

Introduction to Jenkins

Module 3: Jenkins Jobs and Pipeline-as-Code



Topics

- Freestyle vs. Pipeline Jobs
- Declarative Pipeline basics
- Scripted vs Declarative Pipeline
- Pipeline libraries and shared steps



Jenkins Jobs

- A Jenkins job defines
 - What type of job needs to be run
 - When it should run
 - Where it should run
- Jenkins supports multiple job types
- Two basic types
 - Freestyle jobs
 - Pipeline jobs

New Item

Enter an item name

Select an item type



Pipeline

Build, test, and deploy using pipelines. Supports stages, parallel work, and running on multiple agents.



Freestyle project

Classic, general-purpose job type that checks out from up to one SCM, executes build steps serially, followed by post-build steps like archiving artifacts and sending email notifications.



Multi-configuration project

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.



Folder

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.



Multibranch Pipeline

Creates a set of Pipeline projects according to detected branches in one SCM repository.



Organization Folder

Creates a set of multibranch project subfolders by scanning for repositories.



Freestyle Jobs

- This is the legacy model
- Configured entirely through the Jenkins UI
- Step-by-step build actions configured in the UI
 - Easy to start with
 - Hard to scale and maintain
- Typical Use
 - Simple builds
 - Older Jenkins installations
 - One-off tasks

New Item

Enter an item name

Select an item type



Pipeline

Build, test, and deploy using pipelines. Supports stages, parallel work, and running on multiple agents.



Freestyle project

Classic, general-purpose job type that checks out from up to one SCM, executes build steps serially, followed by post-build steps like archiving artifacts and sending email notifications.



Multi-configuration project

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.



Folder

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.



Multibranch Pipeline

Creates a set of Pipeline projects according to detected branches in one SCM repository.



Organization Folder

Creates a set of multibranch project subfolders by scanning for repositories.



Freestyle Jobs

- Plugins extend functionality
 - Usually associated with a specific tool
 - *For example, Maven*
 - Adds additional configuration options for that type of project that are needed for the tool
 - For example
 - *Specifying the POM.xml file to be used in a Maven project*

New Item

Enter an item name

» This field cannot be empty, please enter a valid name

Select an item type



Pipeline

Build, test, and deploy using pipelines. Supports stages, parallel work, and running on multiple agents.



Freestyle project

Classic, general-purpose job type that checks out from up to one SCM, executes build steps serially, followed by post-build steps like archiving artifacts and sending email notifications.



Maven project

Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.



Multi-configuration project

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.



Folder

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.



Multibranch Pipeline












Freestyle Jobs

- Limitations
 - Configuration lives only in Jenkins
 - Hard to version-control
 - Poor support for complex workflows
 - Difficult to review changes
- Conflicts with IaC and DevOps principles
- Freestyle jobs are easy to create but hard to manage



Configure

-  General
-  Source Code Management
-  Triggers
-  Environment
-  Pre Steps
-  Build
-  Post Steps
-  Build Settings
-  Post-build Actions



Pipeline Jobs

- Modern Model
- Jobs are defined using code (pipelines)
 - Stored in source control
 - Written in Groovy-based syntax
 - Supports complex workflows
- Benefits
 - Version-controlled
 - Reviewable
 - Reproducible
 - Scalable
 - Better security and governance
- Moves Jenkins from a UI tool into a programmable automation engine

```
pipeline {
    agent any

    tools {
        // Install the Maven version configured as "M3" and add it to the path.
        maven "M3"
    }

    stages {
        stage('Build') {
            steps {
                // Get some code from a GitHub repository
                git 'https://github.com/jglick/simple-maven-project-with-tests.git'

                // Run Maven on a Unix agent.
                sh "mvn -Dmaven.test.failure.ignore=true clean package"

                // To run Maven on a Windows agent, use
                // bat "mvn -Dmaven.test.failure.ignore=true clean package"
            }
        }

        post {
            // If Maven was able to run the tests, even if some of the test
            // failed, record the test results and archive the jar file.
            success {
                junit '**/target/surefire-reports/TEST-*.xml'
                archiveArtifacts 'target/*.jar'
            }
        }
    }
}
```



Declarative Pipeline Basics

- A pipeline is
 - A sequence of stages
 - *The stages can be freely defined by the pipeline author*
 - *Executed on one or more agents*
 - *Defined as code*
 - *Managed by Jenkins*
- Pipeline stages usually describe
 - Build
 - Test
 - Package
 - Deploy workflows
 - Can be whatever we want them to be

```
1 pipeline {  
2     agent any  
3  
4     stages {  
5         stage('Whoa') {  
6             steps {  
7                 echo 'Howdy'  
8             }  
9         }  
10    }  
11 }
```



Declarative Pipeline Basics

- Core declarative pipeline concepts
 - pipeline {} – root block
 - agent – where the pipeline runs
 - stages – logical phases
 - steps – actions inside stages
 - post – actions after execution
- Enforce structure and consistency
 - Compiles to Groovy scripts

```
pipeline {  
    agent any  
  
    stages {  
        stage('Build') {  
            steps {  
                echo 'Building the application'  
            }  
        }  
  
        stage('Test') {  
            steps {  
                echo 'Running tests'  
            }  
        }  
    }  
  
    post {  
        always {  
            echo 'Pipeline finished'  
        }  
    }  
}
```



Scripted Pipelines

- Both pipeline types use Groovy
 - The difference is in the programming model and rules that Jenkins enforces
- Scripted Pipeline are written as Groovy scripts
 - Gives full control of flow and logic
 - Like a general-purpose program

```
def services = ['auth', 'billing', 'orders']
def buildResults = [:] // Map to store results

node('linux') {

    stage('Initialize') {
        echo "Starting build on node: ${env.NODE_NAME}"
        echo "Services to build: ${services}"
    }

    stage('Build Services') {

        for (service in services) {

            try {
                echo "Building service: ${service}"

                // Simulate a build command
                sh "echo Building ${service}"

                buildResults[service] = 'SUCCESS'

            } catch (Exception e) {
                echo "Build failed for ${service}: ${e.message}"
                buildResults[service] = 'FAILED'
            }
        }
    }
}
```



Scripted Pipelines

- Characteristics
 - Very flexible
 - Imperative style (“do this, then that”)
 - Few guardrails
 - Easy to shoot yourself in the foot
- Scripted pipelines are used for
 - Complex dynamic behavior
 - Advanced logic
 - Custom orchestration
 - Legacy pipelines written before declarative pipelines were introduced

```
def services = ['auth', 'billing', 'orders']
def buildResults = [:] // Map to store results

node('linux') {

    stage('Initialize') {
        echo "Starting build on node: ${env.NODE_NAME}"
        echo "Services to build: ${services}"
    }

    stage('Build Services') {

        for (service in services) {

            try {
                echo "Building service: ${service}"

                // Simulate a build command
                sh "echo Building ${service}"

                buildResults[service] = 'SUCCESS'

            } catch (Exception e) {
                echo "Build failed for ${service}: ${e.message}"
                buildResults[service] = 'FAILED'
            }
        }
    }
}
```



Declarative Pipelines

- Characteristics
 - Follow a defined structure
 - Use a constrained syntax
 - Are validated by Jenkins before running
 - Opinionated
 - *They limit the things you can do*
 - Safer
 - More readable
 - Easier to standardize
 - Built-in best practices

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building'
            }
        }

        stage('Test') {
            steps {
                echo 'Testing'
            }
        }
    }
}
```



Comparison

Aspect	Scripted Pipeline	Declarative Pipeline
Syntax	Free-form Groovy	Structured DSL
Style	Imperative	Declarative
Validation	Runtime	Pre-execution
Error detection	Later	Earlier
Readability	Lower	Higher
Guardrails	Minimal	Strong

Declarative pipelines describe what should happen

Scripted pipelines describe how it happens



Safety and Governance

- Scripted pipelines
 - Can run arbitrary Groovy
 - Can access Jenkins internals
 - Higher security risk
 - Harder to audit
- Declarative pipelines
 - Restricted execution model
 - Safer sandboxing
 - Easier to govern
 - Better suited for enterprises

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



Reason for Declarative Pipelines

- Scripted pipelines came first but users ran into issues when using them
 - Inconsistent pipelines
 - Hard-to-read logic
 - Security concerns
 - Difficult onboarding
 - Difficult for UI visualization
- Declarative pipelines were introduced to
 - Standardize structure
 - Improve safety
 - Make pipelines more readable
 - Enable features like:
 - *post*
 - *options*
 - *environment*
 - *when*

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



Mixing the Two

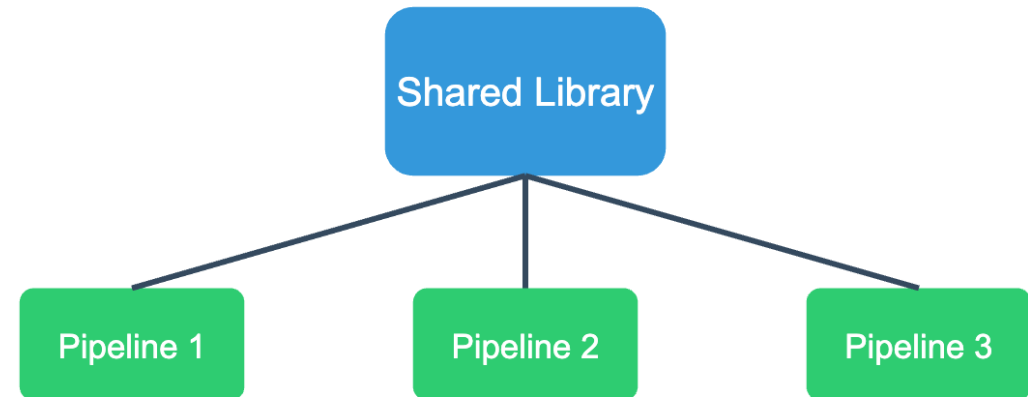
- Declarative pipelines can include scripted logic inside script {} blocks
- This gives
 - Declarative structure
 - Scripted flexibility where needed
 - Best practice
 - *Declarative pipeline + small scripted sections.*
- Declarative pipelines are the recommended default
- Scripted pipelines are a tool for advanced cases

```
stage('Conditional Logic') {  
    steps {  
        script {  
            if (env.BRANCH_NAME == 'main') {  
                echo 'Main branch'  
            }  
        }  
    }  
}
```



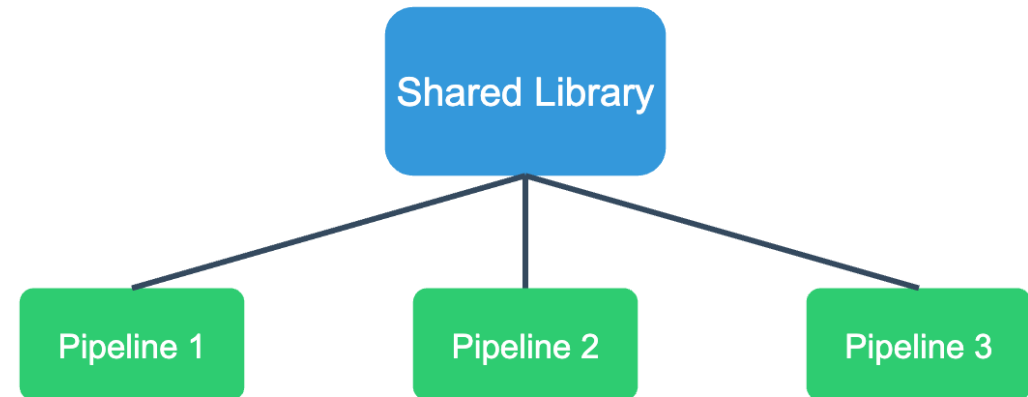
Pipeline Libraries and Shared Steps

- A shared library
 - Stored as a separate Git repository
 - Contains reusable pipeline code
 - Is loaded dynamically by Jenkins
 - Can be used across many pipelines
- Shared libraries allow teams to
 - Reuse logic
 - Standardize behavior
 - Reduce duplication
 - Centralize best practices



Typical Content

- Common functionality in shared libraries
 - Build logic (Maven, Gradle, npm)
 - Test execution
 - Code quality checks
 - Security scans
 - Artifact publishing
 - Deployment logic
 - Notification handling
 - Error handling and retries



Architecture

- Libraries contain only groovy code
- They do not contain declarative scripting code
 - Declarative scripts provide a pipeline structure
 - Not appropriate for callable functionality in the library
- Libraries have a standardized structure
- Jenkins knows how to load them

```
(root)
+-- src                                # Groovy source files
|   +-- org
|       +-- example
|           +-- Jenkins.groovy
+-- vars
|   +-- log.groovy
|   +-- buildApp.groovy
+-- resources                          # resource files (external libraries etc.)
    +-- org
        +-- example
            +-- config.json
```



Using Libraries

- In the example
 - The “jenkins-common” library is imported using the *@Library* annotation
 - The `_` (underscore) at the end is necessary when not immediately referencing a class from the library
 - The functions defined in the shared library, such as `log.info` and `buildApp()` can now be used

```
@Library('jenkins-common@master') _  
  
pipeline {  
    agent any  
    stages {  
        stage('Build') {  
            steps {  
                script {  
                    log.info "Starting build process"  
                    buildApp()  
                }  
            }  
        }  
    }  
}
```



Shared Steps

- Shared steps are:
 - Reusable functions
 - Written in Groovy
 - Exposed to pipelines as simple method calls
 - Instead of writing complex logic in every Jenkinsfile, pipelines just call a shared step
 - *RunTests()*
 - *BuildArtifact()*
 - *deployToEnv('prod')*
 - The actual implementation is defined in the shared library.

```
@Library('jenkins-common@master') _

pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                script {
                    log.info "Starting build process"
                    buildApp()
                }
            }
        }
    }
}
```



Shared Step Example

- The shared step code is stored in the vars directory
 - Anything under vars/ becomes a callable step in a Jenkinsfile

```
(shared-library-repo)
└─ vars/
    └─ runMavenBuild.groovy
```

```
def call(Map config = [:]) {

    // Default values
    def goals = config.get('goals', 'clean test')
    def skipTests = config.get('skipTests', false)

    echo "Running Maven goals: ${goals}"
    echo "Skip tests: ${skipTests}"

    def command = "mvn ${goals}"

    if (skipTests) {
        command += " -DskipTests"
    }

    sh command
}
```



Shared Step Example

- The code example shows using the step code in a pipeline
- At runtime
 - Jenkins loads the shared library
 - Finds vars/runMavenBuild.groovy
 - Exposes call() as runMavenBuild()
 - Executes the Groovy code on the agent
- Usage
 - call() makes the step feel “built-in”
 - Parameters allow customization
 - Logic lives in one place
 - Changes affect all pipelines consistently

```
@Library('company-shared-lib') _

pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                runMavenBuild(
                    goals: 'clean package',
                    skipTests: true
                )
            }
        }
    }
}
```



Questions

