Jenkins Scripted Pipelines - Detailed Notes

# Overview

Scripted Pipeline is one of the two types of Jenkins pipelines (the other being Declarative). It uses Groovy-based syntax and provides maximum control over flow logic, making it ideal for advanced users who need fine-grained customization.

# Key Characteristics

- Language: Groovy-based scripting DSL

- Structure: node { ... } block

- Flexibility: Very high (can write complex logic)

- Error-prone: More than Declarative due to lack of checks

- IDE Support: Limited, but extensible with Groovy plugins

# Basic Structure

node {

stage('Stage Name') {

// Steps here

echo 'Hello, World!'

}

}

# Example: Full Scripted Pipeline

node {

stage('Checkout') {

git url: 'https://github.com/example/repo.git'

}

stage('Build') {

echo 'Building...'

sh './build.sh'

}

stage('Test') {

echo 'Testing...'

sh './run\_tests.sh'

}

stage('Deploy') {

echo 'Deploying...'

sh './deploy.sh'

}

}

# Core Concepts

1. Node Block: Allocates an executor on a Jenkins agent.

2. Stage Block: Used for visual representation and logical separation.

3. Steps: Atomic operations like echo, sh, checkout, etc.

4. Control Flow: Uses if, for, try-catch, etc., for logic control.

# Common Functions

- sh: Runs a shell script on Unix agents

- bat: Runs a batch script on Windows agents

- echo: Prints to Jenkins console

- checkout: Checks out from SCM

- git: Shortcut to checkout Git repositories

- sleep: Pauses execution

- input: Waits for user input

- parallel: Runs multiple branches in parallel

# Parallel Execution

node {

stage('Parallel Stage') {

parallel(

"Branch A": {

echo 'Running A'

},

"Branch B": {

echo 'Running B'

}

)

}

}

# Error Handling

node {

try {

stage('Build') {

sh './build.sh'

}

} catch (Exception e) {

echo "Build failed: ${e.message}"

currentBuild.result = 'FAILURE'

} finally {

echo 'Cleaning up...'

}

}

# Best Practices

- Keep logic simple – avoid deeply nested structures

- Use shared libraries for reusable code

- Limit use of node blocks to reduce overhead

- Always handle errors with try-catch

- Separate configuration from logic for clarity

# Scripted vs Declarative

Scripted Pipeline:

- Syntax: Groovy

- Flexibility: High

- Readability: Lower

- Error Prone: Higher

- Suitability: Complex pipelines

Declarative Pipeline:

- Syntax: DSL (opinionated)

- Flexibility: Moderate

- Readability: Higher

- Error Prone: Lower

- Suitability: Standard CI/CD workflows

# When to Use Scripted Pipelines

- Complex conditional logic

- Dynamic parallelism

- Integration with complex legacy systems

- Need for Groovy scripting capabilities