Create a new note at 0:32

All lectures Sort by oldest

0:56 **2. Introduction & Installation** 2. Introduction to the course

Jenkins is basically an automation server. Which automates the workflow.

2:00 **2. Introduction & Installation** 14. Note: You should keep using putty

Docker: Install Docker Engine on CentOS

Docker-compose: Install the Compose standalone (docker.com)

1. `sudo curl -SL https://github.com/docker/compose/releases/download/v2.16.0/docker-`

`compose-linux-x86_64 -o /usr/local/bin/docker-compose`

2. `sudo chmod +x /usr/local/bin/docker-compose`

1. `docker-compose` (Use this command to test docker-compose is installed or not)

`sudo du -sh /var/lib/docker` => Tells the size of the docker occupying in our system or virtual machine or environment.

If we want to change permissions for a folder , we need to do '`sudo chown uid:gid <directory-name> -R`'

'`docker-compose up -d`' => Triggers the docker-compose file and run up the containers and perform what is in the docker-compose file.

'**docker logs -f <container-name>**' => This is used to check the logs of running container.

2:05 **2. Introduction & Installation** 14. Note: You should keep using putty

To start a docker service:

`sudo systemctl start docker`

To check the status of docker:

`sudo systemctl status docker`

To enable a service to start in the startup

`sudo systemctl enable docker`

If we are getting the /var/run/docker.sock error , we must add our current working user to docker group

2:10 **2. Introduction & Installation** 14. Note: You should keep using putty

If we want to create any local DNS for our IP address , either it can be of any service , we have to edit the hosts file , which is under the '`C:\Windows\System32\drivers\etc`' , there may be accessibility permissions for that for the users , so edit them in the properties section. And add one of the line in the end as

`192.168.0.132 jenkins.local`

It means ,

`<ip-address-that-we-have-to-map> <DNS-in-local-that-we want-to-use>`

After we saving it here, we can use the same DNS to login through putty , instead of ip address. As they are mapped on local hosts file.

If we want to connect to this jenkins ip address through any other terminal on the system or out of the system if it is hosted on the cloud , we can use the ssh , for example:

"`ssh username@ip-address`" or "`ssh username@domain-name`" => In here , we use "`ssh jeevan@jenkins.local`"

1:01 **2. Introduction & Installation** 15. Learn how to work with Docker and Jenkins

To stop a docker service, is created by the docker-compose, we need to do '`docker-compose stop`', which stops the Jenkins service.

If we want to start again , we need to do `docker-compose start`

1:38 **2. Introduction & Installation** 15. Learn how to work with Docker and Jenkins

If we want to restart a particular service, we need to do.

`docker-compose restart jenkins`

`docker-compose restart <service-name>`

1:49 **2. Introduction & Installation** 15. Learn how to work with Docker and Jenkins

If we want to delete everything related to a particular service, we need to use `docker-compose down`, which deletes everything about the service. But it won't delete the volume mount directory, whenever, we initialize the service, we get everything about the service back again, as files are persisted on the hard drive. The volume is declared under the docker-compose yaml file.

1:02 **3. Getting Started with Jenkins** 18. Hands On! Create your first Jenkins Job

A task in Jenkins called as a Job

2:11 **3. Getting Started with Jenkins** 18. Hands On! Create your first Jenkins Job

Whatever, we perform the task in the Jenkins and software's that we are used on Jenkins refers to the software's installed on Jenkins server or a container, that doesn't mean it is referring to the system software's or the Linux server from where we are running the container, it refers to the container and its software's installed in it.

2:35 **3. Getting Started with Jenkins** 18. Hands On! Create your first Jenkins Job

In short, Jenkins pipeline refers to the Jenkins container that we are running, but not the server where we are running the Jenkins container.

If we want to go inside the Jenkins container, we need to do,

`docker exec -it jenkins bash`

`docker exec -it <container-id> bash`

After execution of this command, we go into different shell, which is "inside the container." Whatever we execute in here doesn't affect the server where we run our container.

0:03 **3. Getting Started with Jenkins** 20. Redirect your first Job's output

For the first Jenkins Job, we gonna configure the Job, by adding a stage in "Freestyle project" under the "Build Steps" and add a build step of "Execute Shell."

```
echo "Hello World"
echo "Current user is $(whoami)"
echo "Current date and time is $(date)"
```

And run the Job by saving that, the details that you gonna get output of the Job is from inside the Jenkins container, which is running on system Linux server , but not from the outside container or from the system Linux server.

2:21 **3. Getting Started with Jenkins** 20. Redirect your first Job's output

We can also use variables with the Jenkins Build Steps as follows,

```
NAME=Jeevan
echo "Hello, ${NAME}. Current date and time is $(date)"
```

We can also redirect the output of build using

```
NAME=Jeevan
echo "Hello, ${NAME}. Current date and time is $(date)" > /tmp/info
```

All the files that we are gonna save or create by container will be and until dead in the container, unless they are mounted by the container.

Here our container that we created is its own independent system with its own memory and operating system. So the /tmp/info will not be created on the host system , it will be in the container.

```
NAME=Jeevan
echo "Hello, ${NAME}. Current date and time is $(date)" > /tmp/info
cat /tmp/info
```

2:58 **3. Getting Started with Jenkins** 21. Learn how to execute a bash script from Jenkins

Now we are creating a script on the local server, called 'script.sh' and give executable permissions for the file, else we will get permission denied error.

```
#!/bin/bash
NAME=$1
LASTNAME=$2
echo "Hello, $NAME $LASTNAME"
```

So, now if we want to use this file in the Jenkins Job, we must copy or create a similar file into the container, else we need to mount the file, for now we will create the file out of the container and copy that to the Jenkins container and execute the file in the Jenkins Job.

Now, normally, if we want to execute this shell script, we need to execute this file by going into the directory where this file exists.

`./script.sh Jeevan Sai`

We will get output as `Hello, Jeevan Sai`

4:11 **3. Getting Started with Jenkins** 21. Learn how to execute a bash script from Jenkins

If we want to copy a file from out of the container to inside , we need to use the following command.

`docker cp script.sh jenkins:/tmp/script.sh`

`docker cp <file-name> <container-name>:<path-to-store-in-container>`

`docker cp <file-path-with-filename> <container-name/id>:<path-to-store-in-container>`

Now if we do that, we will get the script inside the container, and we can execute that script from the Jenkins Job as the file is accessible to Jenkins user and present in Jenkins container.

`/tmp/script.sh Jeevan Sai`

Jenkins Job actually executes this Script and provides the output as mentioned in the script.

6:56 **3. Getting Started with Jenkins** 21. Learn how to execute a bash script from Jenkins

```
NAME=JEEVANSAI
LASTNAME=KANAPARTHI
/tmp/script.sh $NAME $LASTNAME
```

We can also use this as the build step, we can declare the variables in Jenkins Job build steps and pass it to the scripts as required.

4:07 **3. Getting Started with Jenkins** 22. Add parameters to your Job

If we want to give parameters dynamically, we need to configure the Job, such that, we build with parameters. In the configuration, we need to select "build with parameters" option or something like that, then we need to save the Job and come out then, we can see option like 'Build with parameters' in Jenkins job dashboard, so when we click that, we can see the Jenkins asks us to declare the values and we can configure default values for parameters while configuring the job. And we can build with parameters then.

In configuration, we have declared FIRST_NAME & LAST_NAME as the parameterized values, and in the script, we are using them as follows, we can use those parameters in the build steps using $VARIABLE_NAME.

5:24 **3. Getting Started with Jenkins** 22. Add parameters to your Job

Parameters are noting but interactive variables for the Jenkins Job.

2:22 **3. Getting Started with Jenkins**
23. Learn how to create a Jenkins list parameter with your script

To create an input, and have only certain options to select, we need to select the choice parameter in the configuration of the Job. So, we could select the possible values in the defined list. And we can display and use them as $VARIABLE_NAME.

6:58 **3. Getting Started with Jenkins** 24. Add basic logic and boolean parameters

We can also use the Boolean parameter as build runtime parameter. And pass the value for the following script through the executable shell script by the build step and this shell script is stored inside the container as we earlier done by storing the file inside the container.

```bash
#!/bin/bash

NAME=$1
LASTNAME=$2
SHOW=$3

if [ "$SHOW" = "true" ]
then
        echo "Hello, $NAME $LASTNAME"
else
        echo "Sorry, you choose to not display."
```

```
fi
```

Here, we are using input for SHOW as Boolean parameter, and we are conditionally generating the output to the Jenkins console.

For now, we have chosen NAME as a string parameter, LAST_NAME as choice parameter, SHOW as Boolean parameter for a single JOB.

Don't forget that we can copy the files of our local linux server to the container using `docker cp` command

1:03 **4. Jenkins & Docker** 27. Docker + Jenkins + SSH - II

We are going to create another service for Docker and use that service which is running in an individual container by ssh , so for that , we need to create an image first. We use the following Dockerfile.

```
FROM centos:7

RUN yum -y install openssh-server

RUN useradd remote_user && \
echo "1234" | passwd remote_user --stdin && \
mkdir /home/remote_user/.ssh && \
chmod 700 /home/remote_user/.ssh
```

Now we have to create a ssh key for current user , so we could access other user by using and giving permissions for this user by the another user.

For generating an ssh key for current user (centos 'VM') , we need to use

`ssh-keygen -f remote-key`

Then we are gonna get , two remote-key in folder that we executed , one is remote-key & remote-key.pub. Here remote-key is a private key and remote-key.pub is the public key.

1:59 **4. Jenkins & Docker** 27. Docker + Jenkins + SSH - II

To make a connection through ssh by the ssh-key generated of current user.

5:48 **4. Jenkins & Docker** 27. Docker + Jenkins + SSH - II

```
FROM centos:7

RUN yum -y install openssh-server

RUN useradd remote_user && \
echo "1234" | passwd remote_user --stdin && \
mkdir /home/remote_user/.ssh && \
chmod 700 /home/remote_user/.ssh

COPY remote-key.pub /home/remote_user/.ssh/authorized_keys

RUN chown remote_user:remote_user -R /home/remote_user/.ssh/ && \
chmod 600 /home/remote_user/.ssh/authorized_keys

RUN /usr/sbin/sshd-keygen

CMD /usr/sbin/sshd -D
```

5:43 **4. Jenkins & Docker** 27. Docker + Jenkins + SSH - II

Here , basically , we are using the centos 7 , as the base image , and installing openssh-server to enable the ssh connections through and for the server , and later we are creating a user , creating a password for that , we also creating a home ssh directory and declaring permissions for that ssh folder only to access by the remote user. And later we are copying our linux terminal ssh public key into the centos newly created image authorized keys , so there will be passwordless ssh connection , and later , we are changing the ownership of ssh folder recursivily for the ssh directory with our new user 'remote_user'. And give only read and write permissions for the remote_user rather than executable permission. And no other user will not have any permissions to use that.

6:43 **4. Jenkins & Docker** 28. Docker + Jenkins + SSH - III

Now, if we want to setup the password less connection and if we want to connect to another container having remote_host user, we need to go into Jenkins container.

From there, we need to use this command.

`ssh remote_user@remote_host`

And accept the authenticity, then our Jenkins user or any other user from which we triggered ssh , will be added to the known hosts file of the destination ssh server.

The name that we declare in docker compose file as `jenkins:` or `remote_host:` is simple like a DNS, which acts as a mask or pointer for their IP addresses, so when we establish ssh connection between the server, we can even 'ping' them, to see the network is interchangeable and transactable or not.

It is similar to,

`ping www.google.com`

9:34 **4. Jenkins & Docker** 28. Docker + Jenkins + SSH - III

So , now when we are in jenkins container , and want to login to the remote_host container , we use

`ssh remote_user@remote_host`

Then , we will be asked a password , as we kept one , so to do that passwordless , we need to use the remote-key (private) that we created. And use this following syntax to login without password.

`ssh -i remote-key remote_user@remote_host`

Here this , private key that we are passing will be authorized with the public key that we stored in authorized keys and login us without asking password , if the authorization success.

2:34 **4. Jenkins & Docker** 29. Learn how to install Jenkins Plugins (SSH Plugin)

In here, to connect to the remote_hosts from the Jenkins we going to need to download ssh plugin, for that we need to go to 'Manage Jenkins' and then to 'Plugin Manager' , there we can download the plugins what we need to , we are going to install the 'ssh plugin' , and then we need to click install without restart , and after everything is installed, we need to choose, 'Restart Jenkins when installation is complete and no jobs are running'

3:43 **4. Jenkins & Docker** 30. Integrate your Docker SSH server with Jenkins

The main moto in here with the ssh plugin is , we declared a docker-compose file , in which we have two containers running under the same network , so they can connect to one and another.

But when coming to host linux server on which containers are running , it can't connect to either of the containers through their hostnames , so as you remeber earlier , we created a ssh key for this server and generated private and public keys and copied the private key inside the centos image , so we can use that private key to communicate with the docker container directly.

3:48 **4. Jenkins & Docker** 30. Integrate your Docker SSH server with Jenkins

For integrating our docker ssh server with Jenkins, we need to download ssh plugin in Jenkins, which we already did. And later we need to add a ssh user in `Configure`

`System>SSH remote hosts`, and where we will be asked about the hostname that we want to add and credentials for this host. And to add credentials, we couldn't do it on spot, we have to create it under `Configure System > Manage Credentials` and add the username with the docker ssh server's private key, which is copied into authorized keys of remote_host.

Here, the main thing is, we are trying to connect to the remote_host which is of same network as Jenkins from the docker ssh server, we launched our container.

And remember the hostname and the username both are different; hostname is similar to the domain name. he defaults port number is 22 for the host.

0:48 **10. Jenkins & Maven** 88. Introduction: Jenkins & Maven

We typically use Maven because it is a build tool that allows generating libraries like jar files, downloading dependencies (libraries or JAR files) for each project, and following best practices for development. Maven also has a standard directory structure and a single configuration file (pom.xml) that contains most of the information required to build a project

0:23 **10. Jenkins & Maven** 88. Introduction: Jenkins & Maven

Maven is used to build the Java applications. Maven is also well used with Jenkins. We are going to learn how to build entire project, how to build the code, how to test it and more.

1:21 **10. Jenkins & Maven** 88. Introduction: Jenkins & Maven

For example, if we want to say, what is continuous Integration. For example, there will be n number of developers, who are working on a project, and in which for every commit they made the code gets updated, so the build must be triggered by the Jenkins Job, during build, Jenkins take help of tools like maven to build and generate a new Jar file for updated code and stays up to date.

1:34 **10. Jenkins & Maven** 88. Introduction: Jenkins & Maven

In this tutorial, we basically learn how to integrate maven, git and Jenkins and perform a real time operation for CICD.

1:03 **10. Jenkins & Maven** 89. Install the Maven Plugin

Maven Plugin & Maven Integration Plugin are the necessary plugins that we need for the building the maven project. We can download them through `Manage Jenkins>Manage Plugins>Available Plugins`

And Git Client Plugin & Git Plugin are also needed.

1:10 **10. Jenkins & Maven** 91. Learn how to clone a GIT/GITHUB repository from Jenkins

If our code is located on the GitHub then, we need to set the source code management in build configuration as 'Git'.

4:17 **10. Jenkins & Maven** 91. Learn how to clone a GIT/GITHUB repository from Jenkins

For each and every build we will have the workspace dedicated, in general, here, we will store all kinds of the source code which is downloaded through git, and artifacts and binaries build during runtime are stored in here, for each and every job, we will have a dedicated workspace.

We can go and check the workspace by going into the container using,

`docker exec -it <container-name> sh`

And go to the Jenkins home directory, by default it would be `/var/jenkins_home/`, under which we will have workspace, in that, we will have folders which are named after the Job names. Each and every folder dedicated to respective Job.

1:27 **10. Jenkins & Maven** 92. Learn how to build a JAR using maven

The tools that we want to use in our Job's, must be declared in 'Global Tool Configuration', we can install the tools by downloading the plugins first, and later download the software 'automatically' or download it manually in the container of Jenkins and give the path of the software binary.

And later, we use them in Job Configuration or pipeline script to the workflow.

4:07 **10. Jenkins & Maven** 92. Learn how to build a JAR using maven

"-B -DskipTests clean package" Here, we have used this maven command to execute in the repository, then as we know we don't have maven installed, so it downloads maven from the Apache and perform the build step.

After completion of the build step , we get a jar file created in target folder.

And this clears us to usage of git and maven for build in Jenkins file

4:16 **10. Jenkins & Maven** 92. Learn how to build a JAR using maven

The Jar file created is present under ,

`/var/jenkins_home/workspace/maven-job/target/my-app-1.0-SNAPSHOT.jar`

2:40 **10. Jenkins & Maven** 93. Learn how to test your code

Now in here, we are going to add a maven test command by configuring the Job. We done this in local jenkins in "maven-job" Job , so in there , we need to go to the "Build Steps" > Add Build Step > Invoke top-level Maven targets, and in there , we must give the maven version that we want to use in our build and this maven version that we are declaring here , we declare it under the 'Configure System' of Jenkins>Manage Jenkins. We can auto install maven in configure system.

In Job configuration we will have two inputs to provide to "Invoke top-level Maven targets"

1) Maven version (use the one , which we declared in configure system)

2)Goals (command that we want to execute)(Here we use "test" command)

3:03 **10. Jenkins & Maven** 93. Learn how to test your code

If we want to add any environmental variables for Jenkins , we can do that under

Jenkins > Manage Jenkins > Configure System > Global properties > Environmental variables.

Declare the necessary environment variables in here.

3:34 **10. Jenkins & Maven** 93. Learn how to test your code

In this maven job, we are doing mainly three step process.

1) Getting the latest code from the repository.

2) Building the code using maven.

3) Testing the maven code.

This is the basic workflow, the continuous integration pipeline does.

2:33 **10. Jenkins & Maven** 94. Deploy your Jar locally

We add one more build step of "Execute shell" after the maven build and the test. which actually deploys the jar.

```
echo "************************"
echo "Deploying JAR"
echo "************************"
java -jar /var/jenkins_home/workspace/maven-job/target/my-app-1.0-SNAPSHOT.jar
```

After adding this execute shell step in the build, we will get Hello World! as output it means the application got deployed successfully in the "container". We can do manual test for deployment by going into the container and executing this step

`java -jar /var/jenkins_home/workspace/maven-job/target/my-app-1.0-SNAPSHOT.jar`

We get `Hello World!` as an output if we execute this line in container, because the maven job is successfully built, and successfully tested in previous stages, so as the jar is correctly built, when we execute this jar the application is correctly deployed.

0:07 **10. Jenkins & Maven** 96. Archive the last successful artifact

If we want to have some "post build actions", among those one is "publish JUnit test result report", under which we can declare our result xmls file path, our success or failure maven build, this file stores the results of the build. We need to do these operations in Job configuration post build steps.

Default path of the result xml is under `target/surefire-reports/*.xml`

1:44 **10. Jenkins & Maven** 96. Archive the last successful artifact

And if we want to archive the artifacts on post of successful build, we need to configure the post build actions. In where we find 'Archive the artifacts' option, in where we can give the files to archive location, in default that would be `target/*.jar` , and select the option "Archive artifacts only if build is successful." And save the configuration and build the Job. And if the artifact successfully built and archived, we can see the archived one on the homepage.

2:36 **10. Jenkins & Maven** 97. Send Email notifications about the status of your maven project

To send the email notification post the build failure, we are going to request an email to the person who run the build, so that, we can keep track of the build's which take long time and fail.

We basically need to configure this "email notification" post build actions, under the job configuration.

Here, we configured in such a way that, we get a notification only when we got a build failure.

1:53 **11. Jenkins & GIT** 98. Create a Git Server using Docker

To get the system information, use the following command.

`cat /proc/cpuinfo`

`free -h` shows how much ram is used and currently available.

2:48 **11. Jenkins & GIT** 98. Create a Git Server using Docker

Important link to download Gitlab:

`https://docs.gitlab.com/ee/install/docker.html`

5:37 **11. Jenkins & GIT** 98. Create a Git Server using Docker

To make the email notifications work , we must have download the email based plugins under manage plugins , and then we need to edit our configuration under Configure system (Email Notification & Extended Email notification) , if we are using Gmail , we use SMTP server as smtp.gmail.com & SMTP Port as 465 , username is email id and password is the 'app' password , ensure we check in the Use SMTP Authentication and Use SSL under the Email Notification. And save the configuration after filling the required.

Follow this link for doubts : (`https://youtu.be/F01HOaklPeM`)

7:25 **11. Jenkins & GIT** 98. Create a Git Server using Docker

If we want to assign the hostnames to the IP addresses then , we must edit the hosts file of the system , and map the IP address with the domain name or hostname , so instead of using IP address , we can use the hostname in the PC browser.

8:00 **11. Jenkins & GIT** 98. Create a Git Server using Docker

port pattern `<external-port>:<internal-port>`

external port is the one we use to connect to the service; internal port is the one on which the service actually runs.

8:06 **11. Jenkins & GIT** 98. Create a Git Server using Docker

Login page displayed instead of password change page

1. Open Rails console. In order to do so, open the bash of you git-server container using command 'docker exec -ti <container name> bash

2. Then type the command 'gitlab-rails console'. This would open the Rails console.

8:11 **11. Jenkins & GIT** 98. Create a Git Server using Docker

3. Finally reset the password using the below set of commands. You can see these steps here "https://docs.gitlab.com/ee/security/reset_user_password.html#reset-your-root-password".

i.) Find the user either by username, user ID or email ID:

user = User.find_by_username 'root'

ii) Reset the password.

user.password = '<Your password>'

user.password_confirmation = '<Your password>'

iii) Save the changes

user.save!

iv) Exit the console. Now you can login to gitlab using your new password and username as root.

0:46 **11. Jenkins & GIT** 99. Create your first Git Repository

As Git lab has been created , we need to create a new group under the groups section , for now , we are naming the group as the `jenkins` , then we can create a new project , it means we are creating a new repository in here.

1:39 **11. Jenkins & GIT** 99. Create your first Git Repository

And after creating the group called Jenkins, we need to create a new repository under that group, for now we are creating the "maven" repository, maven as the repository name. This can be performed under projects section, where we can select the group and create a repository under that.

The new repository link will be similar to this `http://192.168.0.114:8090/jenkins/maven` Jenkins is the group name and maven is the repository name under that group. If we go in here, we can see our files and folders under our repository if exists. We can clone, push and pull, edit our files from here in GitLab.

0:20 **11. Jenkins & GIT** 100. Create a Git User to interact with your Repository

To add a user, for the GitLab, we must go to the profile section of GitLab, and then if you are currently admin, go to the "Administrator @root" option, then you will be redirected to a different page. And in there , we will have an option , "View user in admin area" , then you will be directed to "Admin Area" page , under which you can see "Overview" section under which you can see "Users" tab.

1:05 **11. Jenkins & GIT** 100. Create a Git User to interact with your Repository

Create a new user, and with password. The type of the user is "Regular" user, it need not to be an admin user.

1:45 **11. Jenkins & GIT** 100. Create a Git User to interact with your Repository

Now under the same admin section, we can see a tab called 'Projects' under the 'Overview', so, we need to select the repository, for which we want to add another user into, and then, you have to see something like 'edit' under the repository, and then after getting into repo, we have to see something like 'Manage Access'. under which we can 'invite a member', that we have just created earlier. And if needed, we can give the expiration date and invite the user and give his role either as admin or maintainer or guest or anything which determine his access permissions.

2:14 **11. Jenkins & GIT** 100. Create a Git User to interact with your Repository

Now if we go and check members of the repository, there will be the user that we added.

3:47 **11. Jenkins & GIT** 101. Upload the code for the Java App in your Repo

Now basically our aim is to have the maven code that we earlier executed from a GitHub server from other's repository to the GitLab server of our own repository , so for that , we have created a repository in GitLab called 'maven' which is now empty , we need to accumulate with the content from simple-java-maven-app which is in GitHub and public repository , so we are cloning both the repositories in the level of docker-compose.yml file , so we can transfer the data from them.

Commands used to clone them.

Command to clone a public repository

`git clone https://github.com/jenkins-docs/simple-java-maven-app.git`

This repo is of our own GitLab server that we hosted on our virtual machine.

`git clone http://jeevan:':sb_HXpVW.VwDQ5'@192.168.0.114:8090/jenkins/maven.git`

We store them both repositories in the same folder.

4:15 **11. Jenkins & GIT** 101. Upload the code for the Java App in your Repo

copy command we used to copy files from simple-java-maven-app folder to the maven folder by being in the maven folder.

`cp -r ../simple-java-maven-app/* .`

5:01 **11. Jenkins & GIT** 101. Upload the code for the Java App in your Repo

By being in the folder of maven which is cloned by the git repository, we need to do the following steps to commit and push the files into the GitLab main branch of our 'maven' repository.

`git add .` => to add all files to staging (ready to be commit) `git status` => to know the status of the branch (anything need to be commited or the branch is not / is upto date.

`git commit -m "Adds maven files"` => commit the files into our local repository with a commit message. files will be updated in the local.

`git branch` => to know , what branch we are in currently.

`git push origin <branch-name>` => to push the local repository's committed changes to the remote repository.

5:31 **11. Jenkins & GIT** 101. Upload the code for the Java App in your Repo

Now as code is shifted from the public online repository for which someone's owner to a GitLab repository for which we are owner. And now, we want to use this GitLab maven repository in our maven job, that we have executed earlier.

4:34 **11. Jenkins & GIT** 102. Integrate your Git server to your maven Job

Now we want to use the GitLab repository, which we created through git server that we created through docker-compose file. And we need to get http clone from the GitLab repository for the maven repository.

The original URL from the GitLab which refers repository looks like

`http://gitlab.example.com/jenkins/maven.git`

gitlab.example.com is the one , which we given as hostname during creating the container.

Now we need to refer this repository in Jenkins Job which does maven build.

But we need to refer the repository like this in network.

`http://git:80/jenkins/maven.git`

4:38 **11. Jenkins & GIT** 102. Integrate your Git server to your maven Job

There is one key point that we need to remember.

```
git:
        container_name: git-server
        image: 'gitlab/gitlab-ee:latest'
        hostname: 'gitlab.example.com'
        ports:
          - '8090:80'
        networks:
          - net
```

we can see we are declaring the 'git' which is the service name, and in here, we are also seeing that we have declared the network name, it means all the services that are under the network of 'net' can refer to this GitLab through the 'git' service.

And also, we can see the port numbers, one is external port number, and one is internal port number, external one, we use it for exposing out of the network through remote host and internal one we do directly attach to the server which is used to communicate within the network.

So this is the one , we use

`http://git:80/jenkins/maven.git`

And also remember to connect to this GitLab server, we need to set the credentials under the Manage Jenkins > Manage Credentials and add that one in the configuration.

1:20 **11. Jenkins & GIT** 103. Learn about Git Hooks

Issue: **can not find jenkins/maven project under /var/opt/gitlab/git-data/repositories/ in gitlab-server container**

```
root@gitlab:/# cd /var/opt/gitlab/git-data/repositories/
root@gitlab:/# ls
+gitaly/ .gitaly-metadata @hashed/
root@gitlab:/# cd \@hashed/

root@gitlab:/var/opt/gitlab/git-data/repositories/@hashed# pwd
/var/opt/gitlab/git-data/repositories/@hashed

root@gitlab:/var/opt/gitlab/git-data/repositories/@hashed# ls -la
total 16
drwxr-s--- 4 git git 4096 Nov 3 08:27 .
drwxrws--- 4 git git 4096 Sep 20 10:18 ..
drwxr-s--- 3 git git 4096 Sep 20 10:18 6b
drwxr-s--- 3 git git 4096 Nov 3 08:27 d4

root@gitlab:/var/opt/gitlab/git-data/repositories/@hashed# gitlab-rake gitlab:storage:rollback_to_legacy ID_FROM=1 ID_TO=50
Enqueuing rollback of 2 projects in batches of 200. Done!
```

1:25 **11. Jenkins & GIT** 103. Learn about Git Hooks

```
root@gitlab:/var/opt/gitlab/git-data/repositories/@hashed# ll
total 16
drwxr-s--- 4 git git 4096 Nov 3 08:27 ./
drwxrws--- 6 git git 4096 Nov 26 06:54 ../
drwxr-s--- 3 git git 4096 Sep 20 10:18 6b/
drwxr-s--- 3 git git 4096 Nov 3 08:27 d4/

root@gitlab:/var/opt/gitlab/git-data/repositories/@hashed# cd ..

root@gitlab:/var/opt/gitlab/git-data/repositories# ll
total 28
drwxr-sr-x 4 git git 4096 Nov 26 06:54 +gitaly/
drwxrws--- 6 git git 4096 Nov 26 06:54 ./
drwx------ 3 git git 4096 Sep 20 10:17 ../
-rw------- 1 git git 64 Sep 20 10:17 .gitaly-metadata
drwxr-s--- 4 git git 4096 Nov 3 08:27 '@hashed'/
drwxr-s--- 4 git git 4096 Nov 26 06:54 gitlab-instance-287738e2/
drwxr-s--- 4 git git 4096 Nov 26 06:54 jenkins/

root@gitlab:/var/opt/gitlab/git-data/repositories# cd jenkins/
```

1:30 **11. Jenkins & GIT** 103. Learn about Git Hooks

```
root@gitlab:/var/opt/gitlab/git-data/repositories/jenkins# ll
total 16
drwxr-s--- 4 git git 4096 Nov 26 06:54 ./
drwxrws--- 6 git git 4096 Nov 26 06:54 ../
drwx--S--- 5 git git 4096 Nov 26 06:54 maven.git/
drwx--S--- 4 git git 4096 Nov 3 08:27 maven.wiki.git/

root@gitlab:/var/opt/gitlab/git-data/repositories/jenkins# cd maven.git/

root@gitlab:/var/opt/gitlab/git-data/repositories/jenkins/maven.git# ll
```

```
total 32
drwx--S--- 5 git git 4096 Nov 26 06:54 ./
drwxr-s--- 4 git git 4096 Nov 26 06:54 ../
-rw------- 1 git git 21 Nov 3 08:27 HEAD
-rw------- 1 git git 101 Nov 26 06:54 config
drwxr-sr-x 2 git git 4096 Nov 9 08:55 info/
-rw-r--r-- 1 git git 187 Nov 9 08:55 language-stats.cache
drwxr-sr-x 27 git git 4096 Nov 9 08:55 objects/
drwxr-sr-x 6 git git 4096 Nov 9 08:55 refs/
root@gitlab:/var/opt/gitlab/git-data/repositories/jenkins/maven.git#
```

1:31 **11. Jenkins & GIT** 103. Learn about Git Hooks

To create a new hook, we must go into this directory, by going into the git-server container in which GitLab is running.

`/var/opt/gitlab/git-data/repositories/jenkins/maven.git`

0:05 **11. Jenkins & GIT** 104. Trigger your Jenkins job using a Git Hook

Now we are trying to add Git hooks for the Gitlab, so whenever any commit happens in the git repository, immediately there will be a trigger for the Jenkins job to get built.

Process for that:

Current working directory: `/var/opt/gitlab/git-data/repositories/jenkins/maven.git`

Create a new directory in maven.git folder called custom_hooks

`mkdir custom_hooks` And go into that directory `cd custom_hooks/` and create a file called post-receive , `touch post-receive`

0:10 **11. Jenkins & GIT** 104. Trigger your Jenkins job using a Git Hook

Add the following content into post-receive,

```bash
#!/bin/bash
#Get branch name from ref head
if ! [ -t 0 ]; then
        read -a ref
fi
IFS='/' read -ra REF <<< "${ref[2]}"
branch="${REF[2]}"
```

```

if [ $branch == "main" ]; then
        crumb=$(curl -u "jeevan:JeevanSai@123" -s 'http://jenkins:8080/crumbIssuer/api/xml?xpath=concat(//crumbRequestField,":",//crumb)')
        curl -u "jeevan:JeevanSai@123" -H "$crumb" -X POST http://jenkins:8080/job/maven-job/build?delay=0sec
        if [ $? -eq 0 ]; then
                echo "**** Ok"
        else
                echo "*** Error"
        fi
fi
```

0:15 **11. Jenkins & GIT** 104. Trigger your Jenkins job using a Git Hook

Give executable permission to the post-receive file. `chmod +x post-receive`

maven.git folder looks similarto this

```
root@gitlab:/var/opt/gitlab/git-data/repositories/jenkins/maven.git# ls
HEAD  config  custom_hooks  gitaly-language.stats  info  objects  refs
```

change teh permissions for the custom_hooks folder such that , gitlab can access it through git service and via all the services under the network 'net' , can access it.

`chmod git:git custom_hooks/ -R`

0:20 **11. Jenkins & GIT** 104. Trigger your Jenkins job using a Git Hook

Now , go into the maven repository folder that we cloned on the centos server and type

```
[jeevan@localhost maven]$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        url = http://jeevan::sb_HXpVW.VwDQ5@192.168.0.114:8090/jenkins/maven.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
```

We get link for the repository, this exact link we give in the Jenkins to access the repository during the build.

0:25 **11. Jenkins & GIT** 104. Trigger your Jenkins job using a Git Hook

Now if you got any error similar like this ,

`<title>Error 403 No valid crumb was included in the request</title>`

Solution:

Step 1: Go to jenkins > Manage Jenkins > Manage Plugins > Available Plugins > Download the Strict Crumb Issuer Plugin

Step 2: Go to jenkins > Manage Jenkins > Configure Global Security > CSRF Protection > Crumb Issuer > Select Strict Crumb Issuer > Under you will have Advanced option > Uncheck everything except "Prevent Breach Attack"

Issue solved :)

2:02 **11. Jenkins & GIT** 104. Trigger your Jenkins job using a Git Hook

Remember the post-receive file that we are declaring is under the gitlab server , (git-server container) So , now whenever there will be a new commit , due to this file it triggers this post-receive file and start the Jenkins Job as we are having the Jenkins cred in the file.

So , to conclude .

New Commit happens on repository > New Build triggered automatically.

0:23 **12. Jenkins & DSL** 105. Introduction: Jenkins DSL

Jenkins DSL: Create Jobs using Code.

0:47 **12. Jenkins & DSL** 105. Introduction: Jenkins DSL

How to build the Jobs using Groovy Code is the whole agenda of this Jenkins DSL.

0:26 **12. Jenkins & DSL** 106. Install the DSL Plugin

To work with DSL scripts and write the code using Groovy, we need to use Job DSL plugin in Jenkins.

0:16 **12. Jenkins & DSL** 107. What is a Seed Job in DSL?

Seed Job is a Job which creates new Jobs, we will configure all the Jobs that we want to create in the Seed Job. Seed Job is a parent Job that we will be creating.

1:37 **12. Jenkins & DSL** 107. What is a Seed Job in DSL?

To create a Seed Job, first we are going to create a new free-style project, under which we have to go to Build Steps stage, in there, there will be a build step "Process Job DSLs."

Under which we will have some options to create DSL's two of them are "Use the provided DSL Script" & "Look on Filesystem" , Use the provided DSL Script is the one, which we write script on the freestyle project itself, but Look on Filesystem is the one, we define the script as a file, and give the path of it.

Here the scripts that we will be creating for seed Job, will create more children Jobs as configured by the seed Job.

1:28 **12. Jenkins & DSL** 108. Understand the DSL Structure

Basic Example of Seed Job, the following is the DSL script for Job creation.

```
job("job_dsl_created") {

}
```

Paste this under `BuildSteps > Add Build Step > Process Job DSLs > Use the provided DSL script`

Now If we run a Job having this Seed Job created under DSL , then the current Job creates a Child Job "job_dsl_created" which we can see on the Jenkins Dashboard as a new Job , for which it shows it is part of " `Seed job: <parent-job-name>` "

2:11 **12. Jenkins & DSL** 108. Understand the DSL Structure

This is the Jenkins DSL API documentation: `https://jenkinsci.github.io/job-dsl-plugin/`

1:06 **12. Jenkins & DSL** 109. Description

To add a description to the child job from a seed Job, here 'job-dsl' is parent/seed job and 'job_dsl_example' is the child job, and the description is declared by its parent job.

```
job('job_dsl_example'){
        description('This is my awesome Job')
}
```

1:29 **12. Jenkins & DSL** 110. Parameters

DSL to add description, string, Boolean choice parameters.

```
job('job_dsl_example') {
        description('This is my awesome Job')
        parameters {
                stringParam('Planet',defaultValue = 'world' , description = 'This is the world')
                booleanParam('FLAG',true)
                choiceParam('OPTION',['option 1(defalut)','option 2','option 3'])
        }
}
```

1:58 **12. Jenkins & DSL** 111. SCM

When we want to add an scm with repository URL and branch name through the DSL scripts , we need to use similar script.

Add the below code in the dsl script of seed job , so it creates another child job with following scm configuration.

```
job('job_dsl_example') {
        description('This is my awesome Job')
        parameters {
                stringParam('Planet', defaultValue = 'world' , description = 'This is the world')
                booleanParam('FLAG', true)
                choiceParam('OPTION', ['option 1 (defalut)', 'option 2', 'option 3'])
        }
        scm {
                git('https://github.com/jenkins-docs/simple-java-maven-app','master')
        }
}
```

1:16 **12. Jenkins & DSL** 112. Triggers

To declare a trigger using DSL Scripts use the following code in seed job's DSL script.

```
job('job_dsl_example') {
        description('This is my awesome Job')
        parameters {
                stringParam('Planet', defaultValue = 'world' , description = 'This is the world')
                booleanParam('FLAG', true)
                choiceParam('OPTION', ['option 1 (defalut)', 'option 2', 'option 3'])
        }
        scm {
                git('https://github.com/jenkins-docs/simple-java-maven-app','master')
        }
        triggers {
                cron('H 5 * * 7')
        }
}
```

2:04 **12. Jenkins & DSL** 113. Steps

If we want to add some build steps through DSL and execute them through same shell, do the following:

```
job('job_dsl_example') {
        description('This is my awesome Job')
        parameters {
                stringParam('Planet', defaultValue = 'world' , description = 'This is the world')
                booleanParam('FLAG', true)
                choiceParam('OPTION', ['option 1 (defalut)', 'option 2', 'option 3'])
        }
        scm {
                git('https://github.com/jenkins-docs/simple-java-maven-app','master')
        }
        triggers {
                cron('H 5 * * 7')
        }
        steps {
                shell("""
                      echo 'Hello World'
                      echo 'Running script'
                      /tmp/script.sh
                      """)
        }
}
```

1:45 **12. Jenkins & DSL** 114. Mailer

If we want to add mailer in one of the build steps, we need to do the following.

```
job('job_dsl_example') {
        description('This is my awesome Job')
        parameters {
                stringParam('Planet', defaultValue = 'world' , description = 'This is the world')
                booleanParam('FLAG', true)
                choiceParam('OPTION', ['option 1 (defalut)', 'option 2', 'option 3'])
        }
        scm {
                git('https://github.com/jenkins-docs/simple-java-maven-app','master')
        }
        triggers {
                cron('H 5 * * 7')
        }
        steps {
                shell("echo 'Hello World'")
                shell("echo 'Hello World2'")
        }
        publishers {
                mailer('me@example.com',true,true)
        }
}
```

3:54 **12. Jenkins & DSL** 115. Recreate the Ansible Job using DSL

Documentation for using ansible in Jenkins DSL

`https://github.com/jenkinsci/ansible-plugin`

Check "Job DSL support" under the above link, where we can see multiple options that we can use for ansiblePlaybook under the build 'steps'.

7:39 **12. Jenkins & DSL** 115. Recreate the Ansible Job using DSL

Now as we want to create 'ansible-users-db-dsl' Job via he seed Job , we are going to add the following DSL under the previous added DSL 'jenkins-dsl-example'.

```
job('ansible-users-db-dsl') {
        description('Update the html table based on the input')
        parameters {
                choiceParam('AGE', ['21','22','23','24','25'])
        }
        steps {
                wrappers {
                        colorizeOutput(colorMap = 'xterm')
                }
                ansiblePlaybook('/var/jenkins_home/ansible/people.yml'){
                        inventoryPath('/var/jenkins_home/ansible/hosts')
                        colorizedOutput(true)
                        extraVars {
                                extraVar("PEOPLE_AGE",'${AGE}',false)
                        }
                }
        }
}
```

7:31 **12. Jenkins & DSL** 115. Recreate the Ansible Job using DSL

We can create DSL of all the jobs that we want to create and keep everything under DSL script of the seed Job, so consecutively multiple child jobs will be created based on the seed Job's DSL Script.

Another Job, which we created by the same seed job is:

```
job('job_dsl_example') {
        description('This is my awesome Job')
        parameters {
                stringParam('Planet', defaultValue = 'world' , description = 'This is the world')
                booleanParam('FLAG', true)
                choiceParam('OPTION', ['option 1 (defalut)', 'option 2', 'option 3'])
        }
        scm {
                git('https://github.com/jenkins-docs/simple-java-maven-app','master')
        }
        triggers {
                cron('H 5 * * 7')
        }
        steps {
                shell("""
                                              echo 'Hello World'
                                              echo 'Running script'
                                              /tmp/script.sh
                                      """)
        }
}
```

9:18 **12. Jenkins & DSL** 115. Recreate the Ansible Job using DSL

If we get an error called '`Destination /var/www/html not writable`' then, we must go into the 'web' container

`docker exec -it web bash`

`chown remote_user:remote_user /var/www/html -R`

6:44 **12. Jenkins & DSL** 116. Recreate the Maven Job using DSL

Basic maven project by DSL script

```
job('maven_dsl') {

        description('Maven dsl project')

        scm {
                git('https://github.com/jenkins-docs/simple-java-maven-app','master',{node -> node / 'extensions' << '' })
        }

        steps {
                maven {
                        mavenInstallation('MAVEN_JENKINS')
                        goals('-B -DskipTests clean package')
                }
                maven {
                        mavenInstallation('MAVEN_JENKINS')
                        goals('test')
                }
                shell('''
                        echo *****************RUNNING THE JAR***********************
                        java -jar /var/jenkins_home/workspace/maven_dsl/target/my-app-1.0-SNAPSHOT.jar
 ''')
        }
        publishers {
                archiveArtifacts('target/*.jar')
                archiveJunit('target/surefire-reports/*.xml')
                mailer('jeevansaikanaparthi@gmail.com',true,true)
        }
}
```

0:33 **12. Jenkins & DSL** 117. Version your DSL code using Git

If we want to use the GitLab repository from the service that we created using container, we must host the GitLab repository and later, we need to create a new project under that, when creating a new repository, we already have a group called 'Jenkins' that we created earlier, so use that, and keep the project slug as 'dsl-logs' and create a new repository.

0:36 **12. Jenkins & DSL** 117. Version your DSL code using Git

As we created a new repository, we needed to add users/members for the repository, so we can set the access permissions for the repository.

To add members into the repository, we do the following.

1) Go to GitLab Dashboard

2) Go under Projects, select the project that you want to edit.

3) Go to members under project information section.

4) Invite a new member to the repository and give him the accessibility for that repository.

1:30 **12. Jenkins & DSL** 117. Version your DSL code using Git

Cloning a new repository through GitLab:

```
git clone http://root::sb_HXpVW.VwDQ5@192.168.0.133:8090/jenkins/dsl-jobs.git
```

0:02 **12. Jenkins & DSL** 118. Magic? Create Jobs only pushing the DSL code to your Git server!

In Jenkins, DSL stands for **Domain Specific Language**. It is a plugin that allows you to define jobs in programmatic form with minimal effort. You can describe your jobs in Jenkins using a Groovy Based Language. Groovy is similar to Java but simpler because it's much more dynamic and it's a scripting language**1**. The "Jenkins Job DSL" is made up of two parts: The Domain Specific Language itself (which allows users to write Job DSL scripts in a groovy-based language); and a Jenkins Plugin which manages the Scripts and updating of the Jenkins jobs which are created and maintained as a result.

0:04 **12. Jenkins & DSL** 118. Magic? Create Jobs only pushing the DSL code to your Git server!

Now as we have pushed our DSL code into the repository, we are going to create a git hook, so, whenever we made a change in the job code in the git repository, the build will be triggered.

To create a git hook, we must go into the git-server container that we have created for the git-lab using.

```
docker exec -it git-server sh
```

2:26 **12. Jenkins & DSL** 118. Magic? Create Jobs only pushing the DSL code to your Git server!

If you want to establish git hooks, we already done that process earlier, so search in the notes below.

3:39 **12. Jenkins & DSL** 118. Magic? Create Jobs only pushing the DSL code to your Git server!

githooks

5:10 **12. Jenkins & DSL** 118. Magic? Create Jobs only pushing the DSL code to your Git server!

When we established the git hooks and made the configuration to fetch DSL from the local filesystem (local workspace), so to use that in Job's we must disable a security, in "`configure global security`". And in there, we must go under `Git plugin notifyCommit access tokens` , in where we need to disable "`Enable script security for Job DSL scripts`".

1:36 **12. Jenkins & DSL** 117. Version your DSL code using Git

Build a maven project using declarative pipeline.

When we are working on pipeline, and we want to use some tools that is pre-installed on Jenkins, but during the build it was not found, we need to declare its path in the environmental variable section of pipeline script.

```
pipeline {
    agent any
    environment {
        PATH = /opt/apache-maven-3.6.3/bin:$PATH
    }
    stages {
        stage('clone_repository'){
            steps{
                git credentialsId:'git_credentials',url:'https://github.com/ravdy/hello-world.git'
            }
        }
        stage('build code'){
            steps{
                sh "mvn clean install"
            }
        }
    }
}
```

1:41 **12. Jenkins & DSL** 117. Version your DSL code using Git

Else , if we downloaded the tool through the Global Tool Configuration , then use it's identifier to declare in the pipeline script.

```
pipeline {
    agent any
    tools {
            maven 'MAVEN_JENKINS'
        }
        stages {
            stage('clone_repository'){
                steps{
                    git
credentialsId:'git_credentials',url:'https://github.com/ravdy/hello-world.git'
                }
            }
            stage('build code'){
                steps{
                    sh "mvn clean install"
                }
            }
        }
    }
```

0:06 **13. CI/CD - Definitions** 120. Continuous Integration

CI/CD: It is nothing else but a *methodology/strategy to deploy code faster to production*.

InCase , when CI/CD isn't there , there will be a person who works to push the app source code and to deploy that into the production. then the person himself must do lot's of manual things , like testing it yourself , compiling , deploying , etc. As it is done by human , it is going to take so much time , and humans are prone to mistakes.

So , to reduce this overhead , CI/CD helps us , we define an entire workflow that will build , test and deploy automatically for us.

0:08 **13. CI/CD - Definitions** 120. Continuous Integration

CI/CD is a process , in which first starting at CI which is continuous integration , where we build and test our code and then we optionally pass that to continous delivery which deploys our built and tested app to dev/stg/qa environment to test it again. If the CI/CD process failed before , then it means there must be an issue with code , if they pass there won't be any errors most of the time , but it will be tested by human again in dev/qa/stg stage and finally deploy in the production.

1:46 **13. CI/CD - Definitions** 120. Continuous Integration

Release Pipeline:

Code > Continuous Integration > Continuous Delivery > Continuous Deployment

The above is the general release pipeline structure, to explain in detail, we write our code from developer side and commit that into dev/release as required. And using git hooks or any other way, we catch the request and start the CI process, basically in CI, we do build the code & test the code. Testing includes Unit testing, Integration Testing, Functional Testing. In Continuous delivery, we test the artifact that is generated by the build.

1:17 **13. CI/CD - Definitions** 121. Continuous Delivery

Continuous Delivery is an *optional step*, where we deploy the tested code into QA environment, and basically perform acceptance testing on that. So that, we can assure the test results are up to mark and can be processed into production environment.

1:14 **13. CI/CD - Definitions** 122. Continuous Deployment

After so many types of testing by CI/CD pipeline, we finally get the code, which is error free (at least expected). And later, we are going to deploy the code in some production environment, continuously, whenever there is a stage which passes CI/CD pipeline.

We can choose, which stages to be executed in the pipeline, for example developers, only work with CI pipeline, testers work with CD pipeline, on the whole CI/CD pipeline brings an error free code into the production.

0:18 **14. Jenkins Pipeline - Jenkinsfile** 123. Introduction to Pipeline

Pipeline is nothing but a workflow that we execute in a CI/CD process.

`https://www.jenkins.io/doc/book/resources/pipeline/realworld-pipeline-flow.png`

It can be any process that happens between the development and production stage.

In the workflow, we will have the stages, where each stage is a step of the workflow. generally, the first step of the workflow would be "SCM checkout", which downloads all the new code, and later will be the following steps.

Basic steps of the workflow:

1) SCM Checkout

2) Build/Launch (through Docker)

3) Test

4) Deploy

5) Workflow end

In the stage of testing, we may have integration testing, smoke testing or any other types of testing involved. And based on those test results we deploy the code in QA/Production.

1:17 **14. Jenkins Pipeline - Jenkinsfile** 124. Introduction to Jenkinsfile

There are two ways of writing Jenkinsfile - Scripted way and Declarative way.

Scripted way is the older way where we have to write a lot of logics using groovy DSL.

Declarative way is the newer way which comes with a lot of inbuilt functions.

1:45 **14. Jenkins Pipeline - Jenkinsfile** 124. Introduction to Jenkinsfile

Declarative pipeline is the code that we use to declare Jenkins file, this is mostly used by the newcomers to the Jenkins or mostly by the people who don't have understanding of coding. Whereas groovy pipeline, it involves code in it.

1:47 **14. Jenkins Pipeline - Jenkinsfile** 124. Introduction to Jenkinsfile

About Jenkins pipelines:

`https://www.jenkins.io/doc/book/pipeline/`

In Declarative Pipeline syntax, the pipeline block defines all the work done throughout your entire Pipeline.

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                //
            }
        }
        stage('Test') {
            steps {
                //
            }
        }
        stage('Deploy') {
            steps {
                //
            }
        }
    }
}
```

2:51 **14. Jenkins Pipeline - Jenkinsfile** 124. Introduction to Jenkinsfile

Pipeline validation better in declarative Jenkins syntax

* If we are having any syntactical issue in both declarative and scripted pipelines, declarative pipeline parses the whole code and catch the error first, faster than scripted pipeline, because in declarative pipeline, first syntax parsing is done, in scripted pipeline syntax parsing is not done initially, it will be done during the normal workflow of the groovy code execution.

Restart from stage option is present under declarative pipeline.

* We can start a stage from the mid pipeline, if declarative pipeline fails in earlier execution, but when coming to scripted pipeline, we need to execute the whole pipeline again, to check up the working of the pipeline.

Declarative pipelines have more options (at least, it makes us easy to use) in code to perform than that of scripted pipeline.

0:51 **14. Jenkins Pipeline - Jenkinsfile** 125. Install the Jenkins Pipeline Plugin

Install the "pipeline" plugin in the Jenkins.

1:02 **14. Jenkins Pipeline - Jenkinsfile** 126. Create your first Pipeline

In this declarative pipeline syntax, we can see an option like "agent" which have the value "any", it denotes, how the pipeline needs to be executed.

"agent any" => means any available agent will execute this pipeline.

3:22 **14. Jenkins Pipeline - Jenkinsfile** 126. Create your first Pipeline

Create a new item > pipeline project.

Basic pipeline code:

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

0:03 **14. Jenkins Pipeline - Jenkinsfile** 128. Retry

How to add multiple steps in a single build

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'echo "My first pipeline"'
                sh '''
                    echo "By the way, I can do more stuff in here"
                    ls -alh
                '''
            }
        }
    }
}
```

1:52 **14. Jenkins Pipeline - Jenkinsfile** 128. Retry

How to add retry option in Jenkinsfile

```
pipeline {
    agent any
    stages {
        stage('Timeout') {
            steps {
                retry(3) {
                    sh 'This message is not in echo Command , so it wont work'
                }
            }
        }
    }
}
```

2:29 **14. Jenkins Pipeline - Jenkinsfile** 129. Timeouts

There will be a scenario that we need to timeout the code, whenever, we think it is talking overtime than expected, so, we are going to define a time threshold in pipeline, so the code in that block crosses that time, then the pipeline will be timeout and the Job will be aborted.

```
pipeline {
    agent any
    stages {
        stage('Deploy'){
            steps {
                retry(3){
                    sh 'echo hello'
                }
                timeout(time:3,unit:'SECONDS'){
                    sh 'sleep 5'
                }
            }
        }
    }
}
```

The above job will be aborted, because the code will take 5 seconds to complete, but we give a timeout for 3 seconds.

1:38 **14. Jenkins Pipeline - Jenkinsfile** 130. Environment variables

Declaring environmental variables in Jenkins pipeline

```
pipeline {
    agent any
    environment {
        NAME = 'jeevan'
        LASTNAME = 'sai'
    }
    stages {
        stage('Build'){
            steps {
                sh 'echo $NAME $LASTNAME'
            }
        }
    }
}
```

1:30 **14. Jenkins Pipeline - Jenkinsfile** 131. Credentials

If we want to configure some secret text and use that in Jenkins pipeline , first we need to declare that secret text , and that will be happened under

`Dashboard > Manage Jenkins > Credentials > System > Global Credentials (unrestricted) > Add Credentials`

After creating a "Secret text" in here, we can use the same in the pipeline, the main benefit of this is the users other than admin can't find the secret text, just by looking at the code of the pipeline. unless they have some admin access because having editable and viewable access on the pipeline will also not able to see the value of this variable because, the following variable will be masked in the pipeline, but it will be used in the pipeline.

2:51 **14. Jenkins Pipeline - Jenkinsfile** 131. Credentials

```
pipeline {
    agent any
    environment {
        secret = credentials('SECRET_TEXT')
    }
    stages {
        stage('Example stage 1'){
            steps {
                sh 'echo $secret'
            }
        }
    }
}
```

We can handle the secret data using this option in the pipeline.

3:13 **14. Jenkins Pipeline - Jenkinsfile** 132. Post actions

Some of the post action examples of Jenkinsfile

```
pipeline {
    agent any
    stages {
        stage('Test') {
            steps {
                sh '''
                echo "Fail!";
                exit 1
                '''
            }
        }
    }
    post {
        always {
            echo 'I will always get executed :D'
        }
        success {
            echo 'I will only get executed if this success'
        }
        failure {
            echo 'I will only get executed if this fails'
        }
        unstable {
            echo 'I will only get executed if this is unstable'
        }
    }
}
```

1:52 **7. Jenkins & Security** 67. Intro - Learn how to Enable/Disable Login in Jenkins

To enable/disable the password prompt that we are having , which we access the Jenkins server on the VM or the clound , we must go to the

Manage Jenkins > Security > Authorization > Anyone can do anything

If we want to restrict the access to the users and for anyone , we can change it in here.

0:57 **7. Jenkins & Security** 68. Allow users to sign up

If we want to give user the sign-up capability for the Jenkins, we must navigate through the Jenkins.

`Jenkins Dashboard > Manage Jenkins > Security > Security Realm > Allow users to sign up`.

0:51 **7. Jenkins & Security** 69. Install a powerful security plugin

To improvise the Jenkins security in authorization, we are taking help from external plugin that is `Role-based Authorization Strategy`, download and install it in Jenkins.

2:05 **7. Jenkins & Security** 69. Install a powerful security plugin

And later that, we need to go through.

`Dashboard > Manage Jenkins > Security > Authorization > Role-based Strategy`.

And then, when we go to the home page, we can see a new option that is `Manage and Assign roles`, under "Manage Jenkins", which helps us to assign roles to the users, to access and view Jenkins.

0:42 **7. Jenkins & Security** 70. Create users manually in the Jenkins DB

If we want to manage users of Jenkins, we must go through the `Dashboard > Manage Jenkins > Users`, and we can also create a new user in here.

3:08 **7. Jenkins & Security** 70. Create users manually in the Jenkins DB

If we are going to use this external Role-based Authorization Strategy plugin, we need to explicitly declare the user permissions, else all other users other than admin, won't have access to the Jenkins server and its services.

1:50 **7. Jenkins & Security** 72. Assign the role that you created to a particular user

The core concept , is to create a role as you wish , if it isn't exist , for example , we can create a role , that have only read permissions or read write permissions . So , we can create a role under

Manage Jenkins > Manage and Assign Roles > Manage Roles , here we can see an option "Role to add" , we can give the name of the role , and assign permissions as required.

And secondly , we also need to assign this role to a user , only then user will get a role , so to assign the role to the user , we need to go through

`Dashboard > Manage Jenkins > Manage And Assing Roles > Assign Roles`

And in here , we can create a new role assignment , like , creating a new user and assigning a role for in under "Global Roles" of "Assign Roles".

2:28 **7. Jenkins & Security** 74. Learn how to restrict Jobs to users using Project Roles.

Sometimes, we want to create a rule such that, all jobs which starts with ansible word, should have some certain rules, these types of rules are called project roles. We can create a project role under

`Manage Jenkins > Manage and assign roles > Manage Roles > Item Roles`, where we need to create a new project role, and specify the pattern of the Job name that should be considered under a project.

As we remember, we have two roles, Global roles and Item roles, where we have declared a user group with specific permissions globally, and we are also having an item role, where we map the user, that we created earlier mapped to the pattern that we created earlier under Item Roles, so here, we will have two filters, one globally, and another internally.

`Manage Jenkins > Manage and assign roles > Assign Roles > Item Roles`

5:13 **7. Jenkins & Security** 74. Learn how to restrict Jobs to users using Project Roles.

If we want to display only some specific jobs to some specific users, we use these project roles (Item Roles).

0:52 **8. Jenkins Tips & Tricks** 75. Global environment variables in Jenkins

There are some default environmental variables, that we can use in Jenkins pipeline, these are no need to declare in prior, for usage in pipeline. These are global environmental variables.

`https://devopsqa.wordpress.com/2019/11/19/list-of-available-jenkins-environment-variables/`

3:18 **8. Jenkins Tips & Tricks** 75. Global environment variables in Jenkins

We can use these pre-defined default environmental variables anywhere in pipelines, without any prior declaration, because these are declared in default. We can use these variables even while sending a notification to user, either failure or success.

1:01 **8. Jenkins Tips & Tricks** 76. Create your own custom global environment variables

If we want to declare the global environmental variables, then we must declare them under

`Dashboard > Manage Jenkins > System.`

Under, where we can see global properties, we can see Environmental variables, where we can add list of variables to use them under many jobs as required.

0:09 **8. Jenkins Tips & Tricks** 78. Meet the Jenkins' cron: Learn how to execute Jobs automatically

If we map the IP address with domain name in local, but Jenkins doesn't know your local changes, so it shows the proxy error, so, we must declare the locally declared domain name in Jenkins, so Jenkins will have idea about that.

And for that, we need to go to,

`Dashboard > Manage Jenkins > Configure System / System > Jenkins URL`

Where we must give our URL, with our new domain name.

1:02 **8. Jenkins Tips & Tricks** 78. Meet the Jenkins' cron: Learn how to execute Jobs automatically

If we want our Jenkins Job to execute periodically at a particular time, without manual execution, then we are having an option like cronjob, we have to declare that under

`Dashboard > Any Job that we want to execute periodically > Configure.`

Under Build Triggers, we will be having an option like "Build periodically."

There, we can give the cronjob string, so the build happens periodically.

If we want to execute the Job, every other minute, we need to give the cronjob string like "* * * * *".

3:17 **8. Jenkins Tips & Tricks**

79. Learn how to trigger Jobs from external sources: Create a generic user

If we want to trigger a Job remotely from another Jenkins user, then we must give access for all the jobs to the Jenkins user, that we want to trigger remotely. So, for that, first we need to create a new Jenkins User, for that, we should navigate through.

`Dashboard > Manage Jenkins > Users > Create User`

After creating user, we need to create a role in Jenkins, such that, we can trigger the Job, from the remote user, by using Job triggering link, or we can access the Job remotely from that user through Jenkins.

So, next step is to create a role, we can create one, under

`Dashboard > Manage Jenkins > Manage and Assign Roles > Manage Roles`

Where under manage roles, we can create a role under Global roles and create one.

And later, we need to assign a role to the user, so all the actions of that role can be performed by the user. We can assign the role under,

`Dashboard > Manage Jenkins > Manage and Assign Roles > Assign roles.`

Under Global roles, assign the role to the user created or existed.

0:00 **8. Jenkins Tips & Tricks** 81. Trigger your Jobs from Bash Scripts (No parameters)

This is the referral link for creation of crumb and usage of it in a file, to trigger a Jenkins Job. Crumb is necessary to automate the Jenkins Job, while triggering it remotely through scripts.

`https://github.com/ricardoandre97/jenkins-resources/blob/master/jenkins/crumb.sh`

Line 1: is used to creation of crumb, we need to provide our username and password for creation of crumb and use that in scripts to trigger the Jenkins Job.

Line 2: is used to trigger the Job with help of the crumb.

Note: This is a way of triggering the Job, without parameters.

We can have this crumb.sh in Job B, and trigger Job A from Job B.

For Git hooks throwing 403 forbidden errors, refer lesson 80

6:15 **8. Jenkins Tips & Tricks** 82. Trigger your Jobs from Bash Scripts (With Parameters)

Now, if we want to trigger a Job by a URL, by passing parameters to a Job through a URL. We must follow the syntax, which was mentioned in Line 3

`https://github.com/ricardoandre97/jenkins-resources/blob/master/jenkins/crumb.sh`

And rest the process, is same, we can trigger the Job through this link, as we are triggering the Job through Build URL with the help of Crumb..

0:59 **15. CI/CD + Jenkins Pipeline + Docker + Maven** 133. Introduction

Here, we are going to build a real time project, where we are going to have a maven application, and create a docker image for running that maven application, test that application and push the tested image into the antifactory and deploy the latest image.

Here, we are going to use Jenkins software for CI/CD, docker for image building, and maven for building the application.

6:53 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

134. Learn how to install Docker inside of a Docker Container

This is the docker file, where you will have the commands to install: docker, Jenkins, ansible, docker-compose in a single image.

`https://github.com/ricardoandre97/jenkins-resources/blob/master/jenkins/pipeline/Dockerfile`

And docker-compose file, which is used for build of this Docker file, and launching other services is present in this link,

`https://github.com/ricardoandre97/jenkins-resources/blob/master/jenkins/docker-compose.yml`

9:15

15. CI/CD + Jenkins Pipeline + Docker + Maven

134. Learn how to install Docker inside of a Docker Container

To change the ownership of a file, we need to use the following command.

`sudo chown 1000:1000 /var/run/docker.sock`

sudo is used for root privileges.

8:46

15. CI/CD + Jenkins Pipeline + Docker + Maven

136. Build: Create a Jar for your Maven App using Docker

`docker run --rm -it -v $PWD/java-app:/app -v /root/.m2/:/root/.m2/ -w /app maven sh`

Command used to build the maven application using the maven image and mounting the source code from locally downloaded git repository into the image and downloading pom.xml dependencies inside the image and getting it out of the container with help of volume mounting. So, even if we re-execute the maven application, we will not take more time to download the dependencies, because they are already downloaded in previous executions and mounted to the local server.

`docker run --rm -v $PWD/java-app:/app -v /root/.m2/:/root/.m2/ -w /app maven mvn -B -DskipTests clean package`

3:30

15. CI/CD + Jenkins Pipeline + Docker + Maven

137. Build: Write abash script to automate the Jar creation

We can give the build process of the application and generate a Jar, by keeping the code in a file to execute the maven commands and build the application.

```
#!/bin/sh
```

```

echo "****************************************"
echo "*********** Building Jar **************"
echo "****************************************"

docker run --rm -v $PWD/java-app:/app -v /root/.m2/:/root/.m2/ -w /app maven "$@"
```

3:29 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

139. Build: Create a Docker Compose file to automate the Image build process

When we generated a Jar for the application and if we want to use that in another image, we must copy that in the Dockerfile in such a way.

```
FROM openjdk:8-jre-alpine

RUN mkdir /app

COPY *.jar /app/app.jar

CMD java -jar /app/app.jar
```

And if we want to build the image using docker-compose, then we must have such a docker-compose Dockerfile

```
version: '3'
services:
  app:
    image: "app:$BUILD_TAG"
    build:
      context: .
      dockerfile: Dockerfile-Java
```

If we want to trigger the build, then we need to use the following command.

`docker-compose -f <file-name> build`

1:32 **2. Introduction & Installation** 7. Install Docker

Install docker on centos:

`sudo yum install -y yum-utils device-mapper-persistent-data lvm2`

`sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo`

`sudo yum install docker-ce`

`sudo systemctl start docker`

`sudo systemctl enable docker`

`sudo usermod -aG docker jeevan_2`

1:22 **2. Introduction & Installation** 8. Install Docker Compose

Docker-Compose installation:

`https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-centos-7`

0:01 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

145. Create a remote machine to deploy your containerized app

We basically have two machines , in which one machine is used for deployment and other machine is used for triggering the deployment in another machine. So , for that , we basically need two machines , which have docker , docker-compose installed in them.

0:35 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

145. Create a remote machine to deploy your containerized app

We can connect from one machine to another machine through ssh , we can also connect to one and another password less , consider machine A & B , where A machine wants to connect with B machine.

So , we connect from one machine to another machine without password using ssh keys , which can be generated through the following command.

`ssh-keygen -f <file-name>`

ssh keys will be generated with the following given file name.

We will have two files, created, one is a private key file, which have the name <file-name> without any extensions, and we will have another file which is extended with ".pub" , which is the public file.

So , now if we want to connect to any other server passwordless , we must copy the contents of ".pub" file and keep that in authorized_keys of B server , which is present under /home/<user>/.ssh/ , if we don't have authorized_keys file created in B server , we can create one , and add the content in the file. And then relogin to both servers.

5:21 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

145. Create a remote machine to deploy your containerized app

And the command, that we want to use to login to another server using the private key file:

`ssh -i <private-key-file-name> <username>@<servername>`

we can use this command from the server A to connect to server b as we added the keys of server A to the server B authorized_keys

2:13 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

146. Push: Create your own Docker Hub account

Create an account in Dockerhub , remember the username and the password that you have created to login to the Dockerhub.

0:10 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

148. Push: Learn how to Push/Pull Docker images to your Repository

To create a repository in dockerhub , go to "Create a repository" option, where we need to give the repository name and create a repository. Then we will be redirected to a page which shows, something similar to this.

`docker push jeevan2001/maven-project:tagname`

0:39 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

148. Push: Learn how to Push/Pull Docker images to your Repository

If we want to push and pull the images through docker in the server, first we need to login to the docker, and then we can pull the images from that repository. And for that, we must do,

`docker login`

Then it will be prompting for the Username and Password, I have chosen my Username and password as

```
Username: jeevan2001
Password: Kanna@123
```

2:40 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

148. Push: Learn how to Push/Pull Docker images to your Repository

Later if we wanted to push our local image to the artifactory , then we must tag our local image using `docker tag` , such a way that it should match your artifactory name.

For example,

`jeevan2001/image-name:tagname`

And then we must push our image into the artifactory , using

`docker push jeevan2001/maven-project:tagname`

And then we can, pull those images by logging in any machine, first we must do, `docker login`

And then, we can do `docker pull jeevan2001/maven-project:tagname`

And use those images for our local.

5:12

15. CI/CD + Jenkins Pipeline + Docker + Maven

151. Deploy: Transfer some variables to the remote machine

We are giving a push automation script in this way.

```
#!/bin/sh

echo "*****************************************"
echo "************ Pushing Image *************"
echo "*****************************************"

IMAGE="maven-project"
echo "*********** Logging In ****************"
docker login -u jeevan2001 -p $PASSWORD

echo "************ TAGGING IMAGE *************"
docker tag $IMAGE:$BUILD_TAG jeevan2001/$IMAGE:$BUILD_TAG

echo "************ Pushing Image *************"
docker push jeevan2001/$IMAGE:$BUILD_TAG
```

6:12

15. CI/CD + Jenkins Pipeline + Docker + Maven

151. Deploy: Transfer some variables to the remote machine

deploy.sh

```
#!/bin/sh

echo maven-project > /tmp/.auth
echo $BUILD_TAG >> /tmp/.auth
echo $PASSWORD >> /tmp/.auth

sshpass -p 1234 scp /tmp/.auth jeevan_2@192.168.0.183:/tmp/.auth
```

We need to give executable permissions for this deployment script.

So, when we execute this script, the IMAGE_NAME, Jenkins BUILD_TAG and the password will be populated into a file /tmp/.auth , and transfer that file into another server, to a particular location.

3:26 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

152. Deploy: Deploy your application on the remote machine manually

For performing manual deployment, we need to execute the ./build.sh & ./push.sh & ./deploy.sh scripts, so we build the docker image, push that image into artifactory and deploy the credentials or the information needed to the other machine to pull the deployed image into the server B, where this file which contains the information is created in the server A.

9:06 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

152. Deploy: Deploy your application on the remote machine manually

Steps of manual deployment

```
export IMAGE_NAME=$(sed -n '1p' /tmp/.auth)
export TAG=$(sed -n '2p' /tmp/.auth)
export PASSWORD=$(sed -n '3p' /tmp/.auth)
```

We need to manually extract the details from the file that we have transfered from the Server A.

And login to the Docker hub manually (one time operation)

`docker login -u jeevan2001 -p $PASSWORD`

Then create a docker-compose.yml file for pulling the latest image and run the container.

```
version: '3'
services:
  maven:
    image: "jeevan2001/$IMAGE_NAME:$TAG"
    container_name: maven-app
```

Then if you want to trigger the image pull and trigger the running the container, we need to do, `docker-compose up`

Then the image will be pulled from our Docker hub artifactory and run that container. This is the manual way of deployment of user-built image.

Following is the content of `/tmp/.auth`

```
maven-project
10
Kanna@123
```

We are accumulating the IMAGE_NAME, TAG, PASSWORD into this file, which we transferred from server A to server B.

3:13 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

153. Deploy: Transfer the deployment script to the remote machine

Now we are creating a script for automation of publish in the server B.

publish.sh

```
#!/bin/sh

export IMAGE_NAME=$(sed -n '1p' /tmp/.auth)
export TAG=$(sed -n '2p' /tmp/.auth)
export PASSWORD=$(sed -n '3p' /tmp/.auth)

docker login -u jeevan2001 -p $PASSWORD

cd ~/maven && docker-compose up -d
```

Here, we take input from the /tmp/.auth which was coming from server A, and then login to the Docker hub and run the container using docker-compose.

2:40

15. CI/CD + Jenkins Pipeline + Docker + Maven

154. Deploy: Execute the deploy script in the remote machine

As we added the publish.sh in server B , but the pipeline runs from the server A , so for each and every pipeline execution , we must transfer the important details , that we want to use to execute publish.sh (i.e IMAGE_NAME & TAG & PASSWORD) , and we need to transfer this publish.sh script from server A to server B , we can keep this publish.sh static in server B , but we does in this way because , we may change the destination server , so we need to copy this deployment script (i.e publish.sh) , And then execute that publish.sh script in server B from server A , using ssh.

deploy.sh

```
#!/bin/sh

echo maven-project > /tmp/.auth
echo $BUILD_TAG >> /tmp/.auth
echo $PASSWORD >> /tmp/.auth

sshpass -p 1234 scp /tmp/.auth jeevan_2@192.168.0.183:/tmp/.auth
sshpass -p 1234 scp ./jenkins/deploy/publish.sh
jeevan_2@192.168.0.183:/tmp/publish.sh
sshpass -p 1234 ssh jeevan_2@192.168.0.183 /tmp/publish.sh
```

0:00

15. CI/CD + Jenkins Pipeline + Docker + Maven

156. Create a Git Repository to store your scripts and the code for the app

Updated Jenkinsfile :

```
pipeline {

    agent any

    stages {

        stage('Build') {
            steps {
                sh '''
                ./jenkins/build/maven.sh mvn -B -DskipTests clean package
                ./jenkins/build/build.sh
                '''
                }
        }

        stage('Test') {
            steps {
                sh './jenkins/test/maven.sh mvn test'
            }
        }

        stage('Push') {
            steps {
                sh './jenkins/push/push.sh'
            }
        }

        stage('Deploy') {
            steps {
                sh './jenkins/deploy/deploy.sh'
            }
        }
    }
}
```

3:59

15. CI/CD + Jenkins Pipeline + Docker + Maven

156. Create a Git Repository to store your scripts and the code for the app

Now, we need to push our logics and code to the GitHub.

So , for that , go to the folder , where you are having the code , and then

1) Initialize the repository

`git init`

2) Add your remote repository to the local through

`git remote add origin git@github.com:jsai2001/pipeline-maven.git`

3) Now we have to add the files that we want to push into the repository

`git add Jenkinsfile java-app/ jenkins/`

4) Commit your changes to the remote repository

`git commit -m "Initial Commit"`

5) We need to push the changes to the remote repository from the local repository

`git push -u origin master`

But before pushing the changes into the remote repository, we must have the write access for the repository, so we can establish that through adding our system ssh keys to the github. Our system ssh keys will be present under `/home/jeevan/.ssh/id_rsa.pub` , we need to copy this file to the ssh keys under github , so we will have writable access for our machine

Add your ssh keys in here.

"https://github.com/settings/ssh"

0:02 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

159. Create the Registry Password in Jenkins

After creating a pipeline-maven repository, we need to create a Jenkins pipeline, to automate the tasks. So, go to Jenkins, create a new item as a pipeline, and Pull the Code form SCM, give our git link in there.

This is our git repository.

`https://github.com/jsai2001/pipeline-maven/tree/master`

If we have any cloning issues or pushing issues, we need to configure ssh in Jenkins and in the server, we are working on , and add those ssh keys into the github under `https://github.com/settings/ssh`

1:35 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

160. Add the private ssh key to the Jenkins container

Until now, if you remember, we are passing our docker artifactory password manually for our manual building process, so as we want to automate this process, we can use Jenkins credentials for that purpose.

`Dashboard > Manage Jenkins > Credentials > System > Global Credentials > Add Credentials`

Add your secret test in here, I am naming it as dockerhub-pass

This key, I can use it in Jenkins, in this way

```
environment {
        PASSWORD = credentials('dockerhub-pass')
    }
```

This is the updated Jenkins script , until now

`https://github.com/jsai2001/pipeline-maven/blob/master/Jenkinsfile`

1:41 **15. CI/CD + Jenkins Pipeline + Docker + Maven**

160. Add the private ssh key to the Jenkins container

If you are not able to connect to another server through Jenkins, as we have included sshpass in the push process, we need to install "sshpass" in Jenkins container.

`apt-get install sshpass`

Then we will be able to run sshpass commands.

3:53 **15. CI/CD + Jenkins Pipeline + Docker + Maven** 161. Add post actions to Jenkinsfile

We need to add the post actions in the Jenkinsfile , as we want to store the successful artifacts and all the test reports.

`https://github.com/jsai2001/pipeline-maven/blob/master/Jenkinsfile`

4:05 **15. CI/CD + Jenkins Pipeline + Docker + Maven** 162. Execute your Pipeline manually

Now we are going to trigger the Job manually by going to Jenkins pipelines.

0:03 **15. CI/CD + Jenkins Pipeline + Docker + Maven**
164. Start the CI/CD process by committing new code to Git!

Now we want to trigger the Jenkins Job build automatically through Githooks , so for that, we need to give a webhook link to our GitHub repository, for which we need to monitor the repo, and execute the Jenkins Job, if there is any change in the repo (when a new push happens in the repo).

So, for that, if we are using some EC2 instance and hosting Jenkins, that would be fine, we can copy the Jenkins URL, I mean until `https://www.xxx.com/` , and add this link in repository settings under githook as `https://www.xxx.com/github-webhook/` , And we also need to configure Jenkins Job, we need to choose following option in build options of Jenkins Job.

`GitHub hook trigger for GITScm polling`

And now the magic happens, whenever we push any change into repository, the Job auto triggers, and the build happens.

0:05 **15. CI/CD + Jenkins Pipeline + Docker + Maven**
164. Start the CI/CD process by committing new code to Git!

But the issue comes, when we are hosting Jenkins on a localhost, refer the video 163, and follow the process, and by using that approach, if we try to deliver the GitHub webhook, we will get such error as mentioned in the link.

`https://stackoverflow.com/questions/42037370/jenkinsgithub-we-couldn-t-deliver-this-payload-couldnt-connect-to-server`

Then we must host our Jenkins server's localhost into ngrok and provide that link in the bitbucket as the GitHub-webhook link, which helps us to host our website through online for a limited amount of time with limited actions, please refer the above link for more details.

1:00 **5. Jenkins & AWS** 32. Introduction: MySQL + AWS + Shell Scripting + Jenkins

Here , we are going to have a project , where we are having an Jenkins user , where we are going to backup the MySQL data , and upload that object to internet and dump it to amazon s3. The main triggering for this job is through Jenkins.

6:02 **5. Jenkins & AWS** 33. Create a MySQL server on Docker

```
version: "3"
services:
  jenkins:
    container_name: jenkins
    image: jeevan2001/custom-jenkins-docker:v1.0
    ports:
      - "8080:8080"
    volumes:
      - "$PWD/jenkins_home:/var/jenkins_home"
    networks:
      - net
  remote_host:
    container_name: remote-host
    image: remote-host
    build:
      context: centos7
    networks:
      - net
  db_host:
    container_name: db
    image: mysql:5.7
    environment:
      - "MYSQL_ROOT_PASSWORD=1234"
    volumes:
      - "$PWD/db_data:/var/lib/mysql"
    networks:
      - net
```

```
networks:
  net:
```

This is the docker-compose.yml , we are using to make a connection between Jenkins , database and the remote_host , all these three are connected through a network called "net"

6:03 **5. Jenkins & AWS** 33. Create a MySQL server on Docker

Later the stage , we can test the sql container is working as intended or not , by getting into the container using following command.

`docker exec -it db bash`

Then type `mysql` in the container's terminal . then we can see , if mysql is installed or not.

Now , if we wanted to login to the mysql , we need to use the following command

`mysql -u root -p`

And then , we need to give the password , which we given while creating the sql container. And that is `1234` , then if the credentials are correct , mysql will be logged in else check verify your credentials.

If logged in , as a test command use the following command , to see mysql is working as intended or not

`show databases`

Then we get such details in the output,

```
    -> ;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.00 sec)
```