Introduction to Jenkins

Module 4: Pipeline Deep Dive

Topics

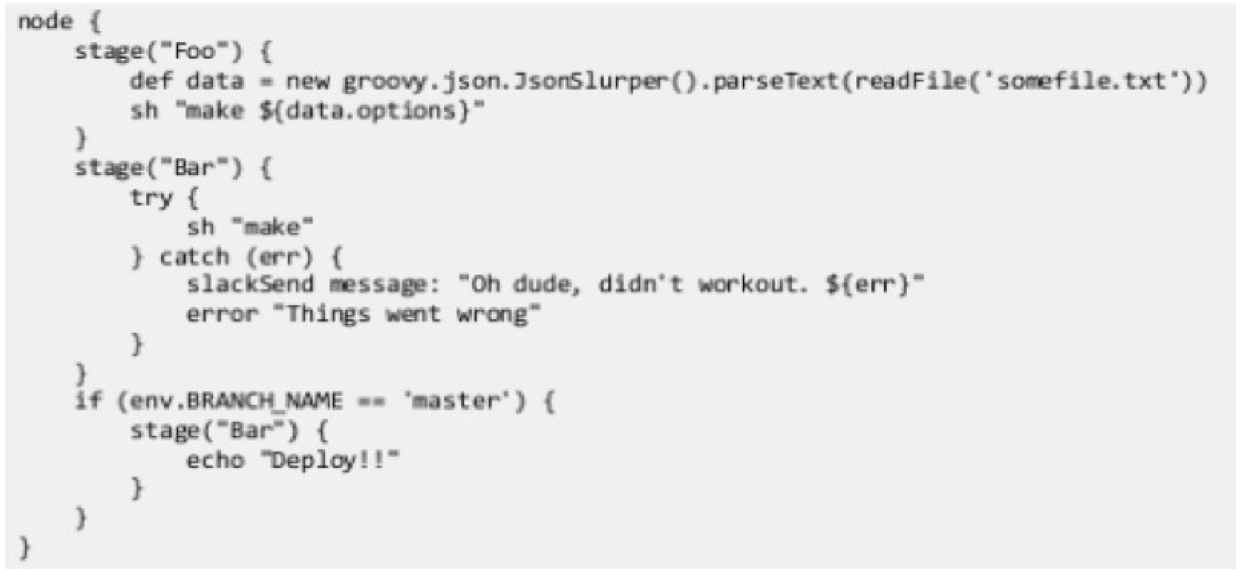
- Scripted vs Declarative Pipeline
- Stages, agents, environment blocks
- Parallel execution and matrix builds
- Docker agents within pipelines

Scripted vs Declarative Pipeline

- Both pipeline types use Groovy
  - They differ in structure and intent:
- Scripted pipeline
  - Jenkins runs the script as a Groovy program
- Declarative pipeline
  - Jenkins interprets the code as describing a structured workflow
  - Translated into executable Groovy code

Scripted Pipeline

- Characteristics
  - Free-form Groovy
  - Imperative style (“do this, then that”)
  - Very flexible and powerful
  - Few guardrails
- Used when there is a need for
  - Highly dynamic behavior
  - Complex branching logic
  - Advanced orchestration
  - Legacy pipelines
- Downside
  - Powerful but harder to read, govern, and secure



```
node {
    stage("Foo") {
        def data = new groovy.json.JsonSlurper().parseText(readFile('somefile.txt'))
        sh "make ${data.options}"
    }
    stage("Bar") {
        try {
            sh "make"
        } catch (err) {
            slackSend message: "Oh dude, didn't workout. ${err}"
            error "Things went wrong"
        }
    }
    if (env.BRANCH_NAME == 'master') {
        stage("Bar") {
            echo "Deploy!!"
        }
    }
}
```

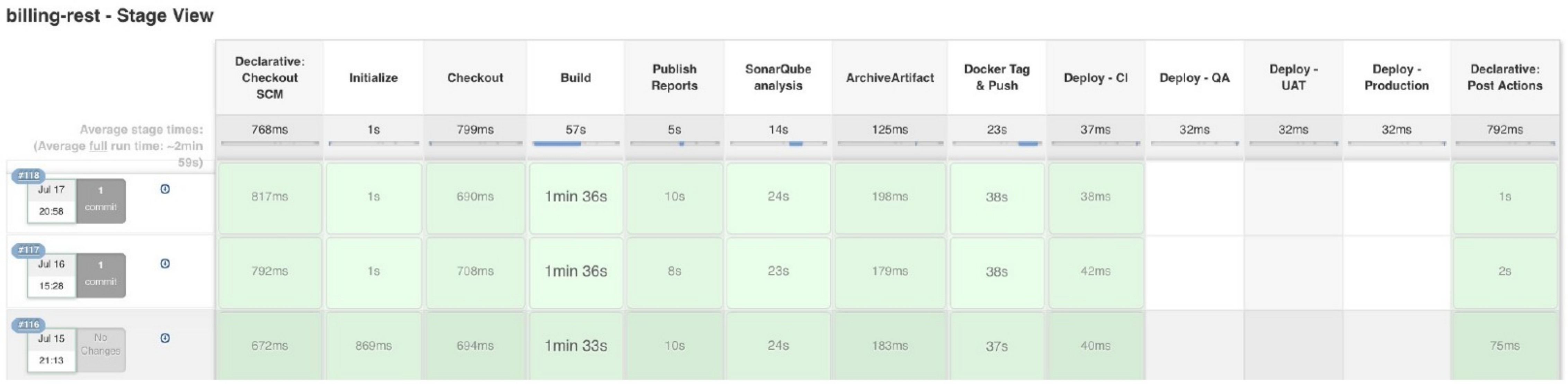
Declarative Pipeline

- Characteristics
  - Structured DSL (Domain Specific Language)
  - Opinionated and constrained
  - Validated before execution
  - Easier to read
  - Easy to develop standardized templates
- Advantages
  - Safer defaults
  - Better visualization
  - Easier onboarding
  - Better governance

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'mvn clean package'
            }
        }

        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
    }
}
```

Declarative Pipeline

- Integrates with Jenkins UI for graphical reporting
  - Various plugins allow detailed feedback



billing-rest - Stage View

	Declarative: Checkout SCM	Initialize	Checkout	Build	Publish Reports	SonarQube analysis	ArchiveArtifact	Docker Tag & Push	Deploy - CI	Deploy - QA	Deploy - UAT	Deploy - Production	Declarative: Post Actions
Average stage times: (Average full run time: ~2min 59s)	768ms	1s	799ms	57s	5s	14s	125ms	23s	37ms	32ms	32ms	32ms	792ms
#118 Jul 17 20:58 1 commit	817ms	1s	690ms	1min 36s	10s	24s	198ms	38s	38ms				1s
#117 Jul 16 15:28 1 commit	792ms	1s	708ms	1min 36s	8s	23s	179ms	38s	42ms				2s
#116 Jul 15 21:13 No Changes	672ms	869ms	694ms	1min 33s	10s	24s	183ms	37s	40ms				75ms

Stages

- Stages represent the logical structure of the pipeline
  - Enclosed in a “stages” block
- Stages represent phases of the pipeline usually
  - Build
  - Test
  - Scan
  - Deploy
- Stages can be whatever we want them to be
  - As long as it represents some logical stage in the pipeline
- Stages provide
  - Clear visualization
  - Failure isolation
  - Easier troubleshooting

```
stages {
    stage('Build') { ... }
    stage('Test') { ... }
}
```

Pipeline Steps

- A step is the smallest unit of work in a Jenkins pipeline
- Steps
  - Execute actions
  - Run commands
  - Interact with Jenkins features
  - Integrate with tools and plugins
  - Only one steps{} block per stage
- Steps are always defined inside a stage
- Stages organize work, but steps do the work

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'mvn clean package'
            }
        }

        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
    }
}
```

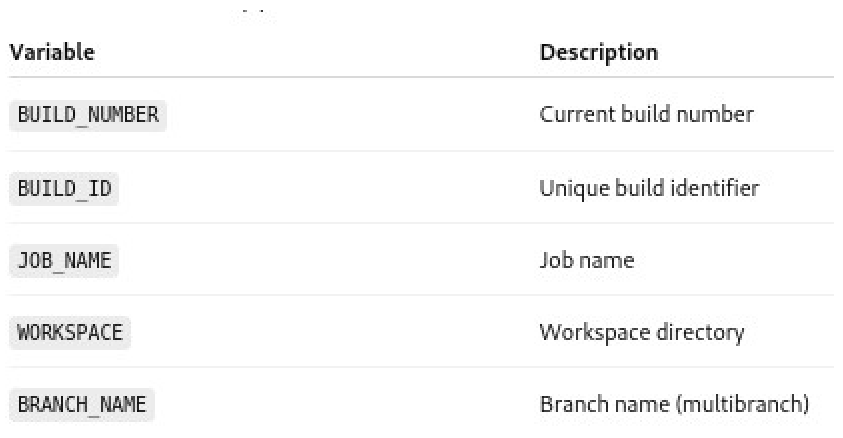
Agents

- The agent defines where a pipeline or stage executes
  - Common options
    - *“agent any” means any available agent*
    - *“agent none” means the stages define agents*
  - Labeled agents
    - *Runs on an agent with a specific label*
    - *Labels are assigned when an agent is defined by Jenkins*
  - Docker or Kubernetes agents can be created that provide specific types of execution environments
- In the example shown, each stage
  - Gets its own execution environment
  - Is scheduled independently
  - Releases its agent after completion

```
pipeline {
    agent none   // No default agent

    stages {

        stage('Build') {
            agent any
            steps {
                echo 'Running build on any available agent'
                sh 'echo Building...'
            }
        }

        stage('Linux Tests') {
            agent { label 'linux' }
            steps {
                echo 'Running tests on a Linux agent'
                sh 'echo Testing on Linux'
            }
        }

        stage('Docker Build') {
            agent {
                docker {
                    image 'alpine:latest'
                }
            }
            steps {
                echo 'Running inside a Docker container'
                sh 'echo Inside container'
            }
        }
    }
}
```

Environment Variables

- Key–value pairs that
  - Provide configuration information to a pipeline
  - Control behavior without changing code
  - Are available to steps at runtime
  - Provide a shared context for a pipeline execution
- Built-in Jenkins variables
  - Jenkins automatically provides variables that provide information about Jenkins, the build, the run, and other related information
  - The example shows some typical ones
  - Referenced with the $VAR notation



Variable	Description
`BUILD_NUMBER`	Current build number
`BUILD_ID`	Unique build identifier
`JOB_NAME`	Job name
`WORKSPACE`	Workspace directory
`BRANCH_NAME`	Branch name (multibranch)

```
sh 'echo Build number is $BUILD_NUMBER'
```

Environment Variables

- Variables are defined directly in a pipeline
  - Defined in an “environment” block
  - Values are available throughout the pipeline
  - Referenced in two different ways
- echo "APP_ENV is set to ${env.APP_ENV}"
  - This line runs inside Jenkins
    - *It’s evaluated by Groovy*
    - *env is a Jenkins-provided object*
    - *APP_ENV is read from Jenkins’ environment map*
    - *Essentially “Ask Jenkins what the environment variable is.”*
    - *Uses Groovy syntax*
  - Evaluated before any shell is started
  - Used in:
    - *echo*
    - *if conditions*
    - *variable assignment*
    - *pipeline logic*

```
pipeline {
    agent any

    environment {
        APP_ENV = 'test'
    }

    stages {
        stage('Show Environment Variable') {
            steps {
                echo "APP_ENV is set to ${env.APP_ENV}"
                sh 'echo APP_ENV from shell is $APP_ENV'
            }
        }
    }
}
```

```
if (env.APP_ENV == 'prod') {
    echo 'Production build'
}
```

Environment Variables

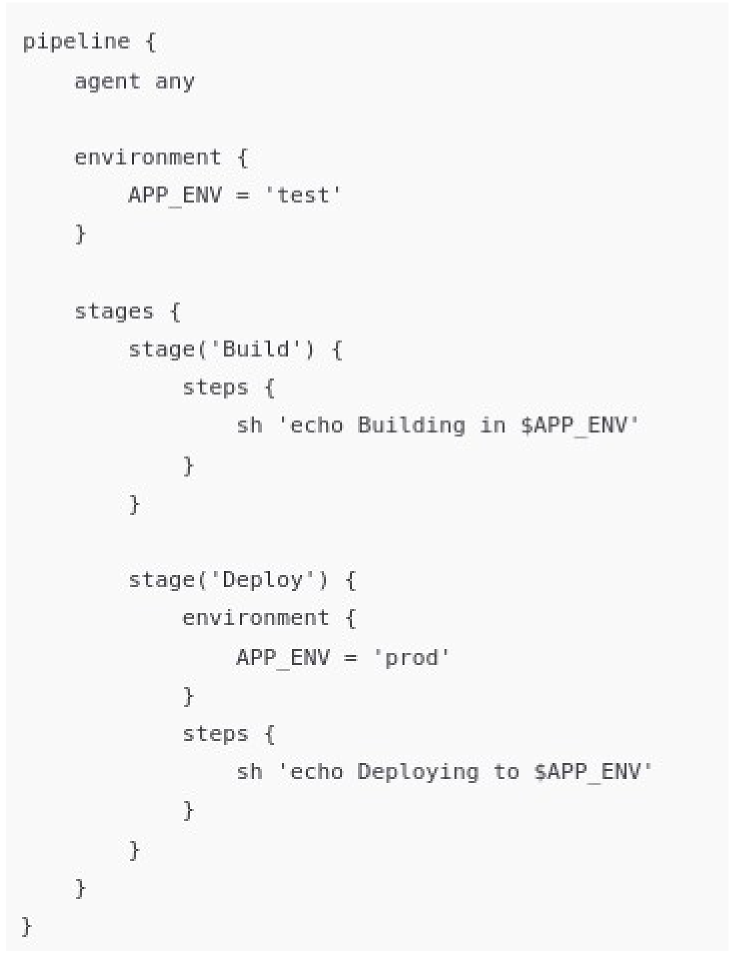
- sh 'echo APP_ENV from shell is $APP_ENV'
  - This line starts a shell process on the agent
    - *$APP_ENV is expanded by the shell*
    - *Jenkins injects the environment variable into the shell's environment*
    - *Essentially "Ask the operating system what the variable is."*
  - Uses shell syntax
    - *Evaluated by /bin/sh (or similar)*
  - Used in:
    - *Build tools*
    - *Scripts*
    - *Command-line utilities*
  - Groovy (Jenkins) evaluates env.VAR
  - Shell (Agent) evaluates $VAR

```
pipeline {
    agent any

    environment {
        APP_ENV = 'test'
    }

    stages {
        stage('Show Environment Variable') {
            steps {
                echo "APP_ENV is set to ${env.APP_ENV}"
                sh 'echo APP_ENV from shell is $APP_ENV'
            }
        }
    }
}
```

```
sh 'if [ "$APP_ENV" = "prod" ]; then echo Production; fi'
```

Environment Variables

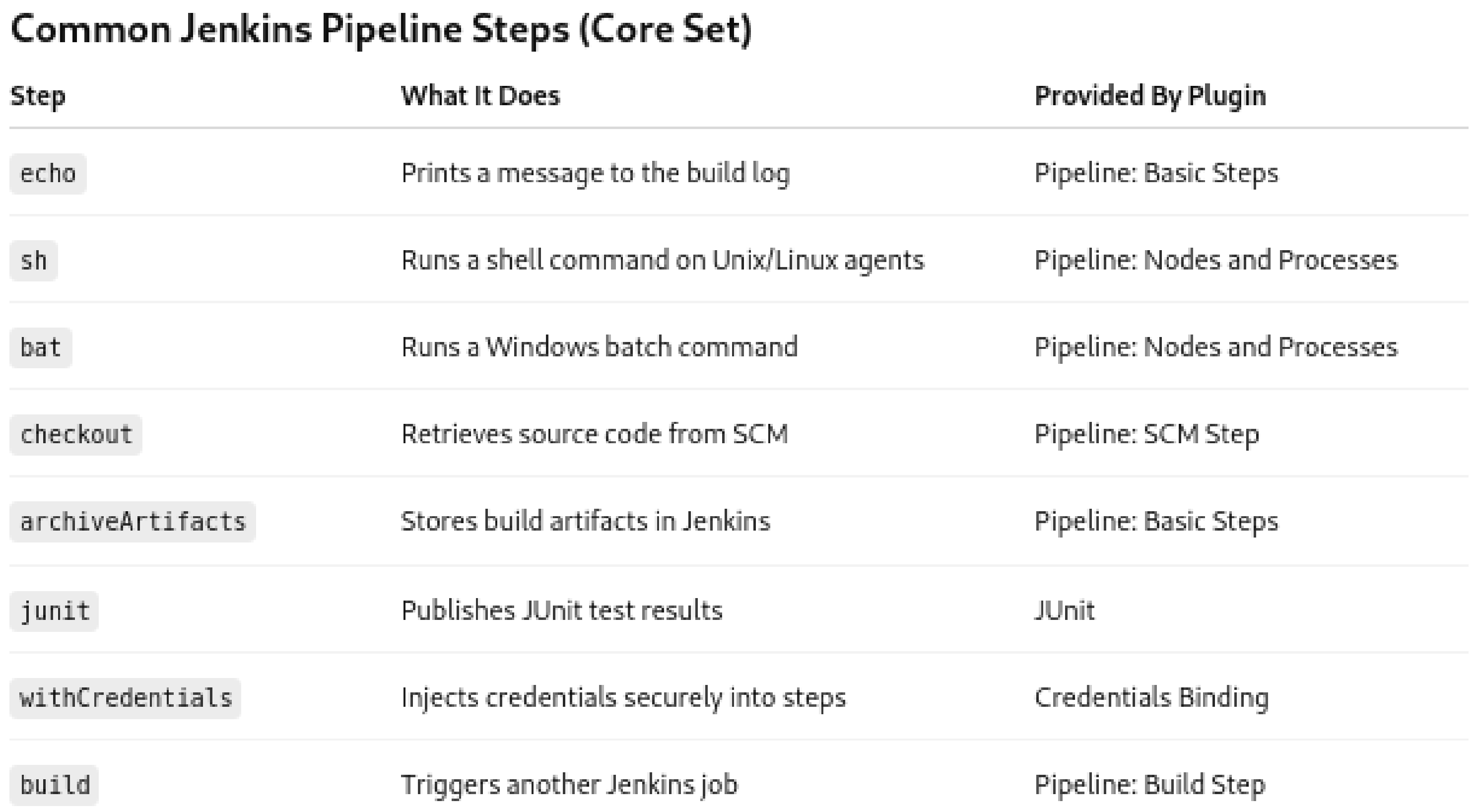
- Stage-specific environment variables
  - Are defined inside a single stage
  - Apply only to that stage
  - Override pipeline-level variables if there is a conflict
    - *Referred to as “shadowing”*
  - Allows each stage to have its own configuration without affecting the rest of the pipeline.



```
pipeline {
    agent any

    environment {
        APP_ENV = 'test'
    }

    stages {
        stage('Build') {
            steps {
                sh 'echo Building in $APP_ENV'
            }
        }

        stage('Deploy') {
            environment {
                APP_ENV = 'prod'
            }
            steps {
                sh 'echo Deploying to $APP_ENV'
            }
        }
    }
}
```

Steps and Plugins

- Most steps come from plugins

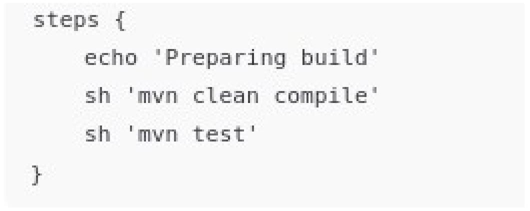


Common Jenkins Pipeline Steps (Core Set)

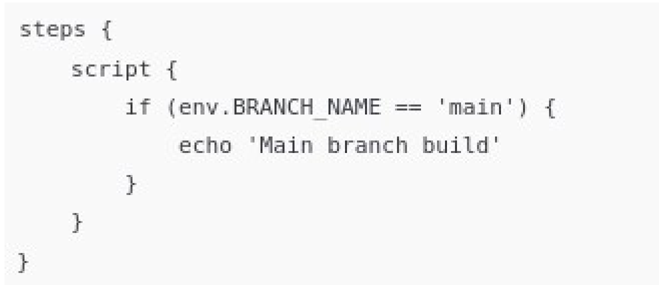
Step	What It Does	Provided By Plugin
`echo`	Prints a message to the build log	Pipeline: Basic Steps
`sh`	Runs a shell command on Unix/Linux agents	Pipeline: Nodes and Processes
`bat`	Runs a Windows batch command	Pipeline: Nodes and Processes
`checkout`	Retrieves source code from SCM	Pipeline: SCM Step
`archiveArtifacts`	Stores build artifacts in Jenkins	Pipeline: Basic Steps
`junit`	Publishes JUnit test results	JUnit
`withCredentials`	Injects credentials securely into steps	Credentials Binding
`build`	Triggers another Jenkins job	Pipeline: Build Step

Step Execution

- Multiple steps inside a stage run sequentially
- If a step fails:
  - The stage fails
  - The pipeline usually stops
  - Some steps (catchError, retry, timeout) modify this behavior
- Allow controlled scripting using a script block
  - The script{} block allows the insertion of Groovy code snippets into a declarative pipeline



```
steps {
    echo 'Preparing build'
    sh 'mvn clean compile'
    sh 'mvn test'
}
```



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

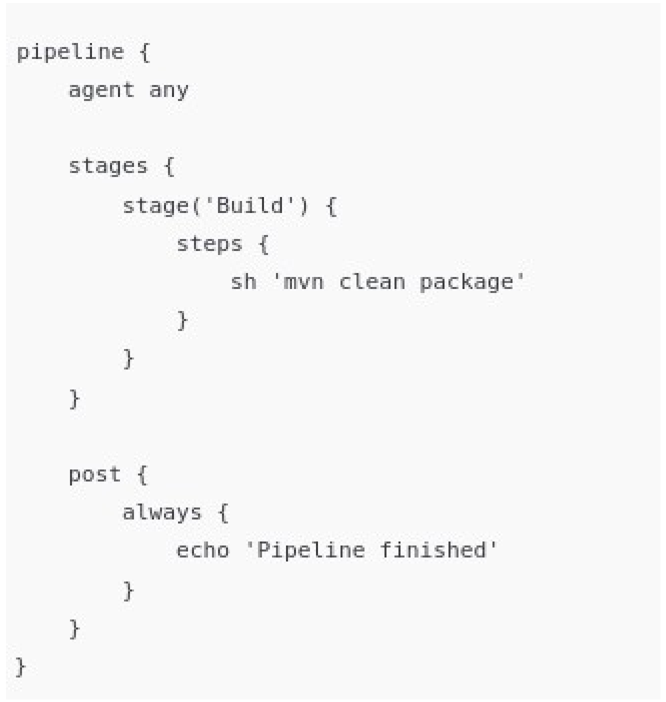
Post Stages

- A post section defines actions that run after a stage or after the entire pipeline completes
- Post stages are used for:
  - Cleanup
  - Notifications
  - Reporting
  - Finalization tasks
- They ensure important actions happen regardless of pipeline outcome
- Pipeline post stages execute at the end of a pipeline

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'mvn clean package'
            }
        }
    }

    post {
        always {
            echo 'Pipeline finished'
        }
    }
}
```

Post Stages

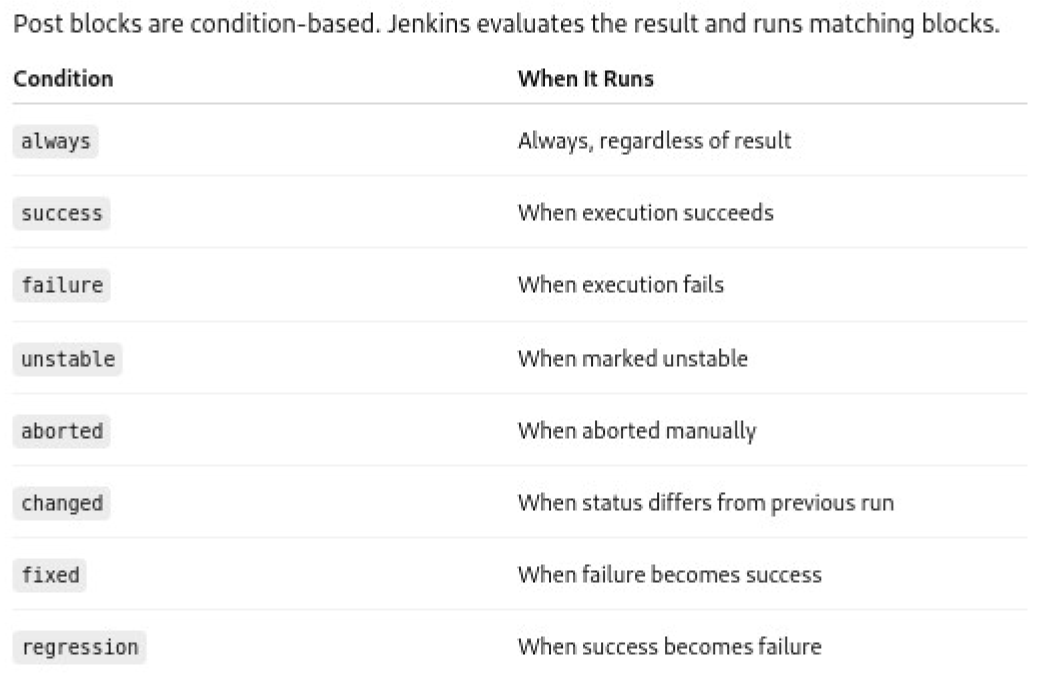
- Stage level post
  - Post stages can also be defined after a stage
  - Runs after a specific stage finishes



```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'mvn clean package'
            }
        }
    }

    post {
        always {
            echo 'Pipeline finished'
        }
    }
}
```

post Conditions

- Post stages define what should happen after the main work completes
  - Based on the result of the stage or pipeline
  - The stage performs the actual build work
  - The post block decides what to do with the result of the work done by the pipeline or stage
  - More than one post block can execute
- Post stages usually handle tasks like
  - Notifications
  - Cleanup
  - Recovery from failed builds
  - Delivering the artifact to a repository

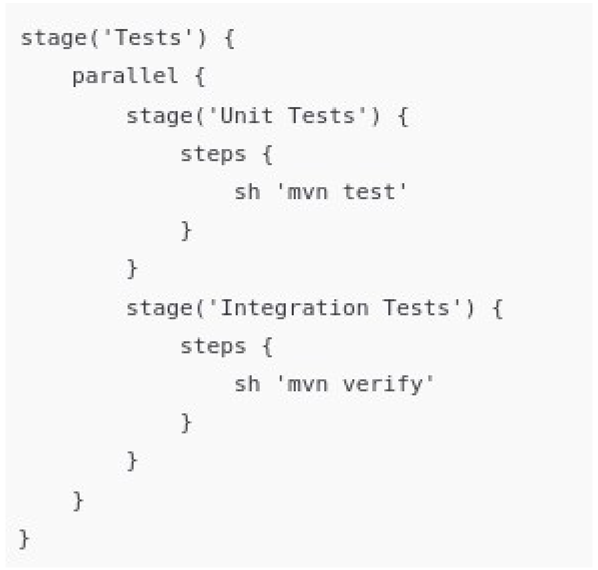


Post blocks are condition-based. Jenkins evaluates the result and runs matching blocks.

Condition	When It Runs
`always`	Always, regardless of result
`success`	When execution succeeds
`failure`	When execution fails
`unstable`	When marked unstable
`aborted`	When aborted manually
`changed`	When status differs from previous run
`fixed`	When failure becomes success
`regression`	When success becomes failure

Parallel Stages

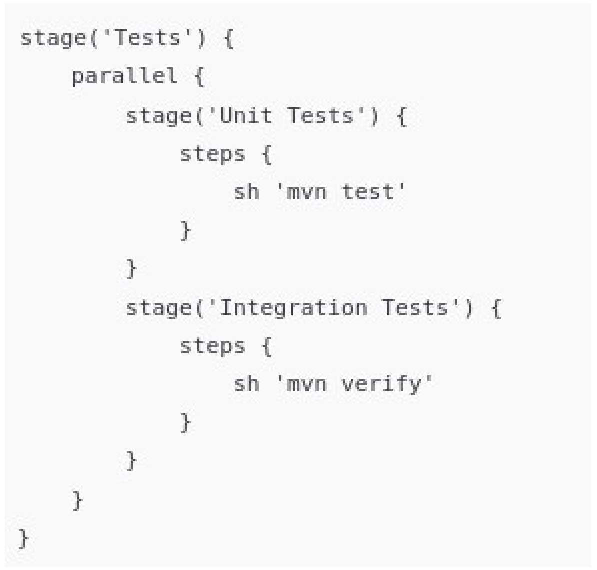
- Allow Jenkins to run multiple stages at the same time instead of sequentially
- Used when
  - Tasks are independent
  - Results don’t depend on each other
  - Faster feedback is desired
- Parallel stages reduce pipeline runtime by doing independent work concurrently



```
stage('Tests') {
    parallel {
        stage('Unit Tests') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Integration Tests') {
            steps {
                sh 'mvn verify'
            }
        }
    }
}
```

Parallel Stages

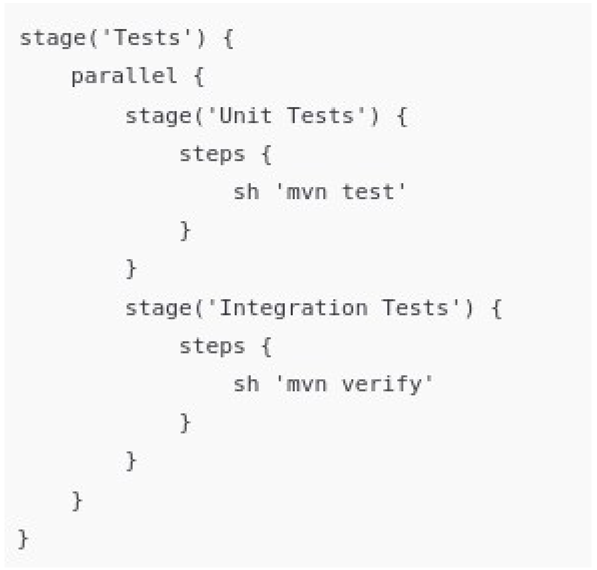
- Parallel execution is defined inside a stage using a parallel block
- Each nested stage
  - Has its own name
  - Has its own steps
  - Runs concurrently with the others
- Jenkins schedules each parallel stage independently
- Each parallel stage
  - Requires its own executor
  - May run on a different agent



```
stage('Tests') {
    parallel {
        stage('Unit Tests') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Integration Tests') {
            steps {
                sh 'mvn verify'
            }
        }
    }
}
```

Parallel Stages

- If any parallel stage fails, the parent stage fails
- Other parallel stages may
  - Continue running
  - Be aborted (depending on configuration)
- The pipeline moves on only after all parallel work completes or is stopped



```
stage('Tests') {
    parallel {
        stage('Unit Tests') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Integration Tests') {
            steps {
                sh 'mvn verify'
            }
        }
    }
}
```

Agents and Parallel Stages

- Parallel stages can
  - Share the same pipeline agent
  - Define their own separate agents
- This allows
  - Platform-specific execution
  - True parallelism across environments
- Common use cases
  - Test matrix execution (discussed later)
  - Multi-platform builds
  - Tasks that can be done independently of a build
    - *Code Quality scans*
    - *Security scans*

```
stage('Tests') {
    parallel {
        stage('Linux Tests') {
            agent { label 'linux' }
            steps {
                sh 'run-tests.sh'
            }
        }
        stage('Windows Tests') {
            agent { label 'windows' }
            steps {
                bat 'run-tests.bat'
            }
        }
    }
}
```

```
Tests
 ├─ Unit
 ├─ Integration
 ├─ Security
 └─ Performance
```

Matrix Builds

- A matrix build runs the same pipeline logic multiple times using different combinations of variables
  - A matrix build defines a set of axes (variables)
  - Jenkins runs the pipeline once for each combination
- Matrix builds test many variations of the same workflow automatically
- The example on the right generates four executions
  - linux + JDK 11
  - linux + JDK 17
  - windows + JDK 11
  - windows + JDK 17

```
pipeline {
    agent none

    stages {
        stage('Matrix Tests') {
            matrix {
                axes {
                    axis {
                        name 'OS'
                        values 'linux', 'windows'
                    }
                    axis {
                        name 'JDK'
                        values '11', '17'
                    }
                }

                stages {
                    stage('Test') {
                        steps {
                            echo "Running on ${OS} with JDK ${JDK}"
                        }
                    }
                }
            }
        }
    }
}
```

Executing Matrix Builds

- Jenkins expands all axis combinations
- Each combination (cell)
  - Runs as an independent execution
  - Has its own environment variables
  - Each axis value becomes an environment variable
  - Can run in parallel
  - Results are aggregated in the UI
  - A failure in one matrix cell
    - *Marks that cell as failed*
    - *Does not stop other combinations*
  - Overall stage result reflects combined outcomes

```
pipeline {
    agent none

    stages {
        stage('Matrix Tests') {
            matrix {
                axes {
                    axis {
                        name 'OS'
                        values 'linux', 'windows'
                    }
                    axis {
                        name 'JDK'
                        values '11', '17'
                    }
                }

                stages {
                    stage('Test') {
                        steps {
                            echo "Running on ${OS} with JDK ${JDK}"
                        }
                    }
                }
            }
        }
    }
}
```

Questions