workflow in a file called YAML file.

- Utilize triggers to initiate the Workflow. Triggers can include events such as pushes, pull requests, comments, or custom events based on your project's requirements.

Example of GitHub Actions Workflow

```
name: CI
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
jobs:
```

```
build:
  runs-on: ubuntu-latest

  steps:
  - uses: xxxxxxxxxxxxxxxx@xx
  - name: Set up JDK 11
  uses: xxxxxxxxxxxxxxxxxx@xx
  with:
    java-version: '11'
  - name: Build with Maven
  run: mvn -B package --file pom.xml
```

GitHub Actions Hosted Runners

- GitHub Actions Hosted Runners are virtual machines that are hosted by GitHub.
- We can use the GitHub Actions Hosted Runners to run our CI/CD pipeline.
- It's just like using Jenkins with EC2 instances (slave nodes).

SonarQube

- SonarQube is a code quality analysis tool that provides detailed reports on code quality metrics.
- It produces comprehensive reports highlighting various aspects of code quality.
- SonarQube stores report data in a database for easy accessibility and reference.
- Sonar Scanner is used to scan code and send analysis reports to the SonarQube server, ensuring continuous monitoring of code quality.

How to install SonarQube?

- We can use Docker Image to run SonarQube.
- We can use the following command to run SonarQube on our local machine.

  docker run -d --name sonarqube -p 9000:9000 sonarqube:latest

- We can access the SonarQube server using the following URL.
  http://localhost:9000

- We can use the default username and password to log in to the SonarQube server.
  username: admin
  password: admin

Login to SonarQube using the default credentials.
Username: admin
Password: admin

Mergify

Mergify is a tool that allows you to automatically merge your pull requests when they are ready. It is a great tool to use when you have a lot of pull requests and you want to automate the process of merging them.

How to install Mergify?

We can use the following steps to install Mergify in our GitHub repository.

- Go to the Mergify website and sign in with your GitHub account.
- Once you are signed in, you can install Mergify in your GitHub repository.
- After installing Mergify, you can configure it to automatically merge your pull requests when they are ready.
- You can also configure Mergify to automatically close your pull requests when they are not ready to be merged.
- You can also configure Mergify to automatically label your pull requests when they are ready to be merged.
- You can also configure Mergify to automatically assign your pull requests to the person who is responsible for merging them.

Mergify Rules

- We can write the rules in a file called .mergify.yml and we can configure Mergify to automatically merge our pull requests based on the rules we have written in the .mergify.yml file.

Example of Mergify Rules

```
pull_request_rules:
  - name: Automatically merge pull requests
    conditions:
      - base=main
      - label=ready-to-merge
    actions:
      merge:
        method: merge
```

Understanding aggregate functions (SUM, AVG, COUNT)

Aggregate functions in databases, such as SUM, AVG, and COUNT, are used to perform calculations on sets of values within a database table. Here's a brief explanation of each:

- **SUM**: This function calculates the total sum of all the values in a specific column. It is commonly used to find the total of numerical values, such as sales revenue or quantity sold.

  SELECT SUM(sales_amount) AS total_sales FROM sales;

- AVG: AVG calculates the average value of a set of numerical values in a column. It adds up all the values and then divides the sum by the total number of values.

  SELECT AVG(sales_amount) AS average_sales FROM sales;

- COUNT: COUNT returns the number of rows in a specified column or the total number of rows that meet a given condition. It can count all rows or rows that satisfy certain criteria.

  SELECT COUNT(*) AS total_records FROM users;

  SELECT COUNT(*) AS active_users FROM users WHERE last_login_date >= '2023-01-01';

These aggregate functions are powerful tools for data analysis and reporting in SQL. They allow users to summarize and derive insights from large datasets efficiently.