

JENKINS SHARED LIBRARY :

What is jenkins shared library ? Why we need it?

Eg: we have a 10 microservices in java-maven all these micro services contribute to one application . In jenkins we would need multi branch pipelines for each of those micro services. We would build each micro service as a separate application .

That means u have multiple projects that have there own jenkins files . That are building a pipeline in jenkins also some of those microservices are also java maven application and we are building docker images from all of those , we have logic inside those jenkinsfiles that are 90 % same

```
script.groovy 520 bytes
```

script.groovy 520 bytes

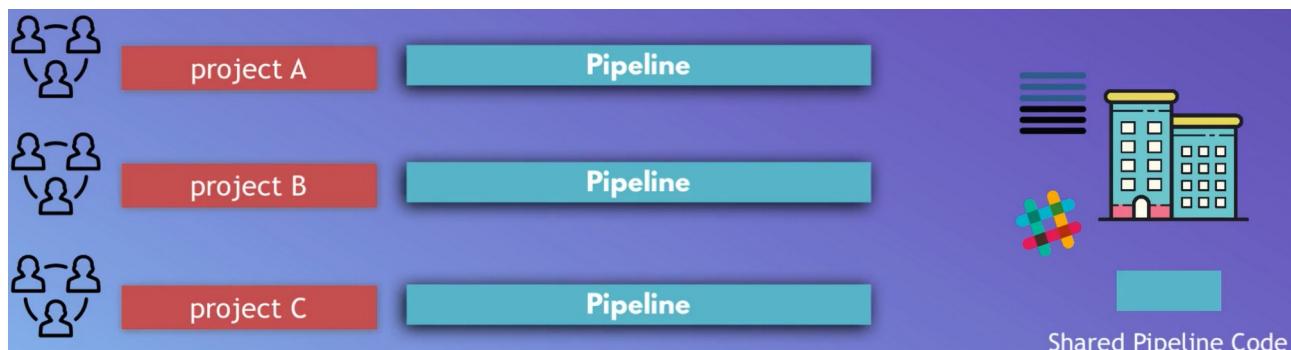
```
1 def buildJar() {
2     echo 'building the application...'
3     sh 'mvn package'
4 }
5
6 def buildImage() {
7     echo "building the docker image..."
8     withCredentials([usernamePassword(credentialsId: 'docker-hub-repo', passwordVariable: 'PASS', usernameVariable: 'USER')]) {
9         sh 'docker build -t nanatwn/demo-app:jma-2.0 .'
10        sh 'echo $PASS | docker login -u $USER --password-stdin'
11        sh 'docker push nanatwn/demo-app:jma-2.0'
12    }
13 }
14
15 def deployApp() {
16     echo 'deploying the application...'
17 }
18
19 return this
20
```

Edit Replace Delete

All of this logic would gonna be same for all of those microservices . That means we need to copy that same jenkins file and groovy script and replicate that all 10 different projects. Solution for that is jenkins shared library .

: JSL is a extension to a pipeline ,
It has its own repo which is written in groovy
We can write all the logic which is shared across many different application and reference that logic in jenkins file in each projects .

ANOTHER USE CASE OF JSL:



- ▶ May not use the same tech stack, but some logic is the same

- ▶ E.g. company-wide Nexus Repository

- ▶ E.g. company-wide Slack channel

- ▶ Don't replicate the code



:code reusability , easy maintenance , consistency and standardisation , faster pipeline creation . Improved collaboration in the company even they are not working in the same application .

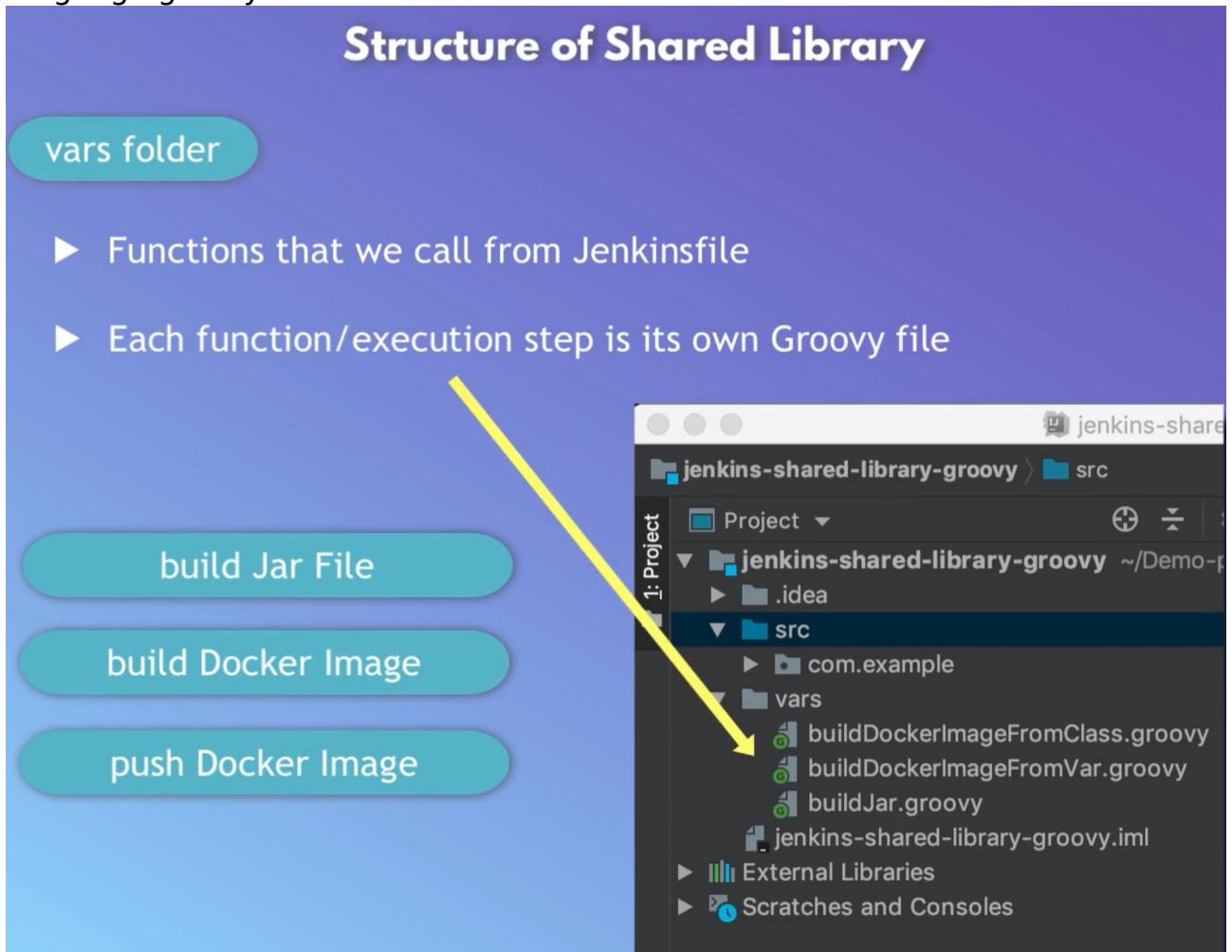
1. Create a Jenkins shared library
2. Make shared library available in jenkins
3. use shared library In Jenkins file in diff projects .

Create shared library project/repo :

- 1.create repo -> write groovy code there .
2. Make this shared library available for our project in jenkins .
- 3.use shared library in Jenkins to extend the pipeline .

Create repo :

1. Create a new project(named jenkins-shared-library) in ide -> choose language groovy



Src folder: helper code

Resource folder - use external library , non groovy files .

2. Create a var folder as new directory ->

Function name in the groovy file should be same in Jenkins shared library functions

```
script.groovy 520 bytes
1 def buildJob() {
2     echo "building the application..."
3     sh 'mvn package'
4 }
5
6 def buildImage() {
7     echo "Building the docker image..."
8     withCredentials([usernamePassword(credentialsId: 'docker-hub-repo', passwordVariable: 'PASS', usernameVariable: 'USER')]) {
9         sh 'docker build -t nanatwn/demo-app:jma-2.0 .'
10        sh 'echo $PASS | docker login -u $USER --password-stdin'
11        sh 'docker push nanatwn/demo-app:jma-2.0'
12    }
13 }
14
15 def deployApp() {
16     echo 'deploying the application...'
17 }
18
19 return this
20
```

so that it can refer easily .

Add : `#!/user/bin/env groovy` :on top of .groovy file : this will let your editor detect that u are working with a groovy script .

The screenshot shows two tabs in an IDE: 'buildJar.groovy' and 'buildImage.groovy'. Both files contain the following code:

```
#!/user/bin/env groovy
def call() {
    echo 'building the application...'
    sh 'mvn package'
}
```

2. Now create a repo in git hub so that we can use it in jenkins -> push our local repo to remote git repo. ->

Cd jenkins-shared-libraray -> git init -> git remote add origin repoURL ->git add .

->git commit -m"abc" -> git push -u origin master

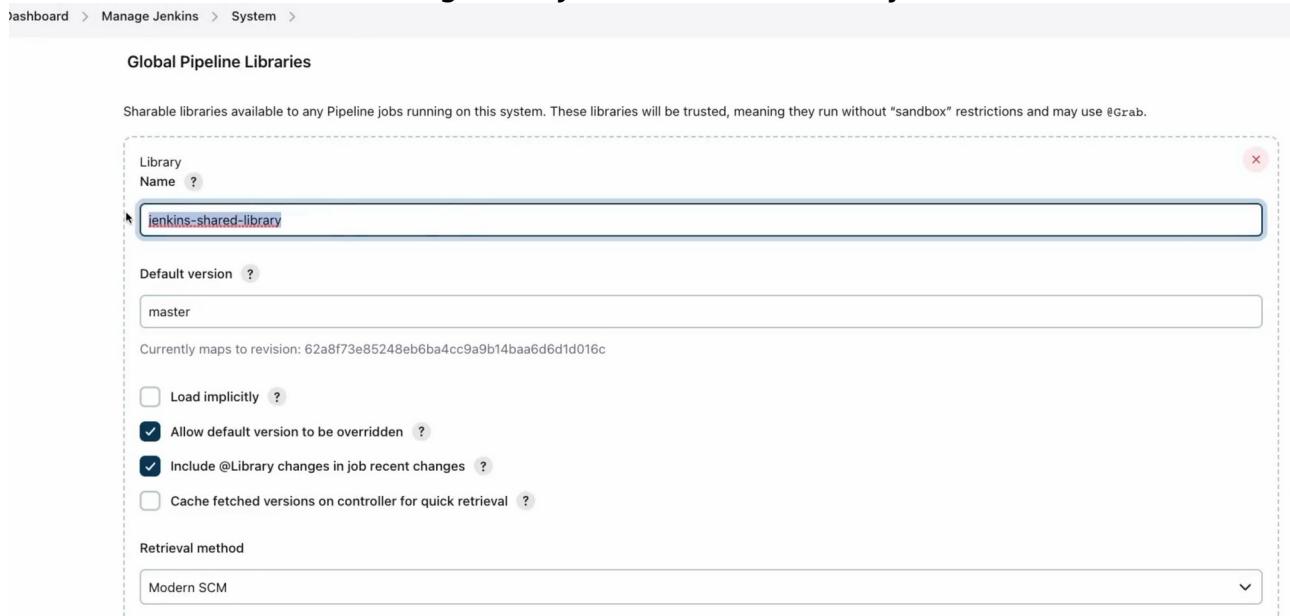
3. Make shared library globally available (to make it available to use in jenkins file):

Go to manage jenkins -> system -> global pipeline libraries -> name it jenkins-shared-library->set default version (master branch = any new change to the repo will be immediately available for whole jenkins pipeline) -> retrieval method (modern SCM)->

Choose git -> jenkins shared library repo URL -> add credentials for GitHub -> save .

This library will be available for jenkins files in all pipelines .

4. Use shared library in jenkins file : go back to java-maven app -> create new branch from master naming it as(Jenkins shared library) ->



Set the same name in the start of the jenkinsfile of global pipeline lib ->this will port the library port to the jenkinsfile

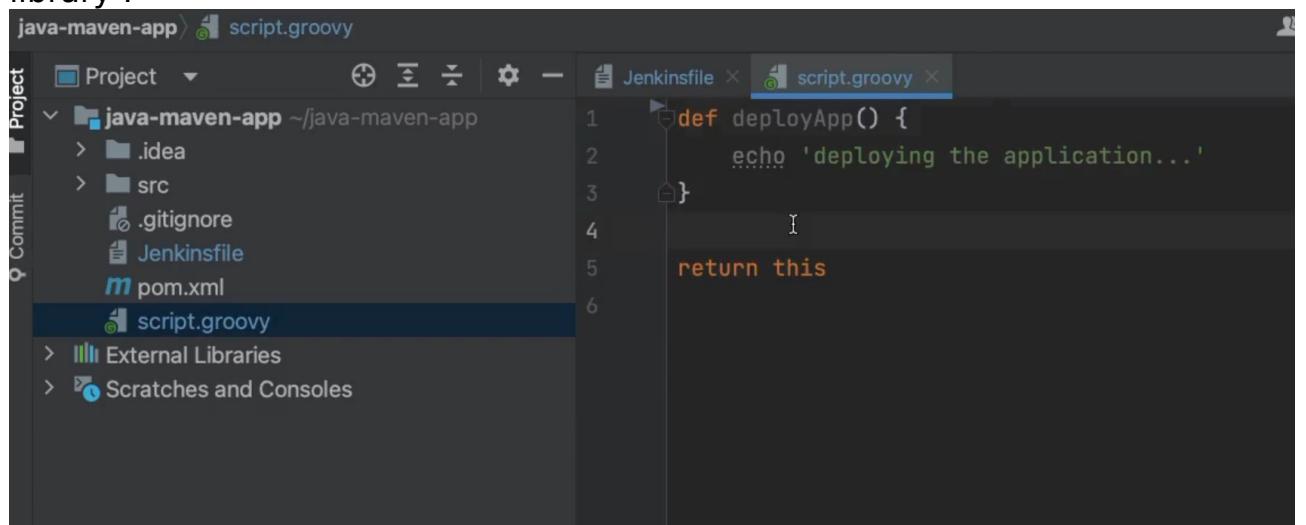
The screenshot shows the IntelliJ IDEA interface with a project named 'java-maven-app'. The 'Jenkinsfile' is open in the editor. The code defines a pipeline with a Maven build and a stage for loading a shared library:

```
#user/bin/env groovy
@Library('jenkins-shared-library')
def gv

pipeline {
    agent any
    tools {
        maven 'Maven'
    }
    stages {
        stage("init") {
            steps {
                script {
                    gv = load "script.groovy"
                }
            }
        }
        stage("build jar") {
            steps {
                script{
                    gv.buildJar()
                }
            }
        }
    }
}
```

If u don't have a **def gv** then u need to add **_** at last eg: @Library()**_**

Remove the script from script.groovy as we are defining them in shared library :



```
java-maven-app > script.groovy
Project Jenkinsfile script.groovy
java-maven-app ~/java-maven-app
src .gitignore Jenkinsfile pom.xml script.groovy
External Libraries Scratches and Consoles

def deployApp() {
    echo 'deploying the application...'
}

return this

stage("build jar") {
    steps {
        script{
            buildJar()
        }
    }
}
stage("build image") {
    steps {
        script{
            buildImage()
        }
    }
}
```

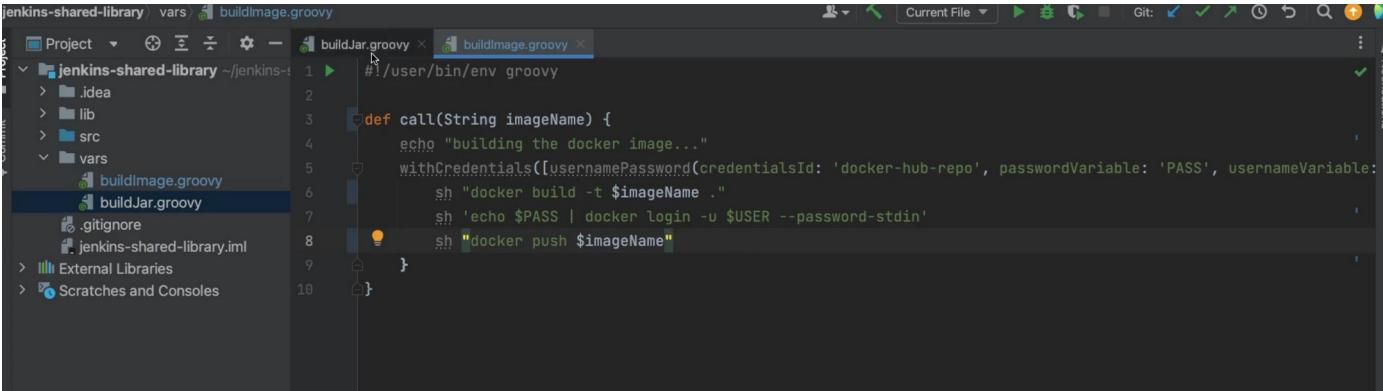
master ▾ jenkins-shared-library / vars /

Name
..
buildImage.groovy
buildJar.groovy

diff

These new function in jenkinsfile are referred from the shared library we defined

5. Commit these changes in remote repo in java-maven-app (jenkins-shared-lib branch)



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

def call(String imageName) {
    echo "building the docker image..."
    withCredentials([usernamePassword(credentialsId: 'docker-hub-repo', passwordVariable: 'PASS', usernameVariable: 'USER')])
        sh "docker build -t $imageName ."
        sh 'echo $PASS | docker login -u $USER --password-stdin'
        sh "docker push $imageName"
}
```

We have hard coded the image name here rename= nanatwn/demo-app : (imageName and tag)jma-2.0 this is not good it can happen each build can produce a new tag so this whole thing should actually be a parameter
Here we have parameterise the build image function so that we can pass imageName ,rep and the tag from Jenkins file .

The screenshot shows the IntelliJ IDEA interface with two tabs open: 'Jenkinsfile' and 'script.groovy'. The 'Jenkinsfile' tab contains Groovy code defining a pipeline. The 'script.groovy' tab shows the actual implementation of the pipeline steps. The code includes stages for building a JAR and a Docker image, and a deployment stage using a shared library named 'gv'. The 'gv' library is defined in the 'script.groovy' file.

```
steps {
    script{
        buildJar()
    }
}
stage("build image") {
    steps {
        script{
            buildImage 'nanatwn/demo-app:jma-2.0'
        }
    }
}
stage("deploy") {
    steps {
        script{
            gv.deployApp()
        }
    }
}
```

Here we have passed the parameters .

Dont forget to commit these change in jenkinsfile of jenkinssharedlib branch .
Also shared_library changes need to be committed

HOW TO TRIGGER JENKINS BUILD JOBS :

1. Manually :use cases if u have a separate pipeline that deploy to production
2. U want pipeline job to be triggered automatically when changes happens in git repo.

We want GitHub to notify jenkins that repo code is change u can build now . .
3. By scheduling them : trigger job/pipeline at scheduled time . Use case: for long running tests and test them self take more times . So u schedule them eg: in night when no one is working .

Automatic triggering :

For that we need jenkins plugins :

1. Do configuration in jenkins and next in our GitHub java-maven-app
2. Manage jenkins-> plugins ->install gitlab plugin(for build triggers)->mange Jenkins -> system-> u can see git Lab configuration ->

Enable authentication for '/project' end-point

GitLab connections

Connection name

A name for the connection

gitlab-conn

Gitlab host URL

The complete URL to the Gitlab server (e.g. http://g

https://gitlab.com/

Credentials

API Token for accessing Gitlab

- none -

Jenkins Credentials Provider



Jenkins

Gitlab access required

Advanced ▾

Save

Apply

When ever a application is talking to our gitlab account we can let that application to authenticate via a access token

Dashboard > Manage Jenkins > System >

Enable authentication for '/project' end-point

GitLab API token

Domain: Global credentials (unrestricted)

Kind: GitLab API token

Scope: Global (Jenkins, nodes, items, all child items, etc)

API token:

ID: gitlab-token

Description:

Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API.

You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.

Add a personal access token

Enter the name of your application, and we'll return a token for it.

Token name: jenkins

For example, the application using the token or the person holding the token, as it will be visible to all project members.

Expiration date: 2023-08-31

Select scopes: Scopes set the permission levels granted to the token.

- api** Grants complete read/write access to the API, including the registry.
- read_api** Grants read access to the API, including all groups.
- read_user** Grants read-only access to the authenticated user's public email, and full name. Also grants access to their profile page.
- read_repository** Grants read-only access to repositories on private networks.
- write_repository** Grants read-write access to repositories on private networks.
- read_registry** Grants read-only access to container registry images.
- write_registry** Grants write access to container registry images.

Create personal access token

Access Tokens

- Profile
- Account
- Billing
- Applications
- Chat
- Access Tokens**
- Emails
- Password
- Notifications
- SSH Keys
- GPG Keys
- Preferences
- Comment Templates
- Active Sessions
- Authentication Log
- Usage Quotas

Enable authentication for '/project' end-point

GitLab connections

Connection name

A name for the connection

gitlab-conn

Gitlab host URL

The complete URL to the Gitlab server (e.g. http://gitlab.

https://gitlab.com/

Credentials

API Token for accessing Gitlab

GitLab API token

Add ▾

Advanced ▾

[Save](#)

[Apply](#)

Dashboard > my-pipeline > Configuration

Configure

General

Advanced Project Options

Pipeline

GitHub project

GitLab Connection

gitlab-conn

Use alternative credential

Pipeline speed/durability override ?

Preserve stashes from completed builds ?

This project is parameterised ?

Throttle builds ?

Build Triggers

```

build.gradle
46
47 defaultTasks 'bootRun'
48
49 group = 'com.myapp'
50 version = '1.0.0-SNAPSHOT'
51
52 description =
53

package.json
1 {
2   "name": "test-js-app",
3   "version": "0.0.1",
4   "description": "Beta",
5   "private": true,
6   "license": "UNLICENSED",
7   "cacheDirectories": [
8     "node_modules"
9   ],
10  "dependencies": {
11    "@babel/polyfill": "7.8

```

maven.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven</groupId>
  <artifactId>test-maven</artifactId>
  <version>1.0-SNAPSHOT</version>
</project>

```

have multiple
gitlab lab accounts
that u all want to
connect with
jenkins u will have
a list of them and u
can choose which
git lab connection
your job should use

Project

- java-maven-app
 - .idea
 - src
 - .gitignore
 - Dockerfile
 - Jenkinsfile
 - pom.xml
 - script.groovy
- External Libraries
- Scratches and Consoles

Current File

pom.xml (java-maven-app) Jenkinsfile

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>java-maven-app</artifactId>
  <version>1.1.0-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>2.3.5.RELEASE</version>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

```



The screenshot shows the Jenkins Pipeline configuration page for a project named "my-pipeline". The "General" tab is selected. Under "Build Triggers", the "Build when a change is pushed to GitLab" option is checked. This triggers section includes "Push Events" (checked) and "Opened Merge Request Events" (checked). Other options like "Merge request" and "Tag push" are unchecked. At the bottom are "Save" and "Apply" buttons.

When ever push events for merge request event happens in gitlab it will send a web hook url to my Jenkins : and trigger the pipeline to build .

Second part is to configure git lab to automatically send notification to jenkins when ever a commit or a push happens
 Got to java-maven-app -> setting -> integration -> choose jenkins ->

The screenshot shows the GitLab settings page for a project named "java-maven-app". The "Integrations" section is selected. Under "Jenkins", the "Trigger" section has "Push" checked. The "Jenkins server URL" is set to "http://165.232.114.245:8080/". The "Project name" is "my-pipeline". The "Username" is "nana" and the "Password" is masked. At the bottom are "Save changes", "Test settings", and "Cancel" buttons.

There use the jenkins remote server url and the project name should be same as the job name and also username and password should be same as Jenkins user name and password

If u want to test without pushing any thing u can use test settings option : u can see on jenkins a build was triggered .

Configuring automatic trigger of jenkins jobs for multi branch pipeline :

We need a another plugin to trigger multi branch pipeline : for that we need another way :

Go to manage jenkins-> plugins -> available plugins -> multi branch scan webhook trigger -> go to dashboard -> inside multi branch job -> configuration -> build config-> scan multi branch pipeline trigger -> scan by web hook (this will let us to trigger build every time we push changes to repo)

This token will be used for the communication in git lab and jenkins or jenkins with any other git repo = name trigger token :any name eg : gitlabtoken .

Go to gitlab ->setting -> web hooks-> url paste :

Search page

Webhooks

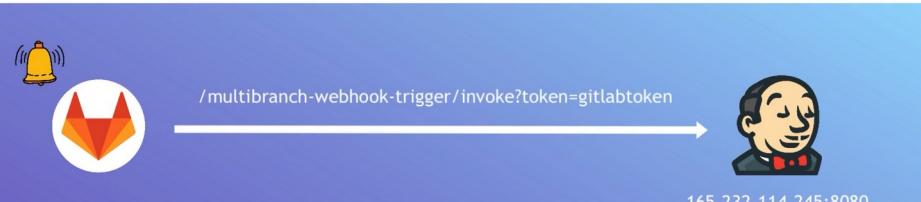
Webhooks enable you to send notifications to web applications in response to events in a group or project. We recommend using an integration in preference to a webhook.

URL
 URL must be percent-encoded if it contains one or more special characters.
 Show full URL
 Mask portions of URL
 Do not show sensitive data such as tokens in the UI.

Secret token

 Used to validate received payloads. Sent with the request in the `X-Gitlab-Token` HTTP header.

Trigger



165.232.114.245:8080

Dashboard > java-maven-app > Configuration

Configuration

Script Path ?

General

Branch Sources

Build Configuration

Scan Multibranch Pipeline Triggers

Periodically if not otherwise run ?
 Scan by webhook ?

Trigger token ?

The token to match with webhook token. Receive any HTTP request, `JENKINS_URL/multibranch-webhook-trigger/invoke?token=[Trigger token]` If a token match, than a multibranch scan will be triggered.
(from [Multibranch Scan Webhook Trigger](#))

Orphaned Item Strategy

Not Secure | 165.232.114.245:8080/job/microservice-user-auth/configure

Dashboard > java-maven-app > Configuration

Configuration

Script Path ?
Jenkinsfile

- General
- Branch Sources
- Build Configuration**
- Scan Multibranch Pipeline Triggers
- Orphaned Item Strategy
- Appearance
- Health metrics

Scan Multibranch Pipeline Triggers

Periodically if not otherwise run ?
 Scan by webhook ?
Trigger token ?
gitlabtoken

Webhooks

Webhooks enable you to send notifications to web applications in response to events in a group or project. We recommend using an integration in preference to a webhook.

URL
http://165.232.114.245:8080/multibranch-webhook-trigger/invoke?token=gitlabtoken
URL must be percent-encoded if it contains one or more special characters.
 Show full URL
 Mask portions of URL
Do not show sensitive data such as tokens in the UI.

Secret token
Used to validate received payloads. Sent with the request in the X-Bitlab-Token HTTP header.

Trigger
 Push events
 All branches
 Wildcard pattern
 Regular expression

VERSIONING YOUR APPLICATION :

Your app has a version and each build tool and package manager tool will maintain that version in its own main build file .

For gradle = build.gradle file , npm = package.json file , maven = pom.xml file .

So every package manager tool will have its own file where dependencies are listed plugins etc . And where the version of the application is also defined

```

1  #!/usr/bin/env groovy
2
3  pipeline {
4      agent any
5      tools {
6          maven 'Maven'
7      }
8      stages {
9          stage('increment version') {
10             steps {
11                 script {
12                     echo 'incrementing app version...'
13                     sh 'mvn build-helper:parse-version versions:set \
14                         -DnewVersion=\${parsedVersion.majorVersion}.\\\${parsedVersion.minorVersion}.\\\${parsedVersion.nextIncrementalVersion} \
15                         versions:commit'
16                     def matcher = readFile('pom.xml') =~ '<version>(.+)</version>'
17                     def version = matcher[0][1]
18                     env.IMAGE_NAME = "$version-$BUILD_NUMBER"
19                 }
20             }
21         }
22         stage('build app') {
23             steps {
24                 script {
25                     echo "building the application..."
26                     sh 'mvn clean package'
27                 }
28             }
29         }
30         stage('build image') {
31             steps {
32                 script {
33                     echo "building the docker image..."
34                     withCredentials([usernamePassword(credentialsId: 'docker-hub-repo', passwordVariable: 'PASS', usernameVariable: 'USER')]) {
35                         sh "docker build -t nanajanashia/demo-app:\${IMAGE_NAME} ."
36                         sh "echo \$PASS | docker login -u \$USER --password-stdin"
37                         sh "docker push nanajanashia/demo-app:\${IMAGE_NAME}"
38                     }
39                 }
40             }
41         }
42     }
43 }

```

<version>1.1.0-SNAPSHOT</version>

This is the version that you or your dev team has decide on to how to version your application .

This versioning is relevant when we are releasing the new version of our application .

The user that is using that application or webpage need to get latest version free from those bugs and to see latest feature added in new version .

The most common way to increase version is ->

-SNAPSHOT : is called a suffix that u can add to your version for more information snapshot is usually used for development version eg ; if u want to test but not release that versions

How to u change the version :

Manually ?

Do it automatically !!

Specially when u are building cicd jenkins pipeline automation for your application .

So when u commit change to jenkins jenkins build pipeline should increment the version and release new application .

Each build tool eg : npm , maven , gradle , etc they all have plugins or commands used for increment the version .

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>java-maven-app</artifactId>
    <version>1.1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>2.3.5.RELEASE</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

```

In cd java-maven-app terminal : we will write a command which will automatically upgrade the version defined here

: command is : ==> mvn build-helper:parse-version versions:set \ next line commands

: -DnewVersion=\\${parsedVersion.majorVersion}.\\$ {parsedVersion.minorVersion}.\\$ {parsedVersion.nextIncrementalVersion} versions:commit

//versions:commit (will remove the old copy of the pom.xml file and replace it with new copy with new version .

// it have majorVersion and nextMajorVersion , nextMinorVersion and minorVersion , incrementalVersion and nextIncrementalVersion.

So build-helper is a maven plugin that has a command called parse-version

This parse-version goes and finds the pom file finds the version tag and parses the version inside into 3 parts ->major , minor , increment .
Versions is a plugin : set is a command this will set the version btw these <version > tags.

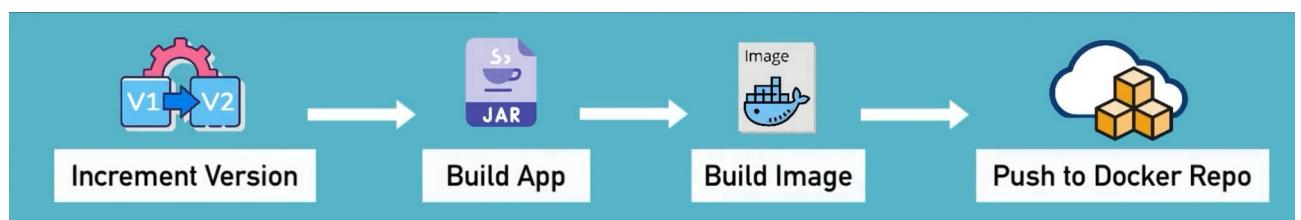
Below we have incremented the incremental patch version .

```
Processing change of com.example:java-maven-app:1.1.0-SNAPSHOT -> 1.1.1  
Processing com.example:java-maven-app  
    Updating project com.example:java-maven-app  
        from version 1.1.0-SNAPSHOT to 1.1.1
```

We have Same concept for different package manager /build tools :
Eg: read version from package.json ,increment and save back and Same for gradle also .

Now if it is a part of CICD we are introduction a new version in every new commit . ?

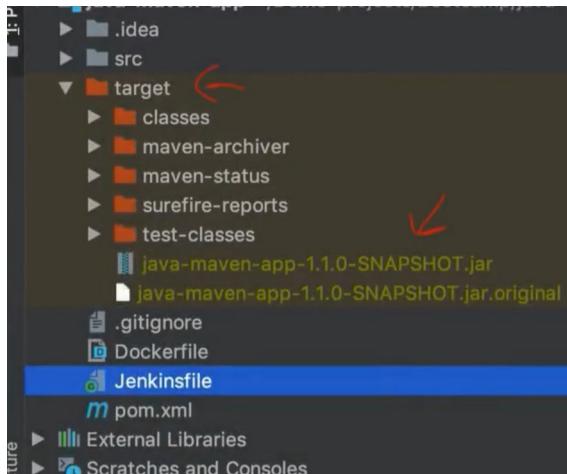
We have executed these commands locally how ever in real life u are not going to do it locally in your computer this should be part of CICD pipeline so these commands should actually be executed by jenkins when build process is running so as a next step we are going to integrate this step of setting a new version pom.xml file into jenkins file as part of build pipeline .



Increment application version on jenkins : java-maven-app:

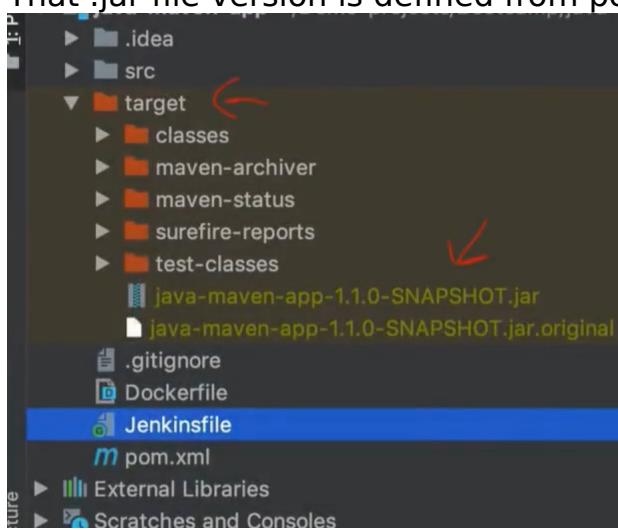
In jenkinsfile : build stage -> sh 'mvn package' = build a jar file with version from pom.xml

And this is where automatically generated build jar file is located in target folder .



And this is a jar file that is build using maven package
Ie: after running command : mvn package = u will see a target folder appeared

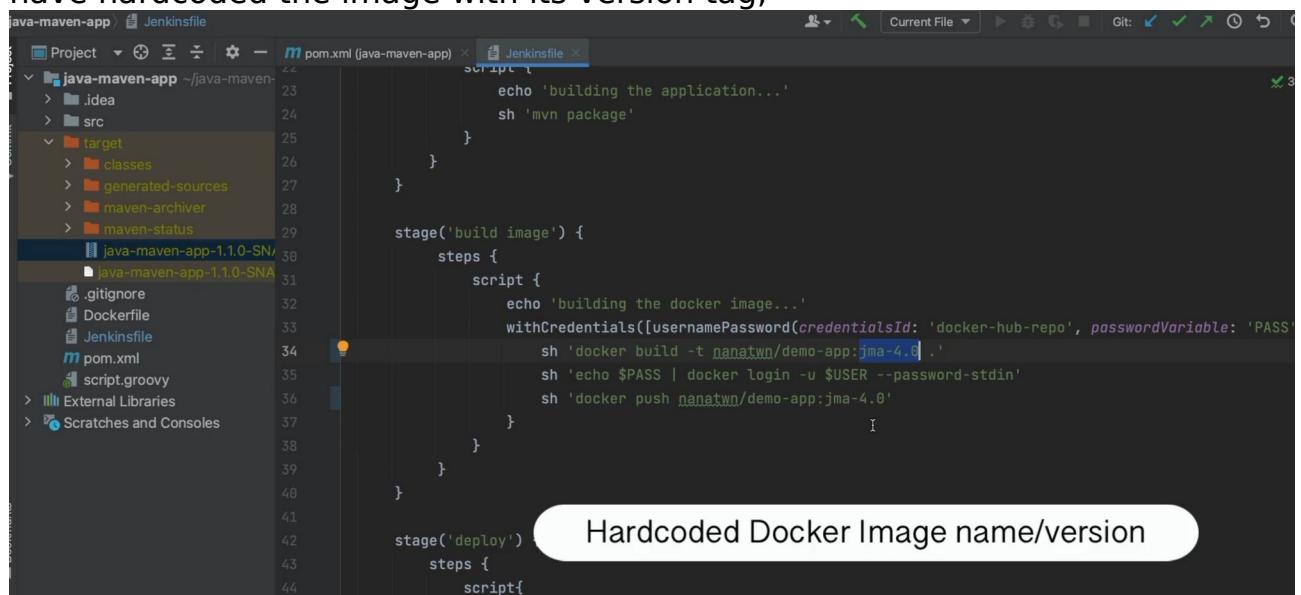
That .jar file version is defined from pom.xml version



That means we need to do version increment before application build so that we get a jar file with correct incremented version .

So before the build app stage we are going to create a new stage 'increment version'

Note: every time we are releasing a new version we are not releasing a jar file but rather docker image every time a application new version is released that means we need a version for our docker image but here we can see we have hardcoded the image with its version tag;



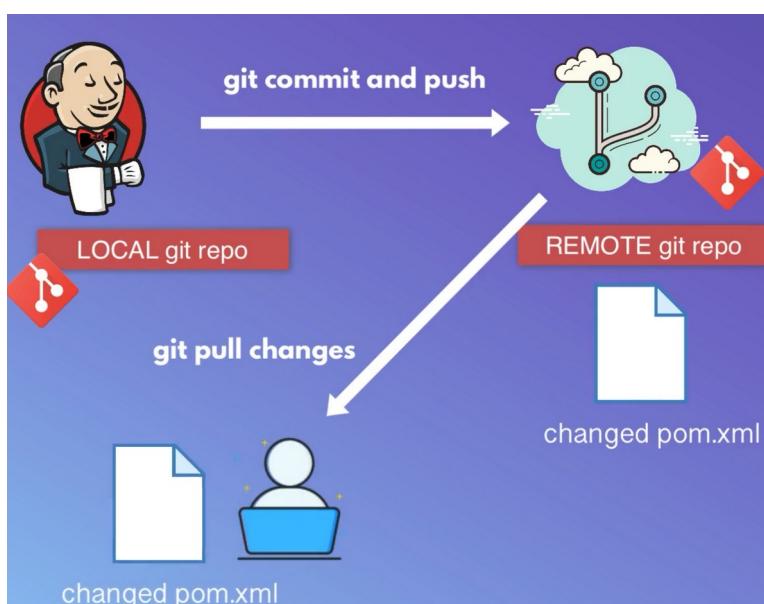
The screenshot shows a Jenkinsfile editor with the following code:

```
java-maven-app> Jenkinsfile
Project > + - pom.xml (java-maven-app) > Jenkinsfile
  > target
    > classes
    > generated-sources
    > maven-archiver
    > maven-status
      java-maven-app-1.1.0-SNAPSHOT
        java-maven-app-1.1.0-SNAPSHOT
          .gitignore
          Dockerfile
          Jenkinsfile
          pom.xml
          script.groovy
        > External Libraries
        > Scratches and Consoles
  23   echo 'building the application...'
  24   sh 'mvn package'
  25 }
  26 }
  27 }
  28
  29 stage('build image') {
  30   steps {
  31     script {
  32       echo 'building the docker image...'
  33       withCredentials([usernamePassword(credentialsId: 'docker-hub-repo', passwordVariable: 'PASS')])
  34         sh 'docker build -t nanatwn/demo-app:jma-4.0.'
  35         sh 'echo $PASS | docker login -u $USER --password-stdin'
  36         sh 'docker push nanatwn/demo-app:jma-4.0'
  37     }
  38   }
  39 }
  40
  41
  42 stage('deploy')
  43   steps {
  44     script{
```

A callout bubble points to the line `sh 'docker build -t nanatwn/demo-app:jma-4.0.'` with the text "Hardcoded Docker Image name/version".

So we need to increment this docker image version as well . Good practice is to use a application version for docker image version and we need to set it dynamically every time run a build pipeline

we
We
as
So of



dynamically every time run a build pipeline
need to pass this \$IMAGE_NAME variable in docker build command every time we build a new application version add " " its a variable not single cots .
we need to get the value major , minor or

increment version of app to map it with imageName version so how do we access these value and save it in a variable solution is by reading it from pom.xml and u need to tell it inside pom.xml I am looking for something like this -> **<version>1.1.0-SNAPSHOT</version>** and since we don't know the version yet add a regular expression in between version tag and save that value in a variable named it matcher .

Matcher is trying to match every singe element it can find in pom.xml and save it into an array and since we have only one version tag in pom.xml it will have only one element .

Ie: index 0 will be the version itself and index 1 will be the text inside version.

```
:def matcher = readFile('pom.xml') =~ '<version>(.+)</version>'
```

We can assign to version variable : def version = matcher[0][1]

Now we can use this variable version as image name

We want to append a build no to this version ->
“\$version-\$BUILD_NUMBER” , here build no is a env variable that jenkins makes available for us in pipeline job.

The screenshot shows the Jenkins Pipeline stage view for the project "jenkins-shared-lib". The stage view displays the following information:

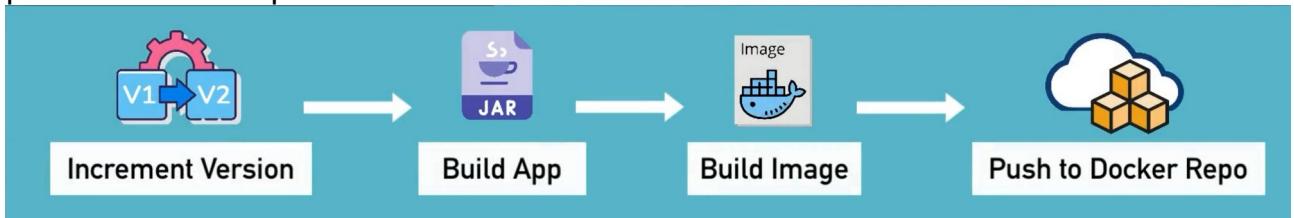
- Stage View:** Shows the stages: Declarative: Checkout SCM, Declarative: Tool Install, init, build jar, build and push image, and deploy.
- Average stage times:** (Average full run time: ~12s)

Stage	Average Time
Declarative: Checkout SCM	759ms
Declarative: Tool Install	73ms
init	275ms
build jar	4s
build and push image	4s
deploy	202ms
- Build History:** Shows builds #32, #31, #22, #4, #3, and #1 with their respective run times and commit counts.
- Permalinks:** Provides permanent links for each build.

These nos on bottom left 31,32.. are the build no. so every build has a no. and is unique for that specific build .

env.IMAGE_NAME = “\$version-\$BUILD_NUMBER” and we are appending that as a suffix to our version . And making this as a IMAGE_NAME ;

So now every time a new pipeline is build after a commit a version will be incremented and new docker image with new image tag created and then pushed to the repo .



REPLACE NEW VERSION IN DOCKER FILE :

Docker built command take a docker file and builds a new image so if we look at our docker file .

```
FROM amazoncorretto:8-alpine3.17-jre
EXPOSE 8080
COPY ./target/java-maven-app-1.1.0-SNAPSHOT.jar /usr/app/
WORKDIR /usr/app
ENTRYPOINT ["java", "-jar", "java-maven-app-1.1.0-SNAPSHOT.jar"]
```

The screenshot shows a Java Maven application project named 'java-maven-app'. The 'Dockerfile' tab is selected in the IDE. The Dockerfile content is as follows:

```
FROM amazoncorretto:8-alpine3.17-jre
EXPOSE 8080
COPY ./target/java-maven-app-1.1.0-SNAPSHOT.jar /usr/app/
WORKDIR /usr/app
ENTRYPOINT ["java", "-jar", "java-maven-app-1.1.0-SNAPSHOT.jar"]
```

These are the command that will be executed to build docker image .

And right now u can see we have hard coded the jar file name . So right now it is matching to this version 1.1.0-SNAPSHOT how ever if we run the pipeline and it increment the version to 1.1.1 these command will not work any more because there is no jar file with version 1.1.1 here in docker file so we need to make this dynamic not hard coded so make it just take a jar file from target folder java-maven-app* so this is a regular expression replace meant of any version after java-maven-app .

And the write jar file will be copies in the image file system
Now inside the image that jar file need to get executed right
So when ever we run this image as a container

Java , jar and jar file will get executed .
Instead of entry point -> CMD java -jar java-maven-app-* .jar

Final docker file look =>

The screenshot shows a code editor interface with a dark theme. On the left is a project tree labeled 'Project' containing files like .idea, src, target (which is expanded to show classes, generated-sources, maven-archiver, maven-status, and two versions of java-maven-app-1.1.0), .gitignore, Dockerfile (selected), Jenkinsfile, pom.xml, script.groovy, External Libraries, and Scratches and Consoles. The main pane displays the Dockerfile content:

```
FROM amazoncorretto:8-alpine3.17-jre
EXPOSE 8080
COPY ./target/java-maven-app-*.jar /usr/app/
WORKDIR /usr/app

CMD java -jar java-maven-app-*.jar
```

We need to make one more change in jenkins file :

When we change to version on pom.xml from 1.1.0 to 1.1.1 and run : mvn package

It will generate another jar file .

So this expression in docker file : COPY ./target/java-maven-app-*.jar /usr/app/ Will match both the jar file in target folder and we only need one in one image so java will not know which one to execute .

Means before we build the jar file we need to clean the target folder with old jar file so new one get generated with single jar file . : use command : mvn clean package

So this command delete the old jar file in target folder before it build new jar .

```
stage('build app') {
    steps {
        script {
            echo 'building the application...'
            sh 'mvn clean package'
        }
    }
}
```

So make change in jenkinsfile .

Stage: Increment version

- 👉 Increment patch version in pom.xml
- 👉 Read new version from pom.xml
- 👉 Put together a new Docker Image Tag
- 👉 Assign it to environmental variable

CONCLUSION :

```
stages {
    stage('increment version') {
        steps {
            script {
                echo 'incrementing app version...'
                sh 'mvn build-helper:parse-version versions:set \
                    -DnewVersion=\${parsedVersion.majorVersion}.\\${parsedVersion.minorVersion}.\\${parsedVersion.nextIncrementalVersion} \
                    versions:commit'
                def matcher = readFile('pom.xml') =~ '<version>(.+)</version>'
                def version = matcher[0][1]
                env.IMAGE_NAME = "$version-$BUILD_NUMBER"
            }
        }
    }
}
```

THESE ABOVE COMMANDS WILL BE EXECUTED ON JENKINS SERVER .

**STAGE BUILD : 1. CLEAN TARGET FOLDER , 2. PACKAGE JAR FILE .
With mvn clean package command jenkins server will clean the target folder and build the new application jar file .**

```
stage('build app') {
    steps {
        script {
            echo "building the application..."
            sh 'mvn clean package'
        }
    }
}
```

Then jenkins will build a docker image -> STAGE : BUILD DOCKER IMAGE : build docker image from docker file and tag image with repo URL and name .

Then jenkins will log in to docker repo and will push the new image to that repo .

```
stage('build image') {
    steps {
        script {
            echo "building the docker image..."
            withCredentials([usernamePassword(credentialsId: 'docker-hub-repo', passwordVariable: 'PASS', usernameVariable: 'USER')]) {
                sh "docker build -t nanajanashia/demo-app:${IMAGE_NAME} ."
                sh "echo $PASS | docker login -u $USER --password-stdin"
                sh "docker push nanajanashia/demo-app:${IMAGE_NAME}"
            }
        }
    }
}
```

Now commit these change to your git repo(in jenkins-job branch) : git add . ->git commit -m"" -> git push ->

Now of to jenkins and build jenkins-job pipeline ->

The screenshot shows the Jenkins Pipeline Jenkins-jobs stage view. On the left, there's a sidebar with links like Status, Changes, Build Now, View Configuration, Full Stage View, and Pipeline Syntax. Below that is the Build History section, which lists builds #12, #10, #9, #8, and #7, each with a timestamp (Jun 16, 16:41, 16:36, 15:31, 15:28) and a note about changes (1 commit, No Changes, No Changes). The main area is titled 'Stage View' and shows a grid of stages: Declarative: Checkout SCM (914ms), Declarative: Tool Install (82ms), increment version (3s), build app (6s), build image (10s), and deploy (182ms). The 'increment version' stage is highlighted with a blue box. Above the grid, it says 'Average stage times: (Average full run time: ~23s)'.

COMMIT VERSION BUMPS FROM THE JENKINS

But what happens now if we make some change commit them to git repo then re run this Jenkins-jobs build again . ?

We can still see the same versioning from 1.1.0 to 1.1.1 but why ?

So this change is present locally in jenkins this change never get committed to the git repo . So every time jenkins build runs . It always starts with this 1.1.0 version .

So it never ends up committing this 1.1.1 new version to the git repo there for in every build it has to start with that old version from the git repo .

How we can commit from jenkins to the git repo >?

After deploy stage in jenkins file ->

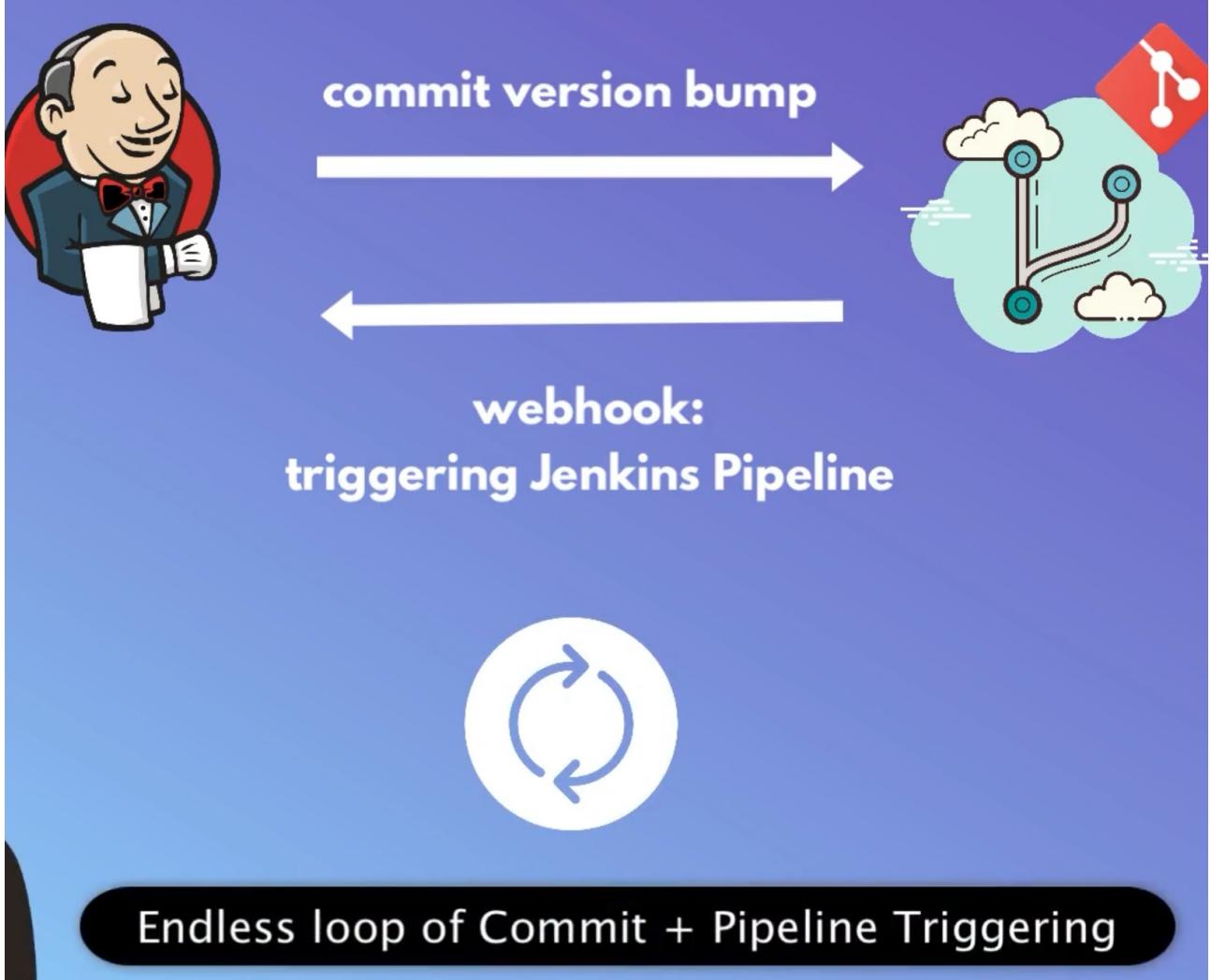
Add new stage -> first we need to access git from here there fore we need access credential from git repo just like to connect to docker hub in build image stage we needed credential to connect to docker hub -> use withCredentails for that

So we need credential id from jenkins credentials -> named gitlab-credentails(in my case) -> we need to tell jenkins what this origin is going to be

```
49      stage('commit version update') {
50          steps {
51              script {
52                  withCredentials([usernamePassword(credentialsId: 'gitlab-credentials', passwordVariable: 'PASS', usernameVariable: 'USER')) {
53                      // git config here for the first time run
54                      sh 'git config --global user.email "jenkins@example.com"'
55                      sh 'git config --global user.name "jenkins"'
56
57                      sh "git remote set-url origin https://\$USER:\$PASS@github.com/Vishalldwivedi/devOps-java_maven_app.git"
58                      sh 'git add .'
59                      sh 'git commit -m "ci: version bump"'
60                      sh 'git push origin HEAD:jenkins-jobs'
61                  }
62              }
63          }
64      }
65  }
```

We need to specify 'git push origin HEAD:jenkins-jobs' to specify to push the commit not to a commit hash but to the Jenkins-jobs.

One more thing to take care when we are comment from jenkins to git repo =>



Endless loop of Commit + Pipeline Triggering

How to solve this problem ??

We need to find a way to detect that commit is made up by jenkins and not up to stop this we hook automatic trigger of building pipeline . And ignore the jenkins commit to git repo . We can solve this problem by using a plugin

Go to manage jenkins -> plugins -> ignore committer strategy -> go to multi branch pipeline config -> build strategies is a option just got added through that new plugin ->

Ignore commiter strategy

```
'commit version update'){
steps {
    script {
        withCredentials([usernamePassword(credentialsId: 'gitlab-credentials', passwordVariable: 'PASS', usernameVariable: 'USER')]) {
            sh 'git config --global user.email "jenkins@example.com"'
            sh 'git config --global user.name "jenkins"'

            sh 'git status'
            sh 'git branch'
            sh 'git config --list'

            sh "git remote set-url origin https://\$${USER}:\$${PASS}@gitlab.com/twn-devops-bootcamp/latest/08-jenkins/ja"
            sh 'git add .'
            sh 'git commit -m "ci: version bump"'
            sh 'git push origin HEAD:jenkins-jobs'
        }
    }
}
```

Dashboard > java-maven-app > Configuration

Configuration

General

Branch Sources

Build Configuration

Scan Multibranch Pipeline Triggers

Orphaned Item Strategy

Appearance

Health metrics

Properties

Add ▾

Property strategy

All branches get the same properties

Add property ▾

Build strategies

Ignore Committer Strategy

Don't trigger builds for pushes by certain Git commit author (comma-delimited list of author emails) ?

jenkins@example.com

Allow builds when a changeset contains non-ignored author(s) ?

Add ▾

Save

Apply