PW SKILLS | DevOps and Cloud Computing

Advanced Jenkins

Objective

- Manage files, plugins, and jobs in Jenkins.
- Understand upstream and downstream job dependencies.
- Apply CI best practices for efficient pipeline execution.
- Scale Jenkins workloads to improve performance and reliability

PW SKILLS

Explain Jenkins job types (Freestyle, Pipeline, Multibranch).

Jenkins Job Type

Jenkins supports several job types, each suited to specific use cases in continuous integration and deployment workflows. Below is a summary of the key job types:

1. Freestyle Project

- Description: A basic and versatile job type that allows you to define simple build steps.
- Use Cases: Execute shell or batch commands, integrate with source code management (SCM), trigger builds, and run build tools like Ant, Maven, or Gradle.
- Flexibility: Ideal for simple automation tasks but lacks advanced features like scripting or branching logic.

Jenkins Job Type

2. Pipeline

- Description: A more advanced job type that uses a script (written in Groovy) to define the entire build process as code.
- Use Cases: Automating complex workflows, including building, testing, deploying, and integrating with tools like Docker.
- Key Feature: Uses a Jenkinsfile stored in the project's SCM to define the pipeline steps, enabling version control of the build process.
- Advantages: Supports sequential and parallel execution, error handling, and reusable code.

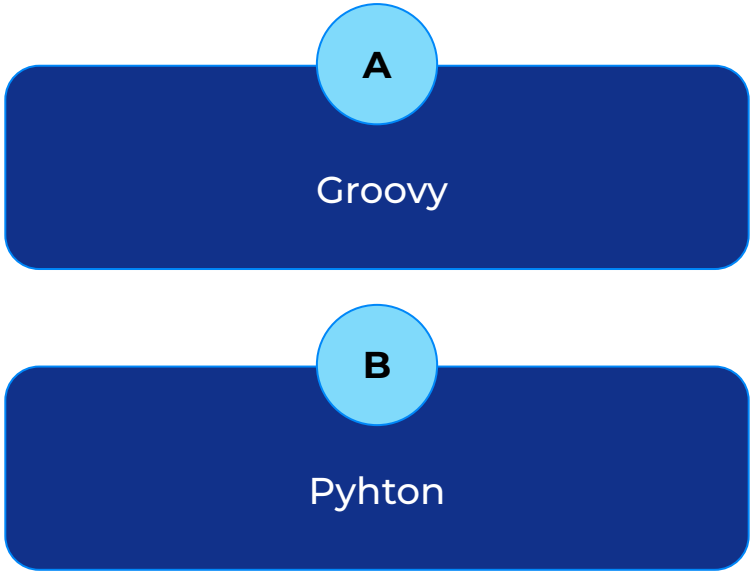
Jenkins Job Type

3. Multibranch Pipeline

- Description: A specialized pipeline job that automatically creates and manages pipelines for all branches in a repository.
- Use Cases: Useful for projects with multiple branches where each branch has its own Jenkinsfile.
- Key Feature: Automatically detects new branches and applies the pipeline configuration without manual intervention.
- Advantages: Simplifies CI/CD for repositories with multiple active development branches.

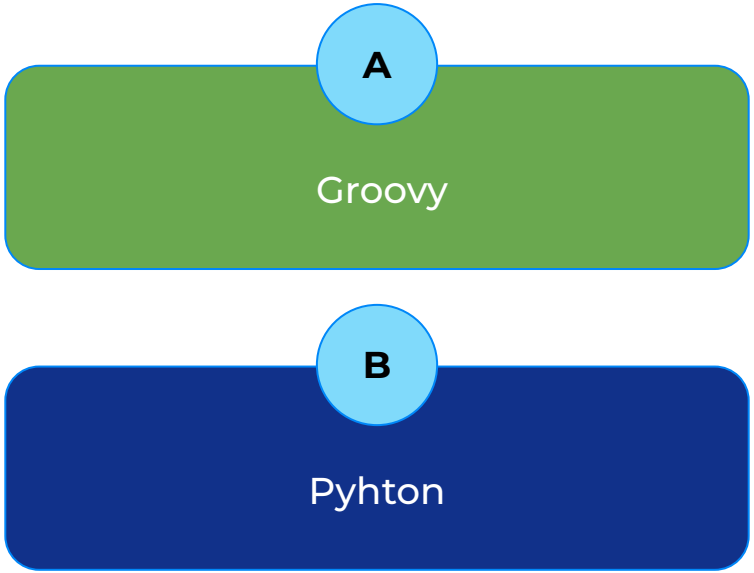
Pop Quiz

Q. What scripting language is used to define a Jenkins Pipeline?



Pop Quiz

Q. What scripting language is used to define a Jenkins Pipeline?



PW SKILLS

Discuss Jenkins file structure and configuration files (config.xml)

Let’s Discuss

Jenkins uses specific files and structures to manage its configurations and pipelines. Here's a concise overview.

Jenkinsfile

Purpose: Defines the pipeline as code using Groovy-based Domain-Specific Language (DSL).

Structure:

- Pipeline Block
- Agent
- Stages
- Steps
- Post Actions

Formats:

- Declarative: Structured and user-friendly, recommended for most use cases.
- Scripted: Offers flexibility but is less readable for complex pipelines.

Configuration Files (config.xml)

- Found in various directories under ‘JENKINS_HOME’.

- **Global Configurations:**

Located at ‘JENKINS_HOME/config.xml’.

Stores system-wide settings like security, plugins, and global environment variables.

- **Job-Specific Configurations:**

Each job has its own ‘config.xml’ in ‘JENKINS HOME/jobs/[JOBNAME]/config.xml’.

Defines job-specific settings like triggers, SCM configurations, and build steps.

- **Folder Configurations:**

Folders have their own ‘config.xml’ to manage nested jobs.

PW SKILLS

Introduce plugins and their role in extending Jenkins functionality

Plugins & their Role

Plugins in Jenkins are extensions that enhance its core functionality, making it highly adaptable for diverse CI/CD workflows. Here's a brief overview:

Role of Plugins in Jenkins:

1. Integration
2. Customization
3. Automation
4. Ease of Use

Examples of Popular Plugins:

- Git Plugin
- Pipeline Plugin
- Docker Plugin
- Kubernetes Plugin

PW SKILLS

Demonstrate managing plugins via the Jenkins UI

Let’s do it

Managing plugins via the Jenkins UI is straightforward and can be done through the "Manage Jenkins" > "Plugins" page. Here's a brief overview of the process:

1. Installing Plugins
2. Updating Plugins
3. Uninstalling Plugins
4. Disabling Plugins
5. Manually Uploading Plugins

PW SKILLS

Explain job dependencies and how to chain jobs

Job Dependencies

- Definition: Dependencies ensure that one job runs after another or only if a preceding job succeeds.
- Use Cases: Building, testing, and deploying applications in sequential steps.

Chaining Jobs

1. Using "Build Other Projects"
2. Using Pipelines:

   Define dependencies and chaining directly in a ‘Jenkinsfile’ using pipeline syntax.

Example:

```
pipeline {
    stages {
        stage('Build') {
            steps {
                echo 'Building...'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing...'
            }
        }
    }
}
```

3. Parameterized Trigger Plugin
4. Multibranch Pipelines

PW SKILLS

Demonstrating setting up: Upstream & Downstream jobs

Let’s do it

Here’s a concise demonstration of setting up upstream and downstream jobs in Jenkins:

1. Upstream Job: Triggered by Another Job

Steps:

1. Create a new job (e.g., "Upstream Job").
2. Configure the job with build steps (e.g., shell commands or scripts).
3. Save the configuration.

Let’s do it

2. Downstream Job: Runs After Upstream Job

Steps:

1. Create another job (e.g., "Downstream Job").
2. In the "Upstream Job" configuration:

- Go to Post-build Actions.
- Select Build other projects.
- Enter the name of the downstream job (e.g., "Downstream Job").
- Optionally, set conditions (e.g., trigger on success).

Save the configuration.

3. Passing Parameters

4. Viewing Dependencies

Pop Quiz

Q. What is an upstream job in Jenkins?

A

A job triggered after another job completes

B

A job that triggers other jobs based on its status

Pop Quiz

Q. What is an upstream job in Jenkins?

A

A job triggered after another job completes

B

A job that triggers other jobs based on its status

Pop Quiz

Q. What is a downstream job in Jenkins?

A

A job triggered by an upstream job's complication

B

A job that triggers other jobs

Pop Quiz

Q. What is a downstream job in Jenkins?

A

A job triggered by an upstream job's complication

B

A job that triggers other jobs

Take A 5-Minute Break!

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

PW SKILLS

Discuss CI best practices

CI Best Practices

1. Keeping Pipelines Modular and Reusable

- Break pipelines into smaller, reusable components (e.g., separate build, test, and deploy stages) to simplify maintenance and improve flexibility.
- Use shared libraries or templates for common tasks to avoid duplication.

2. Reducing Build Times with Caching

- Implement caching mechanisms for dependencies, artifacts, and Docker layers to avoid redundant downloads or builds.
- Tools like Jenkins, GitLab CI, and CircleCI support caching to speed up builds.

CI Best Practices

3. Parallelizing Builds for Efficiency

- Split tests or tasks into parallel jobs to utilize multiple agents or runners, reducing overall pipeline execution time.
- Example: Run unit tests, integration tests, and UI tests concurrently.

4. Implementing Automated Rollbacks on Failure

- Configure pipelines to automatically revert deployments if a failure is detected during post-deployment checks.
- Use tools like Kubernetes or feature flags to enable quick rollbacks without downtime.

PW SKILLS

Demonstrating pipeline optimization techniques

Let’s do it

Here’s a demonstration of optimization techniques using both Scripted and Declarative Pipelines:

1. Parallelizing Stages

Goal: Reduce execution time by running independent tasks concurrently.

Declarative Pipeline Example:

```
pipeline {
    agent any
    stages {
        stage('Parallel Tasks') {
            parallel {
                stage('Build') {
                    steps {
                        echo 'Building...'
                    }
                }
                stage('Test') {
                    steps {
                        echo 'Testing...'
                    }
                }
            }
        }
    }
}
```

Let’s do it

Scripted Pipeline Example:

```
node {
    parallel(
        build: {
            echo 'Building...'
        },
        test: {
            echo 'Testing...'
        }
    )
}
```

2. Caching Artifacts

Goal: Save time by reusing previously built artifacts or dependencies.

Let’s do it

Declarative Pipeline Example:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                stash name: 'build-artifacts',
includes: '**/target/*.jar'
            }
        }
        stage('Deploy') {
            steps {
                unstash 'build-artifacts'
                echo 'Deploying...'
            }
        }
    }
}
```

Let’s do it

Scripted Pipeline Example:

```
node {
    stage('Build') {
        stash name: 'build-artifacts', includes: '**/target/*.jar'
    }
    stage('Deploy') {
        unstash 'build-artifacts'
        echo 'Deploying...'
    }
}
```

3. Conditional Execution

Goal: Skip unnecessary stages based on conditions.

Let’s do it

Declarative Pipeline Example:

```
pipeline {
    agent any
    stages {
        stage('Conditional Stage') {
            when { branch 'main' }
            steps {
                echo 'Running on main branch...'
            }
        }
    }
}
```

Scripted Pipeline Example:

```
node {
    if (env.BRANCH_NAME == 'main') {
        stage('Conditional Stage') {
            echo 'Running on main branch...'
        }
    }
}
```

Let’s do it

4. Incremental Builds

Goal: Rebuild only the changed components to save time.

Declarative Pipeline Example:

```
pipeline {
    agent any
    stages {
        stage('Incremental Build') {
            steps {
                script {
                    if (fileExists('changed-files.txt')) {
                        echo 'Building only changed files...'
                    } else {
                        echo 'Full build required.'
                    }
                }
            }
        }
    }
}
```

Pop Quiz

Q. What is a key advantage of using declarative pipelines over scripted pipelines?

A

Simpler syntax and better readability for maintainability

B

Greater flexibility for complex workflows

Pop Quiz

Q. What is a key advantage of using declarative pipelines over scripted pipelines?

A

Simpler syntax and better readability for maintainability

B

Greater flexibility for complex workflows

PW SKILLS

Explain why Jenkins needs scaling for large workloads

Let’s See

Jenkins needs scaling for large workloads to ensure performance, efficiency, and reliability in CI/CD pipelines. Here's why:

1. Handling Increased Workloads
2. Reducing Bottlenecks
3. Improving Resilience
4. Optimizing Resource Utilization

PW SKILLS

Discuss strategies to scale Jenkins

Strategies

1. Master-Agent Architecture for Distributed Builds

- Description: This architecture separates the Jenkins controller (master) from the build execution nodes (agents). The master handles job scheduling, while agents perform the builds.
- Benefits: Allows horizontal scaling by adding more agents as needed, reducing the load on the master and improving parallelism.
- Implementation: Use cloud services like AWS or Azure to provision agents dynamically.
-

Strategies

2. Using Kubernetes and Jenkins Agents for Dynamic Scaling

- Description: Kubernetes enables dynamic scaling of Jenkins agents based on workload demands. Jenkins can automatically adjust the number of pods running agents.
- Benefits: Ensures optimal resource utilization and handles peak loads efficiently.
- Implementation: Use the Jenkins Kubernetes plugin to integrate with Kubernetes clusters.

3. Running Jenkins in Docker Containers for Isolated Environments

- Description: Containerizing Jenkins allows for isolated environments that are easy to manage and scale.
- Benefits: Provides immutable build environments, simplifies agent setup, and supports rapid deployment.
- Implementation: Use Docker to create Jenkins images and manage containers for build agents.

PW SKILLS

Demonstrate setting up a Jenkins agent node for workload distribution

Let’s do it

Steps to Set Up a Jenkins Agent Node for Workload Distribution:

1. Prepare the Agent Machine:
2. Add a New Node in Jenkins UI
3. Configure the Node
4. Set Launch Method
5. Start the Agent
6. Verify Connection

PW SKILLS

Time for Case Study

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this session
- Review your lecture notes
- Note down questions and queries regarding this session and consult the teaching assistants

Thanks !