

CloudBees Jenkins Platform: Certification Training

4 - Modern Jenkins - Pipelines

Table of Contents

4 - Modern Jenkins: Pipelines	1
Journey Description	1
Starting with Jenkins Pipeline	1
Writing your first Pipeline script.	1
The Snippet Generator	2
A bit more complex Pipeline.	3
Moving to an SCM-based Pipeline.	4
Moving our CD Pipeline to Jenkins Pipeline.	6
Different stages on different agents	7
Manipulate Environment and Shell	7
Stages for Maven Goals and Test Reports	8
From CI to CD	10
Multi-branch Pipelines	12
Pipeline Shared Libraries	15
Prepare the Shared Library Repository	15
Configure the Shared Library in Jenkins	16
Test the Shared Library on the Multi-Branch Project	18
Journey Summary	20

4 - Modern Jenkins: Pipelines

Journey Description

IMPORTANT

This lab requires having covered the [Lab 3](#)

The goal of this Journey is to cover the new Jenkins standard Job: the **Pipeline**.

It is based on what have been done in [Lab 3](#)

We will cover:

- Starting with Jenkins Pipeline
- Implementing our CD Pipeline with a more complex Jenkins Pipeline
- Usage of Multi-branch pipelines for SCM branch management
- Usage of the Shared Global Pipeline Library to reuse code across Pipelines

Starting with Jenkins Pipeline

The goal of this exercise is to start using the Jenkins Pipelines jobs.

Writing your first Pipeline script

- Start from the Jenkins dashboard
- Create a **New Item** of type **Pipeline**, named `demoapp-cd-pipeline`
- Review the configuration, to check what settings are there by default

TIP

Note that the **Execute concurrent builds if necessary** option is checked by default

- Scrolling down to the **Pipeline** section; we will use the integrated editor
- Select **Pipeline Script** as definition
- Copy/Paste the following code, save and run the pipeline:

```
node {  
  echo "Hello World"  
}
```

- Check the Output Log of this first build, as usual. The "Hello World" string is obviously there.

Notice what the **Stage View** section tells you:

TIP

Stage View

This Pipeline has run successfully, but does not define any stages. Please use the **stage** step to define some stages in this Pipeline.

The Snippet Generator

Now let's use the **Snippet Generator** to generate some additional steps:

- Browse again to the job configuration, scroll to the bottom, and click on the **Pipeline Syntax** link

TIP

Direct link: <http://localhost:5000/jenkins/job/demoapp-cd-pipeline/pipeline-syntax>

- The **Snippet Generator** should open in a new tab of your webbrowser
- Select the **stage** as **Sample Step** in the **Steps** section
 - Fill the **Stage Name** field with the string `Printing Hello World`
 - Click the **Generate Pipeline Script** button

+ image::lab-pipeline-hello-world-pipeline-config.jpg[title="Snippet Generator",width=1000]

- Copy the generated snippet and paste it into the Pipeline editor, inside the **node** scope, the line before the **echo**:

```
node {
    stage('Printing Hello World') {
        echo "Hello World"
    }
}
```

- Run the Pipeline again and see the **Stage View** in action:

Jenkins

Jenkins > hello-pipeline >

[Back to Dashboard](#)
[Status](#)
[Changes](#)
[Build Now](#)
[Delete Pipeline](#)
[Configure](#)
[Move](#)
[Full Stage View](#)

Build History [trend](#)

Build	Time
#2	Sep 10, 2016 3:20 AM
#1	Sep 10, 2016 3:16 AM

[RSS for all](#)
[RSS for failures](#)

Pipeline hello-pipeline

[Recent Changes](#)

Stage View

Average stage times:
(Average full run time: ~10ms)

Stage	Time	Changes
#2	Sep 09 20:20	No Changes
#1	Sep 09 20:16	No Changes

Printing Hello World
10ms

Permalinks

- [Last build \(#1\), 4 min 40 sec ago](#)
- [Last stable build \(#1\), 4 min 40 sec ago](#)
- [Last successful build \(#1\), 4 min 40 sec ago](#)

Figure 1. Hello World Pipeline Stage View

A bit more complex Pipeline

- Using the snippet generator and the text editor, create a new pipeline that will:
 - Clone the `demoapp` application, within a stage named `Checkout`
 - Build and package the application, with the `maven3` tool installation, within another stage named `Build`

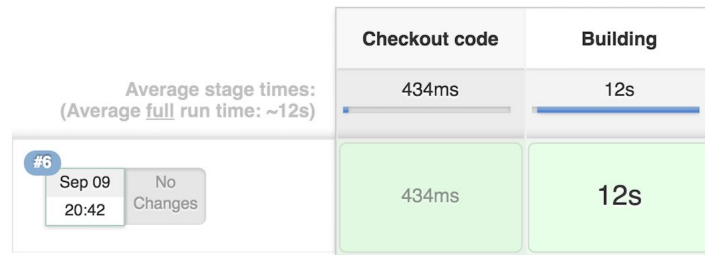
TIP DSL keyword to use: `node`, `stage`, `git`, `tool`, `sh`

- Resulting build should be:

Pipeline hello-pipeline



Stage View



Permalinks

- [Last build \(#6\), 4 min 6 sec ago](#)
- [Last stable build \(#6\), 4 min 6 sec ago](#)
- [Last successful build \(#6\), 4 min 6 sec ago](#)

Figure 2. Maven Pipeline Stage View

- Proposed solution:

```
node {
  stage('Checkout code') {
    git credentialsId: 'butler-gogs-creds',
      url: 'http://localhost:5000/gitserver/butler/demoapp.git'
  }
  stage('Build') {
    def mvnHome = tool 'maven3'
    sh "${mvnHome}/bin/mvn package"
  }
}
```

Moving to an SCM-based Pipeline

We want to move the script to a `Jenkinsfile` stored in the SCM.

- Start with the **WebIDE**; create a `Jenkinsfile` at the root of the project
- Fill this `Jenkinsfile` with the previously proposed solution
- Commit and push it to the remote repository
- Browse to the `demoapp-cd-pipeline` configuration:
 - Change the pipeline **Definition** to **Pipeline script from SCM**
 - Point it out to the git repository and gogs browser:

Figure 3. Configuration for simple Jenkinsfile

- Build it, and check that everything works fine.

TIP

To prevent Jenkins from building the original jobs, update the WebHooks from **GitServer** to point the current job, and use **Test Delivery** to test the build

- Notice that the **git** configuration is redundant here:
 - Configured at the job level
 - Repeated at inside the **Jenkinsfile**
- To avoid future configuration shifting:
 - Edit the **Jenkinsfile** in the **WebIDE** again
 - Replace the **git . . .** line with **checkout scm**

TIP

The keyword **checkout** always resolves to the same git changeset as the checked out Jenkinsfile on the master.

- We also want to "force" the stages to run on the ssh-agent only, by adding the label/agent name to the **node** keyword
- Proposed solution:

```
node('ssh-agent') {
    stage('Checkout code') {
        checkout scm
    }
    stage('Build') {
        def mvnHome = tool 'maven3'
        sh "${mvnHome}/bin/mvn package"
    }
}
```

- Validate that everything stills works:

Pipeline hello-pipeline



[Recent Changes](#)

Stage View

			Checkout code	Building
Average stage times: (Average full run time: ~11s)			483ms	11s
#10	Sep 09 21:06	1 commits	542ms	10s
#9	Sep 09 21:01	No Changes	535ms	9s
#8	Sep 09 20:59	1 commits	422ms	12s
#6	Sep 09 20:42	No Changes	434ms	12s

Figure 4. Stage View of our builds

That's all for this exercise !

Moving our CD Pipeline to Jenkins Pipeline

The goal of this exercise is to implement our CD pipeline by iterating on the Jenkins Pipeline we previously created.

Different stages on different agents

- We are still using the `Jenkinsfile` in the repository, with the Job `demoapp-cd-pipeline`
- The goal of this exercise is re-implement the "3 stages" process we've done with 3 jobs in Part 2 and 3:
 - First, run the Apache Maven build on `ssh-agent`
 - Then we want to build and "smoke test" using Docker, on the agent tagged `docker`
 - The docker image will be named from a variable, with the tag "latest"
- Try to implement a solution in a single Jenkinsfile
 - New Keywords to use: `stash`, `unstash`
- Proposed solution:

```
def DOCKER_IMG_BASENAME='demo-app'

node('ssh-agent') {
    stage('Checkout code') {
        checkout scm
    }
    stage('Build') {
        def mvnHome = tool 'maven3'
        sh "${mvnHome}/bin/mvn package"

        // Stash is like archiveArtifacts,
        // but is only valid for the build duration
        stash includes: '**/target/*.jar',
              name: 'application-binaries'
    }
}

node('docker') {
    stage('Docker Build') {
        // We need Dockerfile in the source code
        checkout scm

        // And the compiled application
        unstash 'application-binaries'
        sh "docker build -t ${DOCKER_IMG_BASENAME}:latest ./"
    }
}
```

Manipulate Environment and Shell

- Goals of this exercise:
 - Use environment variables to avoid passing `${mvnHome}` to each Maven command
 - Set the docker image's tag to the git short changeset fetched from a command line, or `latest` by default
- New keyword to use: `withEnv`

- Proposed solution:

```
def DOCKER_IMG_BASENAME='demo-app'
def GIT_SHORT_CHANGESET='latest'

node('ssh-agent') {
    stage('Checkout code') {
        checkout scm

        // We can set a groovy variable from a "sh" command stdout.
        // Do not forget to remove endlines with trim()
        GIT_SHORT_CHANGESET =
            sh(returnStdout: true, script: 'git rev-parse --short HEAD').trim()
    }
    // The environment variable will be appended by the Maven install path
    // letting you use the command 'mvn' in any "sh" inside the withEnv block
    withEnv(["PATH+MAVEN=${tool 'maven3'}/bin"]) {
        stage('Build') {
            sh "mvn package"
            stash includes: '**/target/*.jar',
                name: 'application-binaries'
        }
    }
}

node('docker') {
    stage('Docker Build') {
        checkout scm
        unstash 'application-binaries'
        sh "docker build -t ${DOCKER_IMG_BASENAME}:${GIT_SHORT_CHANGESET} ./"
    }
}
```

Stages for Maven Goals and Test Reports

- Goals of this exercise:

- Split the Maven goals into different stages

TIP

Getting quick feedback about which step of your pipeline failed is one of the most important optimizations to do. Splitting the job into different stages helps a lot.

- Publish the Test reports from Maven (both unit and integration)
- Do not fail the build if integration tests are failing: we want the build to be "unstable"

TIP

If unit tests fails, the build must fail. It is something not acceptable.

- New keywords to use: `junit`
- Proposed solution:

```
def DOCKER_IMG_BASENAME='demo-app'
def GIT_SHORT_CHANGESET='latest'

node('ssh-agent') {
    stage('Checkout code') {
        checkout scm
        GIT_SHORT_CHANGESET =
            sh(returnStdout: true, script: 'git rev-parse --short HEAD').trim()
    }
    withEnv(["PATH+MAVEN=${tool 'maven3'}/bin"]) {
        stage('Build') {
            sh "mvn package"
            junit '**/target/surefire-reports/*.xml'
            stash includes: '**/target/*.jar',
                name: 'application-binaries'
        }
        stage('Integration Tests') {
            // We do not want Maven to fail if tests fails
            sh "mvn verify -fn"
            // Because junit takes care of setting the build
            // to "UNSTABLE" state if tests fails
            junit '**/target/failsafe-reports/*.xml'
        }
    }
}

node('docker') {
    stage('Docker Build') {
        checkout scm
        unstash 'application-binaries'
        sh "docker build -t ${DOCKER_IMG_BASENAME}:${GIT_SHORT_CHANGESET} ./"
    }
}
```

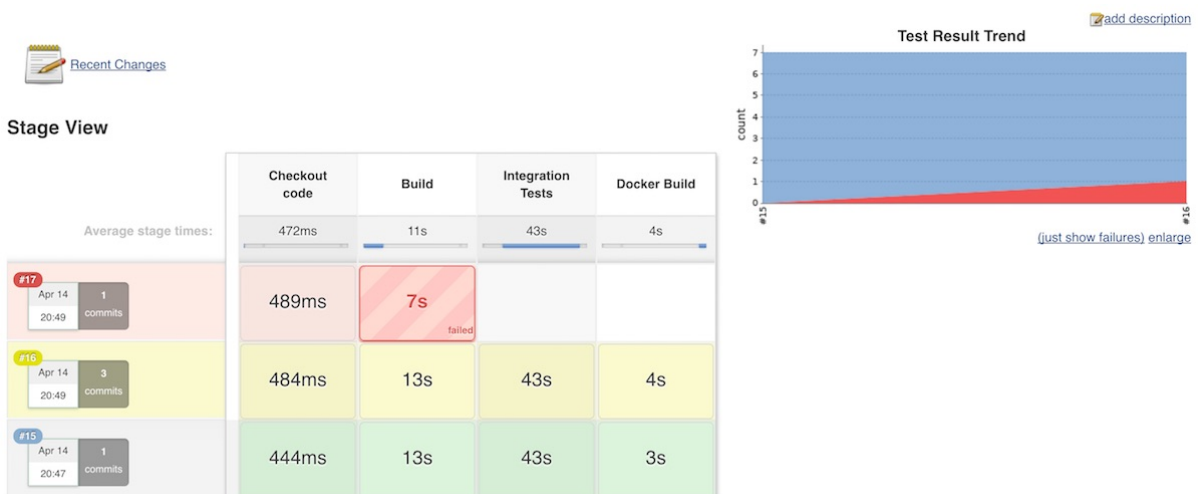


Figure 5. Example of Failed/Unstable Build with junit step

From CI to CD

- Goals of this exercise:
 - Add a stage for deploying the docker image to staging
 - We need to call the existing freestyle job demoapp-staging-deployer, which expects a `DOCKER_IMAGE` parameter

TIP

We did that in Lab Part 2 and 3, in freestyle jobs

- Continuous Deployment means "human action" at a step transition: we want to ask for manual validation **before** deploying the docker image
- Last but not least: if the person in charge of validation is not available, we want the job:
 - To not use any executors while waiting for human action
 - To gracefully stop after waiting for one hour
- New keywords to use: `input`, `timeout`, `build`
- Proposed solution:

```
def DOCKER_IMG_BASENAME='demo-app'
def GIT_SHORT_CHANGESET='latest'

node('ssh-agent') {
    stage('Checkout code') {
        checkout scm
        GIT_SHORT_CHANGESET =
            sh(returnStdout: true, script: 'git rev-parse --short HEAD').trim()
    }
    withEnv(["PATH+MAVEN=${tool 'maven3'}/bin"]) {
        stage('Build') {
            sh "mvn package"
            junit '**/target/surefire-reports/*.xml'
            stash includes: '**/target/*.jar', name: 'application-binaries'
        }
        stage('Integration Tests') {
            sh "mvn verify -fn"
            junit '**/target/failsafe-reports/*.xml'
        }
    }
}
node('docker') {
    stage('Docker Build') {
        checkout scm
        unstash 'application-binaries'
        sh "docker build -t ${DOCKER_IMG_BASENAME}:${GIT_SHORT_CHANGESET} ./"
    }
}

// Putting "input" inside a stage will allow
// the GUI to visualize it and show the popup
// If not you will have to go to the Console Output of the build
stage('Waiting Approval') {
    // We do not put "input" within a node
    // to avoid eating an executor while waiting
    timeout(time:1, unit:'DAYS') {
        input message: "Is it ok to deploy docker image?", ok: 'Deploy'
    }
}

// We do not put "build" within a node for same reasons as "input"
stage('Deploy') {
    // This will call the downstream job and block until it finishes
    // Unless disabled, downstream job status will be reported to the pipeline
    build job: 'demoapp-staging-deployer',
        parameters: [string(name: 'DOCKER_IMAGE',
            value: "${DOCKER_IMG_BASENAME}:${GIT_SHORT_CHANGESET}")]]
}
```

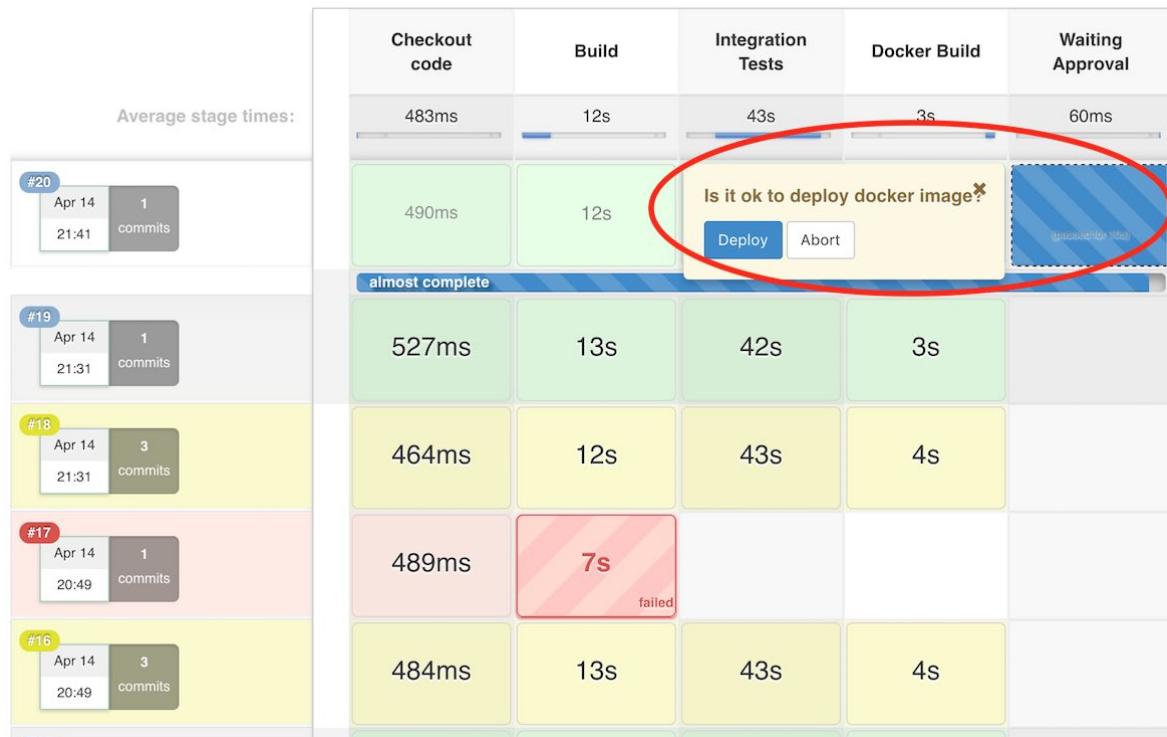


Figure 6. Example of Input with our CD Pipeline

That's all for this exercise !

Multi-branch Pipelines

The goal of this exercise is to improve our pipeline, using the Multi-branch capability.

- We want to parallelize the Integration Tests and the Docker build, to speed up our pipeline.
- **Challenge:** we do not want to risk the stability of the main pipeline, because the rest of our team is currently working on it.
 - Solution: we will implement the new Pipeline in an SCM branch named "parallel", and move our Pipeline Job to a Multi-Branch Job
- Create a **New Item** at the root of the Jenkins dashboard:
 - Name: `demoapp-project`
 - Type: **Multibranch Pipeline**
- In the **Branch Sources** section, add a **source**:
 - Type: **Git**
 - Project Repository: <http://localhost:5000/gitserver/butler/demoapp>
 - Set the repository browser to `gogs`
 - Do not select any trigger
 - Enable `*Project`
- Once created:
 - A folder named `demoapp-project` has been created

- Inside that folder is a pipeline project that is created automatically, named **master** and mapped to the SCM branch **master**.
- A build of **master** has started immediately.
- Time to update the Gogs webhook. Set it to this URL: <http://butler:butler@localhost:5000/jenkins/job/demoapp-project/build?delay=0>
 - The **anonymous** user must have the **Overall:Read** right in **Manage Jenkins** → **Configure Global Security**

TIP

This is only required because of **Gogs**. Github and Bitbucket handles the automatic creations without "opening" authorizations.

- This webhook URL scans the whole repository and builds **only** what is needed.
 - Set the number of executors to 2 on the ssh-agent; this allows Jenkins to handle more parallel operations
 - In the **WebIDE**, create a new branch:
- Close all open files
- Use the **CodeGit** panel, and click on **Branches**
- On the **Branches** panel, click on **New**
- Enter the name of the branch (**parallel**), and click **New Branch**
- Select the branch in the drop-down list, and click **Checkout**
- Open the **Jenkinsfile** and set it to the content below, that implements the **parallel** capability

```
def DOCKER_IMG_BASENAME='demo-app'
def GIT_SHORT_CHANGESET='latest'

node('ssh-agent') {
    stage('Checkout code') {
        checkout scm
        GIT_SHORT_CHANGESET =
            sh(returnStdout: true, script: 'git rev-parse --short HEAD').trim()
    }
    withEnv(["PATH+MAVEN=${tool 'maven3'}/bin"]) {
        stage('Build') {
            sh "mvn package"
            junit '**/target/surefire-reports/*.xml'
            stash includes: '**/target/*.jar', name: 'application-binaries'
        }
    }
}

// Parallel Tasks
parallel (
    integrationtests: {
        node('ssh-agent') {
            stage('Integration Tests') {
                withEnv(["PATH+MAVEN=${tool 'maven3'}/bin"]) {
                    checkout scm
                    sh "mvn verify -fn"
                    junit '**/target/failsafe-reports/*.xml'
                }
            }
        }
    }
}
```

```

    }
  },
  dockerbuild: {
    node('docker') {
      stage('Docker Build') {
        checkout scm
        unstash 'application-binaries'
        sh "docker build -t ${DOCKER_IMG_BASENAME}:${GIT_SHORT_CHANGESET} ./"
      }
    }
  }
}

// Putting "input" inside a stage will
// allow the GUI to visualize it and show the popup
// If not you will have to go to the Console Output of the build
stage('Waiting Approval') {
  // We do not put "input" within a node
  // to avoid eating an executor while waiting
  timeout(time:1, unit:'DAYS') {
    input message: "Is it ok to deploy docker image?", ok: 'Deploy'
  }
}

// We do not put "build" within a node for same reasons as "input"
stage('Deploy') {
  // This will call the downstream job and block until it finishes
  // Unless disabled, downstream job status will be reported to the pipeline
  build job: 'demoapp-staging-deployer',
    parameters: [string(name: 'DOCKER_IMAGE',
      value: "${DOCKER_IMG_BASENAME}:${GIT_SHORT_CHANGESET}") ]
}

```

- Commit this file, and push the **new** branch to origin

TIP On the Pull/Push, don't forget to select the new branch instead of master, staying with **origin** remote

- See how the Multibranch pipeline reacts:
 - A new pipeline job is created and runs
 - You must validate/abort the Approval, in the new pipeline branch

demoapp-project

Branches (2)						
S	W	Name ↓	Last Success	Last Failure	Last Duration	
		master	2 min 33 sec - #3	1 hr 41 min - #1	2 min 1 sec	
		parallel	2 min 42 sec - #3	1 hr 33 min - #1	2 min 40 sec	

Icon: S M L



Legend  RSS for all  RSS for failures  RSS for just latest builds

Figure 7. A Multi-Branch Pipeline Project with 2 branches

That's all for this exercise !

Pipeline Shared Libraries

This last exercise covers an important feature of Jenkins Pipeline when your organization growth in size or practices maturity.

Let's take the following case:

- TeamA is currently using the Multi-Branch pipeline demoapp-project from the previous exercise
- TeamB wants to move quickly to a Pipeline-as-Code based workflow, and asked TeamA to share its pipeline
 - TeamB is going to work on a **different** project that uses the same tooling: a Java application built with Apache Maven and Docker, on the same Jenkins instance, with a **different** git repository
- **Challenge:** How to deal with the sprawl of the Pipeline per repositories ?
 - Code duplication: where is the "KISS" principle ?
 - Maintenance overhead: propagating a change would be costly and may not be immediate?
 - Collaboration: how to enable collaboration, security, validation, and avoid silos between teams (TeamA, TeamB, but also Jenkins admins, Business, Ops...)
 - Obviously, copy/pasting (or sending, or duplicating) the `Jenkinsfile` from TeamA will be easy for fast prototyping, but not a sustainable solution
- **Solution:** We're going to use Jenkins **Shared Libraries** to implement the code reusability that solves this challenge
 - Our example will simulate a Jenkins Admin team that provides a reusable primitive for the "Approval & Deploy" steps.

TIP

Always refers to the official reference documentation for Shared Library which is quite complete:
<https://jenkins.io/doc/book/pipeline/shared-libraries>

Prepare the Shared Library Repository

- We will use the `admin-scripts` private repository to store the Jenkins shared library

TIP

Direct URL to the repo is <http://localhost:5000/gitserver/butler/admin-scripts>

- Using the **WebIDE**, switch to the project related to `admin-scripts`
 - Stay on the default branch `master` that contains only a shell script
- Initialize the Shared Library:
 - Create a folder named `vars` on the root of the repository
 - Inside this new folder, create a new file named `approveAndDeploy.groovy`
 - Fill this Apache Groovy file with this content:

```
#!/usr/bin/env groovy

def call(String dockerImageToDeploy) {
    stage('Waiting Approval') {
        timeout(time:1, unit:'DAYS') {
            input message: "Is it ok to deploy ${dockerImageToDeploy}?",
                ok: 'Deploy'
        }
    }

    stage('Deploy') {
        build job: 'demoapp-staging-deployer',
            parameters: [string(name: 'DOCKER_IMAGE',
                value: "${dockerImageToDeploy}") ]
    }
}
```

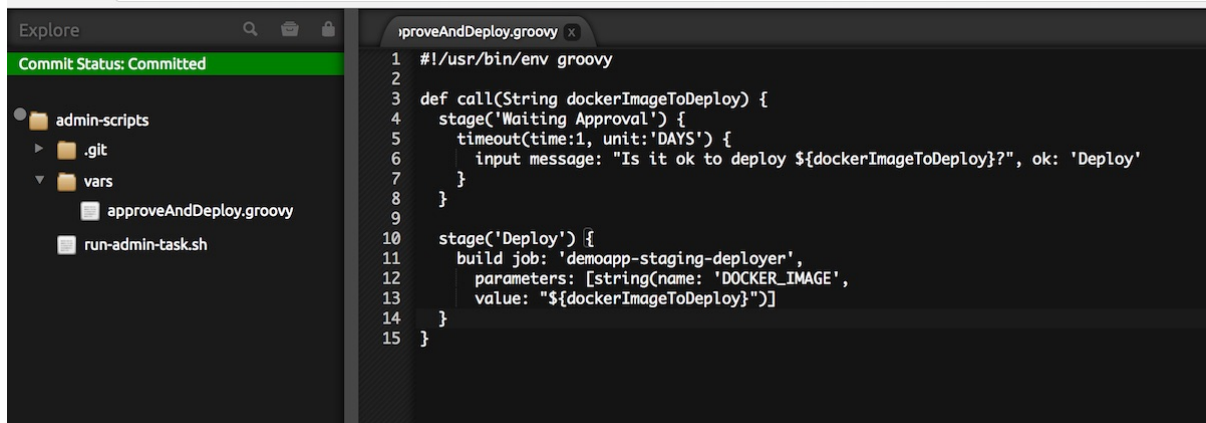


Figure 8. approveAndDeploy Shared Pipeline Library Code

- This code will run the 2 last steps of our previous pipeline exercise,
 - It expects one argument: the docker image to deploy
 - The source code filename will map to the method name, so it **must** be a valid Groovy/Java identifier.
 - The method `call` defines our code block
 - This code is **static**

TIP

We could have written complete classes, used Groovy code delegation, and a lot more.

For more examples, use the [Reference docs](#)

Our brand new Shared Library is now ready !

Configure the Shared Library in Jenkins

The library has been created in the git SCM. As a Jenkins admin, we now have to configure it one time.

- Browse the Jenkins **Configure System** page, from **Manage Jenkins**

TIP Direct URL: <http://localhost:5000/jenkins/configure>

- Scroll to the section name **Global Pipeline Libraries**
- Add a new Library with the following parameters:
 - **Name:** admin-lib
 - **Default version:** master
 - **Load implicitly:** No
 - **Allow default version to be overridden:** Yes
 - **Retrieval method:** Modern SCM
 - **Source Code Management:** Git
 - **Project Repository:** the {adminrepo-name} repository URL

TIP Direct URL: <http://localhost:5000/gitserver/butler/admin-scripts>

- **Credentials:** Select the credential used in Lab 3.2, for user butler. Should be `butler-gogs-creds`.
- You might set the **Repository Browser** to `gogs`
- Review the configuration below and **Save**

Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

Library Name	admin-lib
Default version	master <small>Currently maps to revision: 6a68fb368fdb0f38240cfbc02870aa06971776f5</small>
Load implicitly	<input type="checkbox"/>
Allow default version to be overridden	<input checked="" type="checkbox"/>
Retrieval method Modern SCM	
Source Code Management Git	
Project Repository	http://student-1.lab-certification.class-dryrun.cloudbees-training.com:5000/gitserver/butler/admin-sc
Credentials	butler/***** (Butler's credentials for Gogs) Add
Ignore on push notifications	<input type="checkbox"/>
Repository browser	gogs
URL http://student-1.lab-certification.class-dryrun.cloudbees-training.com:5000/gitserver/butler/adm	
Additional Behaviours	Add
<input type="radio"/> GitHub <input type="radio"/> Subversion <input type="radio"/> Legacy SCM	
Add	
Advanced...	
Delete	

Figure 9. Global Shared Pipeline Library System configuration

Test the Shared Library on the Multi-Branch Project

It is now time to adapt the Jenkins Pipeline to use this **Shared Library**.

- Using the **WebIDE**, switch to the demoapp project
 - We do not want to "break" the main branch; use the `parallel` branch
- We need to provide 2 changes:
 - Explicitly load the **Shared Library** with a Groovy annotation at the beginning of the source code:

```
@Library( 'admin-lib@master' ) _
```

TIP

A pipeline keyword named `library` has also been introduced recently (not in the certification area) to do the same thing, but wherever you want in the code.

- Replace the `Waiting Approval` and `Deploy` by a call to the `approveAndDeploy` method with parameters:

```
approveAndDeploy( "${DOCKER_IMG_BASENAME} : ${GIT_SHORT_CHANGESET} " )
```

- The new `Jenkinsfile` should look like the following:

```
#!/usr/bin/env groovy

@Library('admin-lib@master') _

def DOCKER_IMG_BASENAME='demo-app'
def GIT_SHORT_CHANGESET='latest'

node('ssh-agent') {
    stage('Checkout code') {
        checkout scm
        GIT_SHORT_CHANGESET =
            sh(returnStdout: true, script: 'git rev-parse --short HEAD').trim()
    }
    withEnv(["PATH+MAVEN=${tool 'maven3'}/bin"]) {
        stage('Build') {
            sh "mvn package"
            junit '**/target/surefire-reports/*.xml'
            stash includes: '**/target/*.jar', name: 'application-binaries'
        }
    }
}

// Parallel Tasks
parallel (
    integrationtests: {
        node('ssh-agent') {
            stage('Integration Tests') {
                withEnv(["PATH+MAVEN=${tool 'maven3'}/bin"]) {
                    checkout scm
                    sh "mvn verify -fn"
                    junit '**/target/failsafe-reports/*.xml'
                }
            }
        }
    },
    dockerbuild: {
        node('docker') {
            stage('Docker Build') {
                checkout scm
                unstash 'application-binaries'
                sh "docker build -t ${DOCKER_IMG_BASENAME}:${GIT_SHORT_CHANGESET} ./"
            }
        }
    }
)

approveAndDeploy("${DOCKER_IMG_BASENAME}:${GIT_SHORT_CHANGESET}")
```

- Save, commit and push this pipeline.
 - A build should kick off quickly, thanks to the webhook
 - Observe the new behavior by trying to abort and deploy; it should not have changed, outside the approval message
 - Take time to read the beginning of the build **Console Output** to understand how the Library is fetched at build time
 - You can try to create branches or tags on the `admin-scripts`, and point it on the annotation

TIP

Note that developer do not need to know which job to call to deploy, nor do they need to understand the "input" best practises with timeout and no node allocated

This is all for this

Journey Summary

We covered how to view our *legacy* implementation of a CD pipeline, and the new implementation with a Jenkins Pipeline, using a Multi-branch pipeline.