Table of Contents

Create the Parameterized Job....2
The when Directive (Decision Making)....4
What is SSH Agent?....6

Create the Parameterized Job

Step 1: Create the Parameterized Job

1. Open your Jenkins Dashboard.
2. Click **New Item**, give it a name (e.g., `Full-Parameter-Demo`), select **Pipeline**, and click **OK**.
3. Scroll down to the **Pipeline** section.
4. In the **Definition** dropdown, ensure **Pipeline script** is selected.
5. Paste the code provided in the next section into the script box.
6. **Click Save.**

Step 2: Example Pipeline Code (All Common Types)

This script demonstrates the most frequently used parameter types and how to access them within your stages.

```
pipeline {
    agent any

    parameters {
        // 1. String Parameter
        string(name: 'PERSON_NAME', defaultValue: 'John Doe',
description: 'Who is running this?')

        // 2. Text Parameter (Multi-line)
        text(name: 'BIOGRAPHY', defaultValue: '', description: 'Enter a
short bio')

        // 3. Choice Parameter (Dropdown)
```

```
        choice(name: 'ENVIRONMENT', choices: ['Dev', 'Staging',
'Production'], description: 'Select target env')

        // 4. Boolean Parameter (Checkbox)
        booleanParam(name: 'RELEASE_BUILD', defaultValue: false,
description: 'Is this a release?')

        // 5. Password Parameter (Masked in UI and logs)
        password(name: 'API_KEY', defaultValue: '', description: 'Enter
your secret API key')
    }

    stages {
        stage('Display Parameters') {
            steps {
                echo "Hello, ${params.PERSON_NAME}!"
                echo "Deploying to: ${params.ENVIRONMENT}"

                // Logic based on Boolean
                script {
                    if (params.RELEASE_BUILD) {
                        echo "Warning: This is a RELEASE build!"
                    } else {
                        echo "This is a standard build."
                    }
                }
            }
        }

        stage('Secure Task') {
            steps {
                // Accessing password (automatically masked)
                echo "Using API Key: ${params.API_KEY}"
            }
        }
    }
}
```

Step 3: Triggering the Build

1. **First Run:** Because you just pasted the code, Jenkins doesn't "know" the parameters yet. Click **Build Now** once. It will likely fail or run with defaults.
2. **Refresh:** Refresh the page. You will notice "Build Now" has changed to **"Build with Parameters"**.
3. **Input Values:** Click it, fill out the form (type your name, select an

environment, check the box), and click **Build**.

Parameter Types Reference Table

Type	Syntax	Best Use Case
String	string(...)	Single-line text like a Branch name or Tag.
Text	text(...)	Multi-line input like a release note or SSH public key.
Choice	choice(...)	Fixed options like Environment (Dev/Prod) or Regions.
Boolean	booleanParam(...)	On/Off switches like "Run Tests" or "Clean Workspace".
Password	password(...)	Sensitive tokens or passwords (hidden in UI).

A Jenkins Pipeline's `post` section is where you define actions to run after the main stage's execution. It provides a way to handle cleanup, notifications, or other final tasks, regardless of whether the build was successful or failed.

##

The `when` Directive (Decision Making)

The `when` block is used inside a `stage`. It tells Jenkins: "Check this rule first; if it's false, skip this entire stage."

Jenkinsfile Snippet:

```
stage('Deploy to Production') {
    when {
        branch 'main' // Only runs if the current branch is main
    }
    steps {
        echo "Deploying to the live website..."
    }
}
```

The `post` Directive (Reactionary Logic)

The `post` block runs at the end of a stage or the whole pipeline. It reacts to the **status** of the build.

Common Conditions:

- **`always`**: Runs no matter what (good for cleaning up files).
- **`success`**: Runs only if everything went perfectly.
- **`failure`**: Runs only if the build or tests crashed.

Jenkinsfile Snippet:

```
pipeline {
    agent any
    stages {
        stage('Test') {
            steps {
                sh './run-tests.sh'
            }
        }
    }
    post {
        failure {
            echo "ALARM! The build failed. Sending email to the team..."
        }
```

```
        success {
            echo "Great job! Moving on..."
        }
    }
}
```

Advanced Logic (`expression`)

Sometimes you need more than just a branch name. You can use `expression` to write any logic you want using Groovy code.

```
when {
    expression { params.RUN_TESTS == true }
}
```

"How can you combine multiple conditions in a Jenkins Pipeline?"

- **Answer:** You can use logic operators inside the `when` block:
  - **`allOf`**: All conditions must be true (like an `AND`).
  - **`anyOf`**: At least one must be true (like an `OR`).
  - **`not`**: Inverts the condition.
- **Example:** *`when { allOf { branch 'main'; expression { params.DEPLOY == true } } }`*

##

What is SSH Agent?

Normally, to connect to a remote server, you need an SSH key. If your Jenkins script needs to move a file to a web server, it needs that key.

- **The Problem:** Storing keys in plain text inside a script is a massive security risk.
- **The Solution:** You store the key in Jenkins' "Credentials" vault. The **SSH Agent Plugin** then "provides" that key to your build process temporarily

and safely.

Step-by-Step Setup

Step 1: Install the Plugin

1. Go to **Dashboard** > **Manage Jenkins** > **Plugins**.
2. Click **Available plugins** and search for `SSH Agent`.
3. Install it and restart Jenkins if prompted.

Step 2: Add your SSH Private Key to Jenkins

1. Go to **Manage Jenkins** > **Credentials**.
2. Click on the **(global)** domain -> **Add Credentials**.
3. **Kind:** Select `SSH Username with private key`.
4. **ID:** Give it a name (e.g., `my-server-key`). This is what you'll use in your code.
5. **Username:** The user on the remote server (e.g., `ubuntu` or `root`).
6. **Private Key:** Click **Enter directly** and paste the content of your `id_rsa` file.
7. Click **Create**.

Step 3: Use it in a Pipeline

Now, you use the `sshagent` wrapper in your Pipeline script. This "wraps" your commands in a secure environment.

```
pipeline {
    agent any
    stages {
        stage('Deploy') {
            steps {
                // Use the ID you created in Step 2
                sshagent(['my-server-key']) {
                    // Now you can run SSH commands without a password!
```

```
                    sh 'ssh -o StrictHostKeyChecking=no
ubuntu@192.168.1.50 "ls -la"'
                    sh 'scp index.html
ubuntu@192.168.1.50:/var/www/html/'
                }
            }
        }
    }
}
```

StrictHostKeyChecking=no: When connecting to a new server, SSH usually asks "Do you trust this host?". A Jenkins script can't click "Yes." Use `-o StrictHostKeyChecking=no` to bypass this prompt automatically

```
##########################################################
####################################
```

```
##########################################################
####################################
```

```
##########################################################
####################################
##########################################################
####################################
```

Sections within `post`

The `post` section is a block that contains different **condition blocks**, each of which executes based on the final status of the build. Here are the most common ones:

- **`always`**: This block runs regardless of the build's outcome (success, failure, or aborted). It's useful for cleanup tasks like deleting temporary files or notifying a user about the build's completion.
- **`success`**: This block runs only if all the stages in the pipeline have completed successfully. It's often used for tasks like deploying an application or sending a success notification.

- **`failure`**: This block runs if any stage in the pipeline fails. It's a good place to send a failure notification or archive logs for debugging.
- **`unstable`**: This block runs if the build is "unstable," which typically means a successful build with failing test cases.
- **`aborted`**: This block runs if the build is manually aborted by a user.
- **`changed`**: This block runs if the current build's status is different from the previous build's status. For example, if a successful build is followed by a failed one.

Example Usage

Here is a simple example of a Declarative Pipeline with a `post` section.

Groovy

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building the application...'
                // Simulating a successful build
            }
        }
        stage('Test') {
            steps {
                echo 'Running tests...'
                // Simulating a failed test
                script {
                    error('Tests failed!')
                }
            }
        }
    }

    post {
        always {
            echo 'This will always run, even if the build fails.'
            // Cleanup actions
        }
        success {
            echo 'The build was successful!'
            // Deployment actions
        }
        failure {
            echo 'The build failed. Sending a notification...'
            // Notification actions
        }
        aborted {
            echo 'The build was aborted.'
```

```
            // Abort-specific actions
        }
    }
}
```

In this example, because the **'Test'** stage is designed to fail, the **`failure`** block within `post` will execute, along with the **`always`** block. The **`success`** block will be skipped.
