# Practice M5: Jenkins #2

The infrastructure for this practice should look like:

A picture containing chart

Description automatically generated

## Part 1

Make sure that your environment is up

**vagrant up**

Then open [**http://192.168.99.101:8080**](http://192.168.99.101:8080)in a browser

Create the initial administrator user. For example, **admin** with password **Password1**

Install the default set of plugins

Once done, let us configure the remote access between the machines

### Preparation (on the CLI)

Some of the next steps will be familiar to you as we used some of them in the previous module

#### Docker machine

Open a new session

**vagrant ssh docker**

Now, we must create a new **jenkins** user with **Password1** as its password

**sudo useradd jenkins**

**sudo passwd jenkins**

Grant the new user **sudo** permissions

**sudo visudo**

Enter the following

**jenkins ALL=(ALL) NOPASSWD: ALL**

Save and close the file

Add the **jenkins** user to the **docker** group

**sudo usermod -aG docker jenkins**

Switch to the **jenkins** user to test

**su - jenkins**

**docker info**

Exit the **jenkins** user session

**exit**

Now, we can close the session to the **docker** machine

**exit**

#### Jenkins machine

And open a new session to the **jenkins** machine

**vagrant ssh jenkins**

Make sure that the user has password (for example, **Password1**)

**sudo passwd jenkins**

And that the shell is set to **/bin/bash**. For example, by executing

**sudo usermod -s /bin/bash jenkins**

Switch to the **jenkins** user with

**su - jenkins**

Generate a public/private key pair

**ssh-keygen -t ecdsa -b 521 -m PEM**

Copy the **SSH** key to the **jenkins** machine

**ssh-copy-id jenkins@jenkins.do1.lab**

Test the connectivity

**ssh jenkins@jenkins.do1.lab**

Close the test connection

**exit**

Copy the **SSH** key to the **docker** machine

**ssh-copy-id jenkins@docker.do1.lab**

Test the connectivity

**ssh jenkins@docker.do1.lab**

Close the test connection

**exit**

Exit the **jenkins** session

**exit**

Finally, we can close the session to the **jenkins** machine

**exit**

### Preparation (in Jenkins)

#### Credentials

Navigate to **Manage Jenkins** and then to **Manage Credentials**

Then click **System** and then finally **Global credentials**

Let’s add the first set

Click the **Add Credentials** button

Select **Username with password**

Enter **jenkins** for **Username** and the appropriate password in the **Password** field

Set the **Description** to **Local user with password**

Click **Create**

Now, add the second set

Click the **Add Credentials** button

Change the **Kind** to **SSH Username with private key**

In the **Description** box enter **Credentials from file**

Enter **jenkins** for **Username**

Select **Enter directly**

Then click the **Add** button

And paste the contents of the private key file which can be extracted (on the **jenkins** machine) with

**sudo cat /var/lib/jenkins/.ssh/id\_ecdsa**

Finally, click the **Create**

#### Plugins

Return to the initial page of the **Jenkins** portal

Navigate to **Manage Jenkins** and then to **Manage Plugins**

Switch to **Available plugins** and search for **SSH**

Select it and click **Download now and install after restart**

Then select **Restart Jenkins when installation is complete and no jobs are running**

#### Hosts

Once back, navigate to **Manage Jenkins** and then to **Configure System**

Scroll down to the **SSH remote hosts /** **SSH sites** section

Click the **Add** button once to create a record

- hostname: **jenkins.do1.lab**

- port: **22**

- credentials: **jenkins (Credentials from file)** or **jenkins (Local user with password)**

And then once more for an additional record

- hostname: **docker.do1.lab**

- port: **22**

- credentials: **jenkins (Credentials from file)** or **jenkins (Local user with password)**

Check the connectivity for each one of the hosts with the **Check connection** button

Confirm with the **Save** button

#### Slave Host

Now, we can register (or add) the **docker** machine in **Jenkins**

Open the **Jenkins** portal or return to the home page

Navigate to **Manage Jenkins** and then to **Manage Nodes and Clouds**

Click on the **New Node** command

Enter **docker-node** for **Node** name

And select the **Permanent Agent** option

Confirm with **Create**

Next, set the description to **Docker slave node**

And enter the following values in the appropriate places

Set the **# of executors** to **4**

Use **/home/jenkins** for **Remote root directory**

Enter **docker-node** in **Labels**

For **Usage** set **Only build jobs with label expression matching this node**

Set **Launch method** to **Launch agents via SSH**

For **Host** enter **docker.do1.lab**

Now, for Credentials select **jenkins (Credentials from file)**

Leave the **Host Key Verification Strategy** to **Known hosts file**

Confirm with the **Save** button

Next, click on the newly added host

And then on the logs for the host

Monitor the process of agent installation

Only once done, continue with the next steps

### Build Job on a Slave Host (Local)

Being in the **Jenkins** portal, navigate to the **Dashboard**

Click on the **New Item** action

Select **FreeStyle Project** option

Enter **Slave-Local** for name

Confirm with the **OK** button

We can enter **Execute project locally on a slave node** as job description

Select the **Restrict where this project can run** option and set **docker-node** as a label

Add **Execute shell** step in the **Build** section

Enter **hostname && ps ax** in the **Command** text field

Confirm with the **Save** button

Start the execution with the **Build Now** option

Explore the results

Pay attention to the result of the **hostname** command

The job is executed locally but on the **docker** machine as it acts as **Jenkins** slave

### Build Job on a Slave Host (Remote)

*Please note that you should have executed all preparation steps*

We should be ready to define and execute our next job

Navigate to the **Dashboard**

Click on the **New Item** action

Select **FreeStyle Project**

Enter **Slave-Remote** as name

Confirm with **OK**

Enter **Execute project on a slave node against remote machine** as description

Select the **Restrict where this project can run** and set **docker-node** as a label

Scroll-down to the **Build** section

Select the **Execute shell script on remote host using ssh** option in the drop-down list and add it as a step

Enter **jenkins.do1.lab** as host name

Enter **hostname && ps ax** in the **Command** text field

Confirm with the **Save** button

Start the execution with **Build Now**

Check the results

Pay attention to the result of the **hostname** command

The job is executed locally on the **Jenkins** host but is initiated form the **docker** machine as it acts as **Jenkins** slave

So, it is a remote job, considering the **docker** machine

### Build Job on a Slave Host (Docker)

Return to the initial page of the **Jenkins** portal to create our next job that will include **docker**

Navigate to the **Dashboard**

Click on the **New Item** action

Select the **FreeStyle** job type

Enter **Docker-Hello-World** as job name

Enter the following as description **Docker hello world on a slave node**

Check the **Restrict where this project can be run** option and set **docker-node** as label

Scroll-down to the **Build** section

Select the **Execute shell** to add a new build step

Enter **docker container run --rm shekeriev/welcome-do** in the **Command** text field

Confirm with **Save**

Initiate the execution with **Build Now**

Check the execution progress and result

### Build Job on a Slave Host (BGapp)

Click on the **New Item** action

Select the **FreeStyle Project** job type

Enter **Remote-BGApp** for the name

Confirm with **OK**

Enter **Build and run containerized application from GitHub on a slave** as job description

Check the **Restrict where this project can be run** option and set **docker-node** as label

Scroll down to **Build**

Select **Execute shell** to add a new build step

Enter the following two lines in the **Command** text field

**rm -rf bgapp || true**

**git clone https://github.com/shekeriev/bgapp.git**

Select again **Execute shell** to add a new build step

Enter the following two lines in the **Command** text field

**docker network ls | grep appnet || docker network create appnet**

Select again **Execute shell** to add a new build step

Enter the following two lines in the **Command** text field

**cd bgapp**

**docker image build -t img-web -f Dockerfile.web .**

Select again **Execute shell** to add a new build step

Enter the following two lines in the **Command** text field

**docker container rm -f web || true**

**docker container run -d --name web --net appnet -p 8080:80 -v $(pwd)/bgapp/web:/var/www/html:ro img-web**

Select again **Execute shell** to add a new build step

Enter the following two lines in the **Command** text field

**cd bgapp**

**docker image build -t img-db -f Dockerfile.db .**

Select again **Execute shell** to add a new build step

Enter the following two lines in the **Command** text field

**docker container rm -f db || true**

**docker container run -d --name db --net appnet -e MYSQL\_ROOT\_PASSWORD=12345 img-db**

Click the **Save** button to store the changes

Initiate the execution with the **Build Now** action

Monitor the build process

On success, open a browser tab on the host and navigate to [**http://192.168.99.102:8080**](http://192.168.99.102:8080)to see the result

It works 😊

Try to build it once more

Working again 😊

Don’t forget to clean up. Execute the following manually on the **docker** machine

**docker container rm --force web db**

**docker network rm appnet**

## Part 2

### Jenkins CLI

We can issue commands against **Jenkins** in various ways

#### Over SSH

Once inside **Jenkins**, navigate to **Manage Jenkins**

There click on **Configure Global Security**

Scroll down to the **SSH Server** section

Switch the **SSHD Port** to **Fixed** and enter something. For example, enter **2222**

Click on **Save**

Now, switch to the terminal and open session to the **Jenkins** machine

**vagrant ssh jenkins**

Execute the following command to check if it is ready to serve **SSH** connections

**curl -Lv http://192.168.99.101:8080/login 2>&1 | grep -i 'x-ssh-endpoint'**

We should see the port we set earlier

While we are still here, create a new public/private key pair for the **vagrant** user

**ssh-keygen**

Then, execute the following to dump the key

**cat ~/.ssh/id\_rsa.pub**

Copy the text and return to the **Jenkins** UI and navigate to **Manage Jenkins**

There, click on **Manage Users** and click on the **admin** user

Then click **Configure**

Scroll down to the **SSH Public Keys** section

Paste the text copied earlier and click the **Save** button

Return to the terminal session and let’s try if the **CLI** will work

**ssh -l admin -p 2222 localhost help**

If the above works, we will see a list of commands that are available to us

Let’s experiment with a few

Check the version of **Jenkins**

**ssh -l admin -p 2222 localhost version**

Then check a report about our credentials and permissions

**ssh -l admin -p 2222 localhost who-am-i**

Now, return to the UI and create a simple pipeline

Go to the **Dashboard**

Click on **New Item**

Then, enter **Simple-Pipeline** in the name field

Select **Pipeline** and click **OK**

Scroll down to the **Pipeline** section

Select **Hello World** from the drop-down list to use the sample code

Click **Save**

Then, return on the console and try the following to list the jobs

**ssh -l admin -p 2222 localhost list-jobs**

Okay, we can see it

Now, let’s build the job

**ssh -l admin -p 2222 localhost build Simple-Pipeline**

Hm, we did not see what we would expect

Let’s return to the UI and check if the job executed

Yes, everything is fine there

Now, return to the console and ask for help for the build command

**ssh -l admin -p 2222 localhost help build**

Hm, we can use the **-f** and **-v** options to follow the job execution. Let’s do it

**ssh -l admin -p 2222 localhost build Simple-Pipeline -f -v**

Aha, this is another story 😉

Now, what is we want to see the console output of a build? Of course, we can do it

Try this command

**ssh -l admin -p 2222 localhost console Simple-Pipeline**

*Please note that depending on your setup, the output may include garbage information (such as part of the key)*

Hit **Ctrl+C** to get the control back

Now, let’s try a parametrized job

First, create a copy of the existing one but on the terminal by executing

**ssh -l admin -p 2222 localhost copy-job Simple-Pipeline Param-Pipeline**

Now, return to the UI to adjust the copy

Select the new job and click **Configure**

Tick the option **This project is parametrized**

Then click the **Add Parameter** drop-down list and select **String Parameter**

Set **PAR\_NAME** for the **Name**

And then enter **World** in the **Default Value** field

Next, scroll to the **Pipeline** section

Change the line that prints the string to match this one

**echo "Hello ${PAR\_NAME}"**

Then click on the **Save** button to store the changes

If you like, test the job first in the UI. It should work

Return on the console and execute the job as if it doesn’t have parameters

**ssh -l admin -p 2222 localhost build Param-Pipeline -f -v**

It will print **Hello World** because the default value for the parameter is **World**

Then adjust the command to add a parameter as well

**ssh -l admin -p 2222 localhost build Param-Pipeline -f -v -p PAR\_NAME='Brave\ Man'**

The above will result in printing **Hello Brave Man** text

We must pay attention and handle spaces with care

#### With CLI client

There is another way. We can use the **CLI** client to send commands to Jenkins

Return to the UI

Navigate to **Manage Jenkins** and then to **Jenkins CLI**

Here, we can see that we must download the **jenkins-cli.jar** file in order to use this way of controlling **Jenkins**

We can also see a list of the available commands

And even click on the individual commands to see what they do and what options do they have

Return to the console and execute the following to download the **CLI**

**wget http://192.168.99.101:8080/jnlpJars/jenkins-cli.jar**

Now, we have two modes for communication to Jenkins – via **HTTP** and **SSH**

Let’s test first the **SSH** based mode

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin version**

In the same manner, we can list the jobs

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin list-jobs**

Or execute one of them

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin build Param-Pipeline -f -v -p PAR\_NAME='From CLI'**

Now, let’s test the other connection mode

Fist, we must prepare an **API token**

We can return to the UI and then go to **Manage Jenkins** > **Manage Users** > **admin** > **Configure**

Instead, we can use this URL in a browser tab [**http://192.168.99.101:8080/me/configure**](http://192.168.99.101:8080/me/configure)

Use whichever way you like

Once there, scroll to the **API Token** section

There, click the **Add new Token** button

For name enter for example, **cli-token** and then click the **Generate** button

Copy the token and store it in a safe place

Finally, click **Save** and return to the console

Ask again for the version but with this command

**java -jar jenkins-cli.jar -s http://localhost:8080/ -http -auth admin:<token-code> version**

Then list the jobs

**java -jar jenkins-cli.jar -s http://localhost:8080/ -http -auth admin:<token-code> list-jobs**

And finally, execute one of them

**java -jar jenkins-cli.jar -s http://localhost:8080/ -http -auth admin:<token-code> build Param-Pipeline -f -v -p PAR\_NAME='From HTTP'**

### Export / Import

There are multiple ways to export and import jobs in **Jenkins**

These include all variants of the **CLI**, various plugins, manual copy, etc.

We will use the **CLI** client and manual copy and compress/restore process

#### Using the CLI

First, navigate to the terminal

We will continue with the **SSH mode** of using the client

Then, execute the following to get information about the command that we want to use

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin help get-job**

As we can see, there is nothing so special about this command

So, let’s execute it and dump one of the jobs

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin get-job Simple-Pipeline**

Aha, so it appears on the **stdout**. Let’s redirect it to a file

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin get-job Simple-Pipeline > job1.xml**

Check the file contents

**cat job1.xml**

It seems to be just fine

Now, let’s be brave and delete the **Simple-Pipeline** job from **Jenkins**

We can use the UI, or we can do it with this command

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin delete-job Simple-Pipeline**

If we ask for the list of available jobs, we should not see it anymore there

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin list-jobs**

Even if we check in the UI, we should not see it there

Now, let’s recreate the job

To import it, we can use the following command

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin create-job Imported-Pipeline < job1.xml**

Check the list

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin list-jobs**

It is there. So, let’s build it with

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin build Imported-Pipeline -f -v**

It works 😊

#### Using copy

Okay, let’s try one more way of exporting and importing jobs

List the contents of the folder where **Jenkins** keeps the job-related files

**ls -al /var/lib/jenkins/jobs**

Aha, so our jobs are presented here as folders

Let’s check the contents of the **Param-Pipeline** folder

**ls -al /var/lib/jenkins/jobs/Param-Pipeline/**

So, here are the files

We can check the next build number for the job

**cat /var/lib/jenkins/jobs/Param-Pipeline/nextBuildNumber**

And the actual configuration of the job

**cat /var/lib/jenkins/jobs/Param-Pipeline/config.xml**

In fact, it is enough to create a copy of the **config.xml** file

Let’s create a new folder

**sudo mkdir -m 755 /var/lib/jenkins/jobs/Pipeline-Copy**

And create a copy of the **config.xml** file of the **Param-Pipeline** job

**sudo cp /var/lib/jenkins/jobs/Param-Pipeline/config.xml /var/lib/jenkins/jobs/Pipeline-Copy/**

And finally, adjust the ownership

**sudo chown -R jenkins:jenkins /var/lib/jenkins/jobs/Pipeline-Copy/**

Okay, let’s check if we can see the job

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin list-jobs**

No, we cannot. The same applies if we check in the UI

So, perhaps we must reload the configuration with

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin reload-configuration**

And then if we try again to list the jobs

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin list-jobs**

We should see our copied job

Let’s try to start it

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin build Pipeline-Copy -f -v -p PAR\_NAME='Copied\ Job'**

It works 😊

#### Using compression

Let’s try one more

Go to the pipeline folder

**cd /var/lib/jenkins/jobs/Param-Pipeline/**

And execute the following to create the backup

**tar czvf ~/bak.tgz --transform 's,,backup-pipeline/,' config.xml**

Then, return to **vagrant** user’s home folder

**cd**

And execute the following to test the backup

**tar tzvf bak.tgz**

And then to import the backup

**sudo tar xzvf bak.tgz -C /var/lib/jenkins/jobs/**

Now check the result

**ls -al /var/lib/jenkins/jobs/backup-pipeline/**

And adjust the ownership

**sudo chown jenkins:jenkins /var/lib/jenkins/jobs/backup-pipeline/**

Now, reload the configuration

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin reload-configuration**

And list the jobs

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin list-jobs**

Finally, try to execute the restored job

**java -jar jenkins-cli.jar -s http://localhost:8080/ -ssh -user admin build backup-pipeline -f -v -p PAR\_NAME='tar+gz'**

It works 😊

### Blue Ocean

Navigate to the **Jenkins** UI

Go to **Manage Jenkins** > **Manage Plugins**

Then, in the **Plugin Manager**, switch to **Available plugins**

Enter **Blue Ocean** in the **Search** bar and hit **Enter**

Select the **Blue Ocean** (usually the first option) and click **Download now and install after restart**

Then you may select the **Restart Jenkins when installation is complete and no jobs are running** option

Please note that is a lengthy process and may take some time

Once the system is ready, log in back again

This time, click on **Open Blue Ocean** item in the menu on the left

This will open a different interface compared to the one we used to work with

We can click on one of the existing pipelines and explore them

We can even run one of them

Run for example, the **Param-Pipeline**

It works 😊

Of course, we can edit existing pipelines, but this will return us to the classic interface

Go and do it, change the print text of the **Imported-Pipeline**

Save the changes and return to the **Blue Ocean**

Execute it and check the results

It works 😊

#### Multibranch Pipeline

##### Preparation

Open a terminal session to the **docker** machine

**vagrant ssh docker**

Execute the following to copy locally the **docker-compose.yml** file

**cp /vagrant/docker-compose.yml .**

Explore it and then start it

**docker compose up -d**

Now, return on the host and open a browser tab and navigate to [**http://192.168.99.102:3000**](http://192.168.99.102:3000)

Change at least the following two options:

* Set **Server Domain** to **192.168.99.102**
* Set **Gitea Base URL** to **http://192.168.99.102:3000/**

Accept the rest of the proposed values and click the **Install Gitea** button

Once, the page is refreshed note that initially it may show an error. Refresh it a few times

Click the **Need an account? Register now.** link to create an account

Enter the required data and click the **Register Account** button

Once in, create an empty repository

Click the **+** button (top right corner) and select **New Repository**

Enter **test** in the **Repository Name**

Click the **Create Repository** button

*Let’s execute the instructions to create a local repository and push it to the one in* ***Gitea***

*Note that the above can be done on either of the two machines. Let’s do it on the* ***jenkins*** *one*

Log on to the **jenkins** machine

Create a folder named **test**

**mkdir test**

And enter it

**cd test**

Then execute the proposed steps

**touch README.md**

**git init**

**git add README.md**

**git commit -m "first commit"**

**git remote add origin http://192.168.99.102:3000/<username>/test**

**git push -u origin master**

Return to **Gitea** and check that the changes are reflected

##### The Pipeline

Return to **Jenkins** and make sure you are in **Blue Ocean**

Click the **New Pipeline** button

Select **Git**

Enter the path to the repository in the **Repository URL** field

It should be something like **http://192.168.99.102:3000/<username>/test**

Then, enter the username and password that you used for **Gitea**

Finally, click the **Create Credential** button to store the credentials in **Jenkins**

And then click the **Create Pipeline** button

You will be presented with a dialog that states that there are not any **Jenkinsfiles**

Click the **Create Pipeline** button that appears

Click the first **+** to create a new step which will be our first stage

Enter a name, for example **Stage 1**

Click the **+ Add step** button

Select the **Print Message** function

Enter **Hello World** as **Message**

Click the back arrow

Click the **+ Add step** button

Select the **Shell Script** function

Enter **echo "Running on $(hostname)"**

Click the back arrow

Click the **+** on the right

Enter **Stage 2** as name

Click the **+ Add step** button

Select the **Print Message** function

Enter **Sleep for 30 sec** as **Message**

Click the back arrow

Click the **+ Add step** button

Select the **Sleep** function

Enter **30** in **Time**

Click the back arrow

Click the **+** on the right

Enter **Stage 3** as name

Click the **+ Add step** button

Select the **Print Message** function

Enter **Done** as **Message**

Click the back arrow

Finally, click the **Save** button

Enter **Initial commit** in the **Description** field

Click **Save & run**

Then click the **Show branches** button

Click on the only row to monitor the execution

Once done, click on the stages to explore what happened

Let’s add one parallel flow

Click on the pencil icon (top left)

Click on the **+** bellow the **Stage 2**

Enter **Stage 2.2** as name

Click the **+ Add step** button

Select the **Shell Script** function

Enter **echo "Process running: $(ps ax | wc -l)"**

Click the back arrow

Select the **Stage 2** and rename it to **Stage 2.1**

Finally, click the **Save** button

Enter **Parallel flow added** as **Description**

Click **Save & run**

Then click the new run and monitor the process

Once done, click on the stages to explore what happened

Return to **Gitea** and explore the artefacts and commits

## Part 3

We are ready to test a few more techniques before we create one bigger project

### Pipeline SCM

Navigate to **Gitea**

Create a new repository and name it **scm-pipeline**

*Execute the instructions to create a local repository and push it to the one in* ***Gitea***

*Note that the above can be done on either of the two machines. Let’s do it on the* ***jenkins*** *one*

Log on to the **jenkins** machine

Create a folder named **scm-pipeline** and enter it

Create a **Dockerfile** with the following content

**FROM php:8.0-apache**

**RUN echo '<?php print "Hello World"; ?>' > /var/www/html/index.php**

Create a **Jenkinsfile** with the following content

pipeline

{

    agent

{

        label 'docker-node'

    }

    stages

    {

        stage('Clone')

        {

            steps

            {

                git 'http://192.168.99.102:3000/<username>/scm-pipeline'

            }

        }

        stage('Build')

        {

            steps

            {

                sh 'docker image build -t img-app .'

            }

        }

        stage('Run')

        {

            steps

            {

                sh '''

                docker container rm -f co-app || true

                docker container run -d -p 9090:80 --name co-app img-app

                '''

            }

        }

        stage('Clean')

        {

            steps

            {

                cleanWs()

            }

        }

    }

}

Finally, add the **README.md** file

**echo 'Hello World Web App' > README.md**

And initialize the repository and commit and push all the files

**git init**

**git add .**

**git commit -m "first commit"**

**git remote add origin http://192.168.99.102:3000/<username>/scm-pipeline.git**

**git push -u origin master**

Return to **Gitea** and explore the artefacts

Now, return to **Jenkins** but go to the classic UI

Navigate to the **Dashboard**

Click **New Item**

Select **Pipeline**, type **SCM-Pipeline** for name and confirm with **OK**

Turn on the **GitHub project** option and paste the URL of the project

[http://192.168.99.102:3000/<username>/scm-pipeline](http://192.168.99.102:3000/%3cusername%3e/scm-pipeline)

Scroll down to the **Pipeline** section and select **Pipeline script from SCM**

Set **Git** as **SCM**

Paste [http://192.168.99.102:3000/<username>/scm-pipeline](http://192.168.99.102:3000/%3cusername%3e/scm-pipeline) as **Repository URL**

Click the **Save** button

Click **Build Now** and watch the progress

Once done, open a browser tab and navigate to <http://192.168.99.102:9090>

The app should be there up and working 😉

Try to build it once more

Again, it should work

If you wonder how we came up with some of the commands in the pipeline, then return to its configuration page

Scroll down and click the **Pipeline Syntax** link

A new window will open and here we can try different functions and see their syntax

Once done, close it

### Web Hooks

Return to the **Dashboard** of **Jenkins**

Navigate to **Manage Jenkins** > **Manage Plugins**

Switch to **Available plugins** and search for **gitea**

Select it (usually, the first one is **Gitea**) and click **Download now and install after restart**

Then you may select the **Restart Jenkins when installation is complete and no jobs are running** option

Once the system is ready, log in back again

Return to the **Dashboard** and click on the **SCM-Pipeline** job

Then click **Configure**

Scroll down to **Build Triggers** section

Select both **GitHub hook trigger for GitScm polling** and **Poll SCM**

Click **Save** to confirm the changes

Now, return to **Gitea**

Go to the corresponding repository

Click **Settings** (far right corner)

Switch to **Webhooks**

Click **Add Webhook** and select **Gitea**

Enter <http://192.168.99.101:8080/gitea-webhook/post> in the **Target URL** field

Then click the **Add Webhook** button

Click back on the hook to enter in edit mode

Scroll down and click the **Test Delivery** button

Check the results – explore both tabs – **Request** and **Response**

*If there is an error, check the URL on the first place. Then check the configuration for* ***Gitea*** *and see if there is an environment variable present and configured to allow other hosts to invoke webhooks (in the* ***docker-compose.yml*** *file the environment variable is* ***GITEA\_\_webhook\_\_ALLOWED\_HOST\_LIST****). It should look like:*

***GITEA\_\_webhook\_\_ALLOWED\_HOST\_LIST=192.168.99.0/24***

*If you make any changes, do not forget to restart the container*

***docker compose down***

***docker compose up -d***

Return to the repository

Now, go to the machine where it was initialized and change for example the **Dockerfile** *(change the string)*

Then commit and push the changes

**git add Dockerfile**

**git commit -m 'Dockerfile changed'**

**git push -u origin master**

Now return to **Jenkins** and watch the build progress

Once done, check the result by visiting <http://192.168.99.102:9090>

Then, check the changes in the **Gitea** repository

Feel free to experiment a bit

### Complete Project (Round 1)

Let’s do something more complex and complete

Navigate to **Gitea**

Click on the **+** sign next to the account menu (top right)

Select **New Repository**

Click **New Migration**

Click on **GitHub**

Enter **https://github.com/shekeriev/supercalc** in the **Migrate / Clone From URL** field

Fill **supercalc** in the **Repository Name** field

Click **Migrate Repository**

Once done, explore the structure and the code

Return to the **Dashboard** in **Jenkins**

Click **New Item**

Then select **Pipeline** and enter **Super-Calc** for name and click **OK**

Select **GitHub project** and paste the repository URL

For example, **http://192.168.99.102:3000/<username>/supercalc**

Scroll down to the **Pipeline** section and paste the following code

pipeline

{

    agent

{

        label 'docker-node'

    }

    stages

    {

        stage('Clone')

        {

            steps

            {

                git branch: 'main', url: 'http://192.168.99.102:3000/<username>/supercalc'

            }

        }

        stage('Build')

        {

            steps

            {

                sh 'docker image build -t img-calc .'

            }

        }

        stage('Run')

        {

            steps

            {

                sh 'docker container rm -f co-calc || true'

                sh 'docker container run -d -p 8080:80 --name co-calc img-calc'

            }

        }

        stage('Clean')

        {

            steps

            {

                cleanWs()

            }

        }

    }

}

Note the differences in the **Clone** and **Build** stages compared to our previous attempts

Click the **Save** button

Run a test build with **Build Now**

Once the build is done, navigate to <http://192.168.99.102:8080> and try a few calculations

Note that the subtraction is not working correctly. Do not worry, we will fix it later

### Complete Project (Round 2)

Let’s extend what we did and bring it to another level

Let’s add a **Test** stage and move the clean up as post step so it will execute always

Open the **Configuration** mode of the **Super-Calc** job and change the pipeline code to this

pipeline

{

    agent

{

        label 'docker-node'

    }

    stages

    {

        stage('Clone')

        {

            steps

            {

                git branch: 'main', url: 'http://192.168.99.102:3000/<username>/supercalc'

            }

        }

        stage('Build')

        {

            steps

            {

                sh 'docker image build -t img-calc .'

            }

        }

        stage('Run')

        {

            steps

            {

                sh 'docker container rm -f co-calc || true'

                sh 'docker container run -d -p 8080:80 --name co-calc img-calc'

            }

        }

        stage('Test')

        {

            steps

            {

                script

                {

                    echo 'Test #1 - reachability'

                    sh 'echo $(curl --write-out "%{http\_code}" --silent --output /dev/null http://localhost:8080) | grep 200'

                }

            }

        }

    }

    post

    {

        always

        {

            cleanWs()

        }

    }

}

Click the **Save** button

Run a test build with **Build Now**

**It should pass**

Now, change the URL in the test to something nonexistent. For example, to **http://localhost:8080/ttt**

Save and build again. **This time it will fail**

Return in **Configuration** mode and change the test stage to match this

        stage('Test')

        {

            steps

            {

                script

                {

                    echo 'Test #1 - reachability'

                    sh 'echo $(curl --write-out "%{http\_code}" --silent --output /dev/null http://localhost:8080) | grep 200'

                    echo 'Test #2 - 40 + 2 = 42'

                    sh "curl --silent --data 'opa=40&opr=add&opb=2' http://localhost:8080 | grep 42"

                }

            }

        }

Click the **Save** button

Run a test build with **Build Now**

**It should pass**

### Complete Project (Round 3)

Let’s extend what we did and bring it to even higher level

Now, let’s publish the image to **Docker Hub** if the test part succeeds

Log in to **Docker Hub**

Then go to **Account Settings** and then **Security**

Click the **New Access Token** button

Enter **Jenkins** in the **Access Token Description**

And click **Generate**

Click the **Copy and Close** button and store the token somewhere safe

Return to **Jenkins**

Navigate to **Manage Jenkins** > **Manage Credentials**

Then click on **System** > **Global credentials**

Finally, click the **Add Credentials** link

Enter your **Docker Hub** username in the **Username** field

Then paste the token in the **Password** field

Enter **docker-hub** in the **ID** and **Description** field

Finally, click **Create**

Return in **Configure** mode of the pipeline (**Super-Calc**)

Add the following just after the **agent** section

    environment

    {

        DOCKERHUB\_CREDENTIALS=credentials('docker-hub')

    }

And then, the following two stages after the **Test** stage

        stage('Login')

        {

            steps

            {

                sh 'echo $DOCKERHUB\_CREDENTIALS\_PSW | docker login -u $DOCKERHUB\_CREDENTIALS\_USR --password-stdin'

            }

        }

        stage('Push')

        {

            steps

            {

                sh 'docker image tag img-calc <username>/supercalc'

                sh 'docker push <username>/supercalc'

            }

        }

Click the **Save** button

Run a test build with **Build Now**

**It should pass**

And furthermore, our image is published to the public registry

### Complete Project (Round 4)

We can add a few more tweaks

For example, removal of the test container, one more test for subtraction (which initially will fail), webhook and deployment of the application out of the published image

#### Webhook

Let’s add the webhook first

Go to **Gitea**

Enter the **Settings** of the repository

Switch to **Webhooks**

Click **Add Webhook** and select **Gitea**

Enter **http://192.168.99.101:8080/gitea-webhook/post** in the **Target URL**

Confirm with the **Add Webhook** button

Return to the **Jenkins** UI

Enter **Configure** mode of the pipeline

Turn on both options – **GitHub hook trigger for GITScm polling** and **Poll SCM**

Click **Apply** to save the changes

#### Additional Test

Go to the **Pipeline** section

Add the following to the tests

                    echo 'Test #3 - 150 - 108 = 42'

                    sh "curl --silent --data 'opa=150&opr=sub&opb=108' http://localhost:8080 | grep 42"

Click **Apply** to save the changes

#### Test Container Removal

Add one more stage just after the **Test** stage

        stage('CleanUp')

        {

            steps

            {

                sh 'docker container rm -f co-calc || true'

            }

        }

Click **Save** to save the changes

Then click **Build Now** to test the whole thing

**One of the tests failed** (it should be the last one that we added)

#### Final Deployment

Let’s add one more step before we correct the bug that led to our test #3 not passing

Enter **Configure** mode of the pipeline

Go to the **Pipeline** section

Add the following as a last stage (after the **Push** stage)

        stage('Deploy')

        {

            steps

            {

                sh 'docker container rm -f calcapp || true'

                sh 'docker container run -d -p 80:80 --name calcapp <username>/supercalc'

            }

        }

Of course, normally this will be on another host, and/or with other port mapping, etc.

In our case, we will change just the port to a “production” value of **80** 😉

Click **Save** to save the changes

#### Correct the Code

We can do this directly in **Gitea**

Open the **app/index.php** file for editing

Change the code on **row 28** to match this

    if ($opr == 'sub') { print "$opa - $opb = "; print intval($opa)-intval($opb); }

Confirm with **Commit Changes** button

Return to **Jenkins** UI and watch the process

**It should succeed**. Enjoy 😊

Check the log and then check the app on [**http://192.168.99.102**](http://192.168.99.102)