

# Configuration Manual

MSc Research Project  
Cloud Computing

Conor Deegan  
Student ID: x15023257

School of Computing  
National College of Ireland

Supervisor: Vikas Sahni

National College of Ireland  
Project Submission Sheet  
School of Computing



<b>Student Name:</b>	Conor Deegan
<b>Student ID:</b>	x15023257
<b>Programme:</b>	Cloud Computing
<b>Year:</b>	2019
<b>Module:</b>	MSc Research Project
<b>Supervisor:</b>	Vikas Sahni
<b>Submission Due Date:</b>	12/8/2019
<b>Project Title:</b>	Configuration Manual
<b>Word Count:</b>	XXX
<b>Page Count:</b>	25

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

**ALL** internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

<b>Signature:</b>	
<b>Date:</b>	24th September 2020

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:**

Attach a completed copy of this sheet to each project (including multiple copies).	<input type="checkbox"/>
<b>Attach a Moodle submission receipt of the online project submission</b> , to each project (including multiple copies).	<input type="checkbox"/>
<b>You must ensure that you retain a HARD COPY of the project</b> , both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

<b>Office Use Only</b>	
Signature:	
Date:	
Penalty Applied (if applicable):	

# Configuration Manual

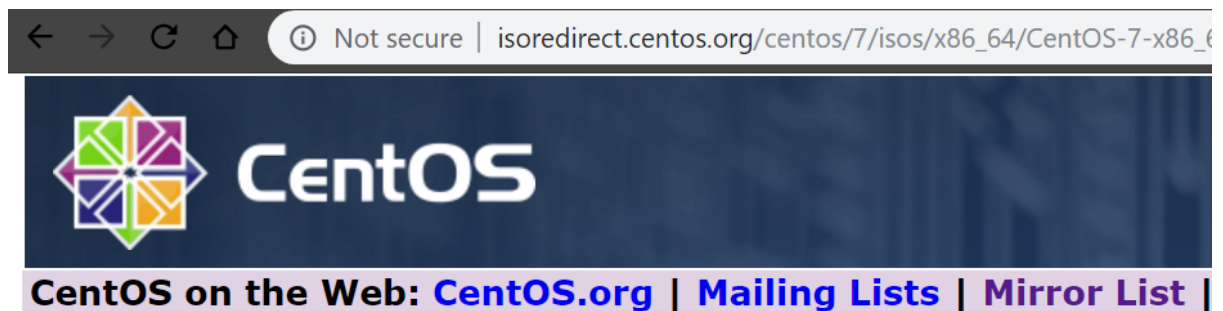
Conor Deegan  
x15023257

## 1 Setting up the environment

The environment for this project was initially set up and deployed on a Virtual Machine running on my local machine. The machine is a Lenovo Ideapad 530S with Windows 10 installed. The specs can be found in Table 1 below.

Model	Lenovo Ideapad 530s
Processor	Intel Core i5 1.6Ghz
Memory/RAM	8GB
Hard Drive	256 SSD
Operating System	Windows 10(64-bit)

Oracle Virtual Box was used as the VMM software. Version 5.2.30 was downloaded from <https://www.virtualbox.org/wiki/Downloads>. For the first phase of this project as single VM was configured with CentOS 7 installed. The virtual machine image for the machine was downloaded from [centos.org](http://centos.org). The minimal ISO was used in order to minimise the footprint of the VM and because this laboratory only requires the command line interface in order to achieve it's goals. It is important to distinguish between 32-bit and 64-bit Operating systems in order to ensure compatibility of the virtual machine image with your host machine. In the case of the host machine used in this laboratory configuration, the 64-bit virtual box image was chose as shown in the figure below.

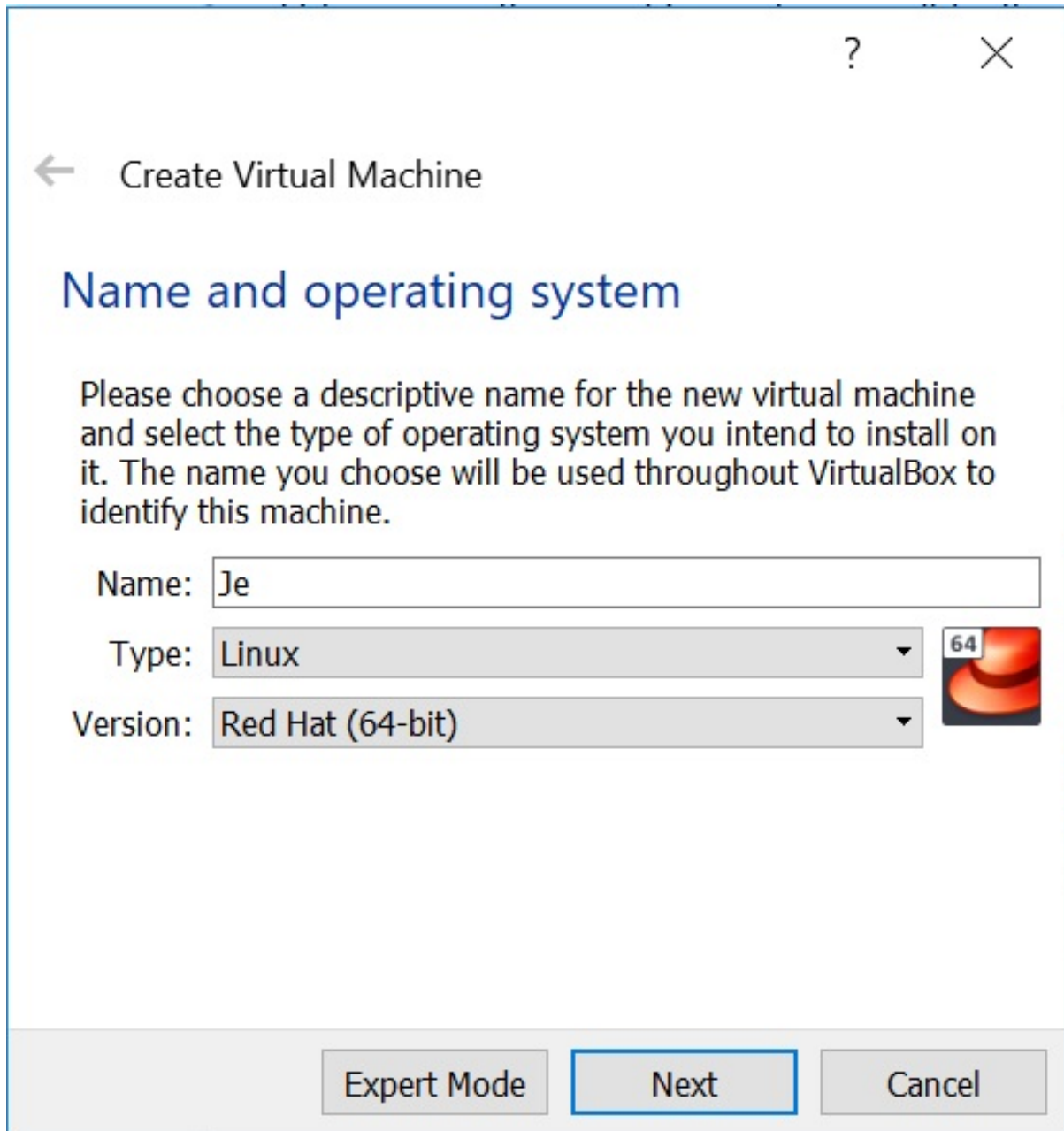


In order to conserve the limited bandwidth available, ISO images are not downloadable from mirrors.

The following mirrors in your region should have the ISO images available:

[http://ftp.heanet.ie/pub/centos/7.6.1810/isos/x86\\_64/CentOS-7-x86\\_64-Minimal-1810.iso](http://ftp.heanet.ie/pub/centos/7.6.1810/isos/x86_64/CentOS-7-x86_64-Minimal-1810.iso)  
[http://mirror.strencom.net/centos/7.6.1810/isos/x86\\_64/CentOS-7-x86\\_64-Minimal-1810.iso](http://mirror.strencom.net/centos/7.6.1810/isos/x86_64/CentOS-7-x86_64-Minimal-1810.iso)

Once both Virtual Box and the appropriate VM image are downloaded, Virtual Box is installed by using the launch wizard with all of the standard configurations. A note is to be taken of the location of the downloaded virtual image. The next step is to Create a new Virtual Machine within VirtualBox. For the purpose of this laboratory, the name Jenkins was chosen and the Linux OS was selected. As CentOS does not appear as an option in the list of distributions, RedHat(64-bit) was chosen to ensure compatibility with our VM image. This can be seen in the image below.



← Create Virtual Machine

## Name and operating system

Please choose a descriptive name for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

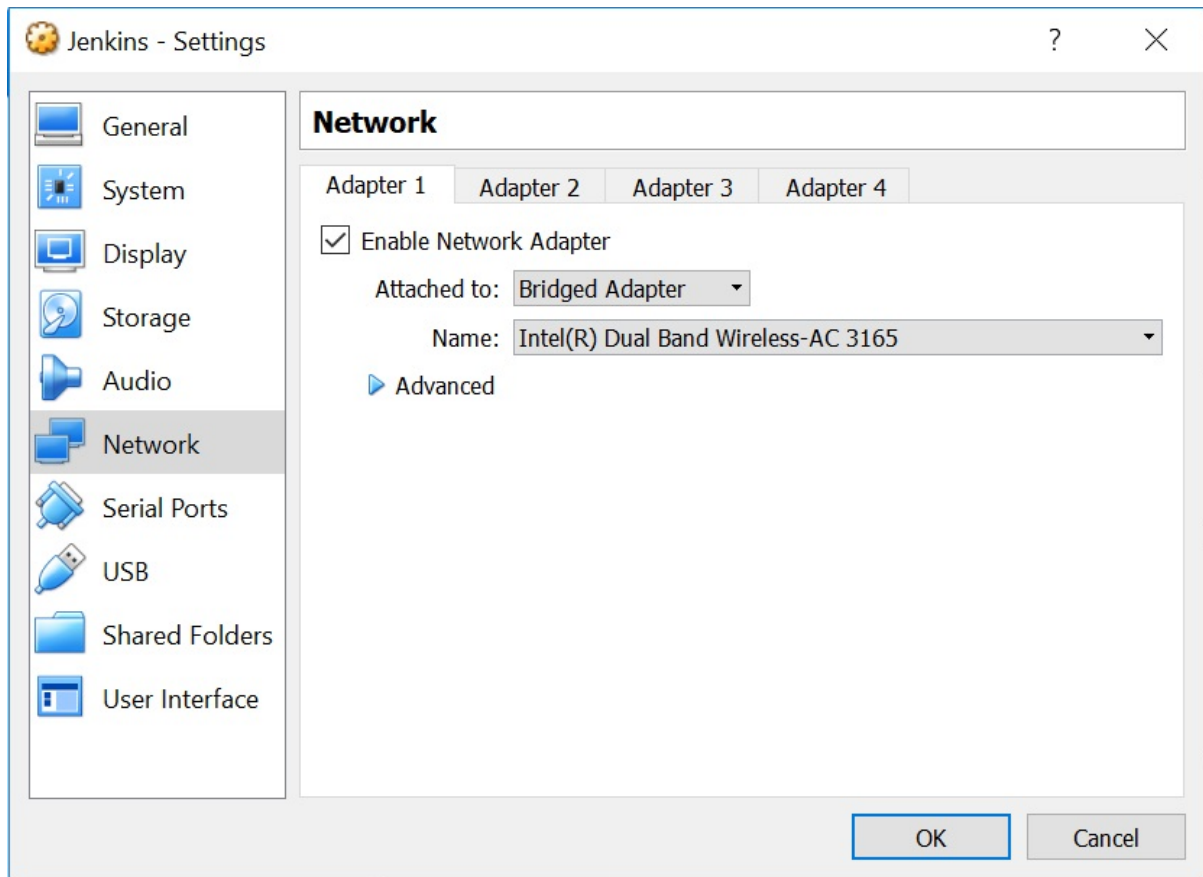
Name:

Type:

Version:

The next screen prompts the user to allocate RAM for the VM. At this stage 2GB was selected. Note that this can be adjusted later if needed. The next stage is creating a virtual hard disk to attach to the VM. There is an option to use an existing vdi(virtual disk image) or create a new one. For this lab it is recommended to specify 20GB and allocate this dynamically. Now that the specifications of the VM is complete it is time to create a network interface for the VM. This will vary depending on the type of connection



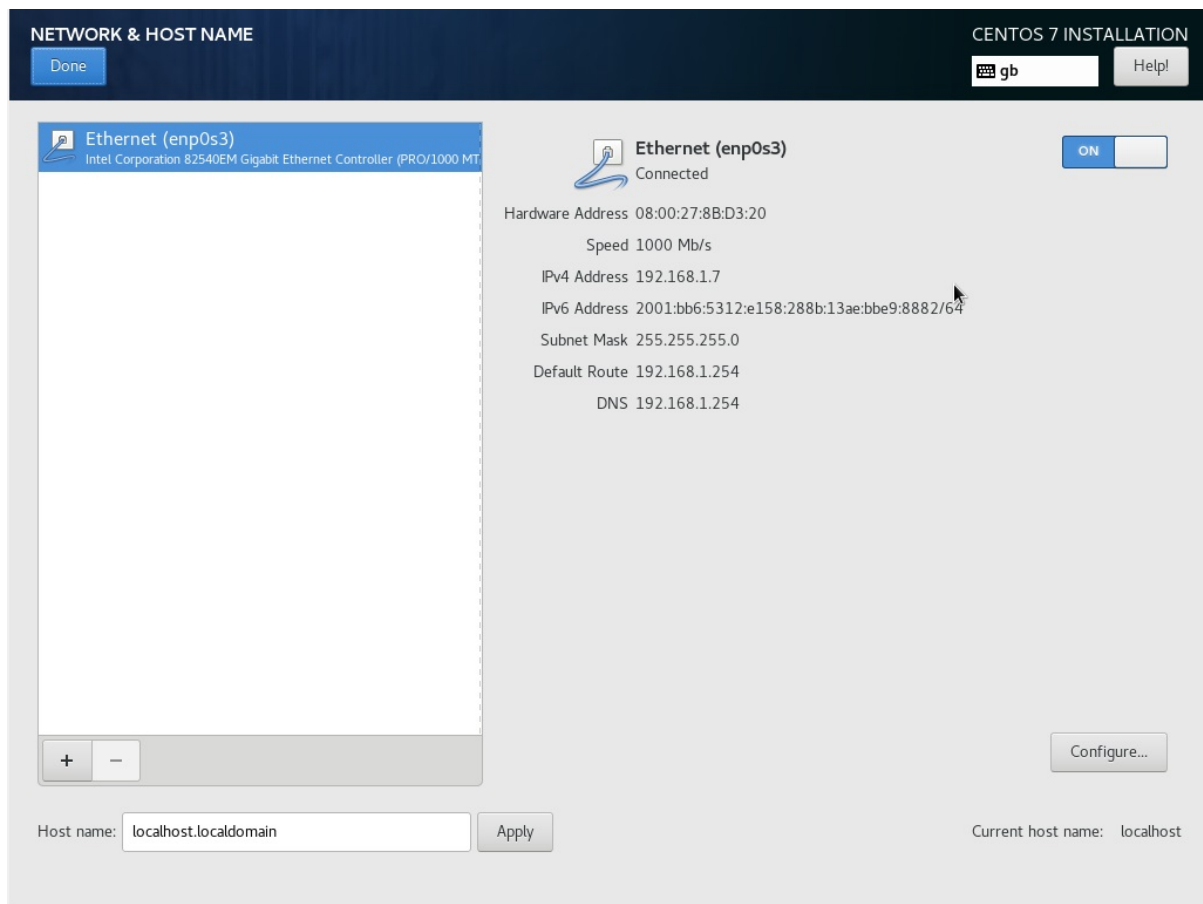


by ticking this box. This adds the user to the 'sudo' group and allows the account the sudo privileges which will be required to complete the lab set up. After this click on 'Finished configurations' and wait for the installation process to complete. The user will be prompted to reboot the machine and then the machine will load. Sign in using the user credentials which were just created. Now we can SSH into the machine using Putty.

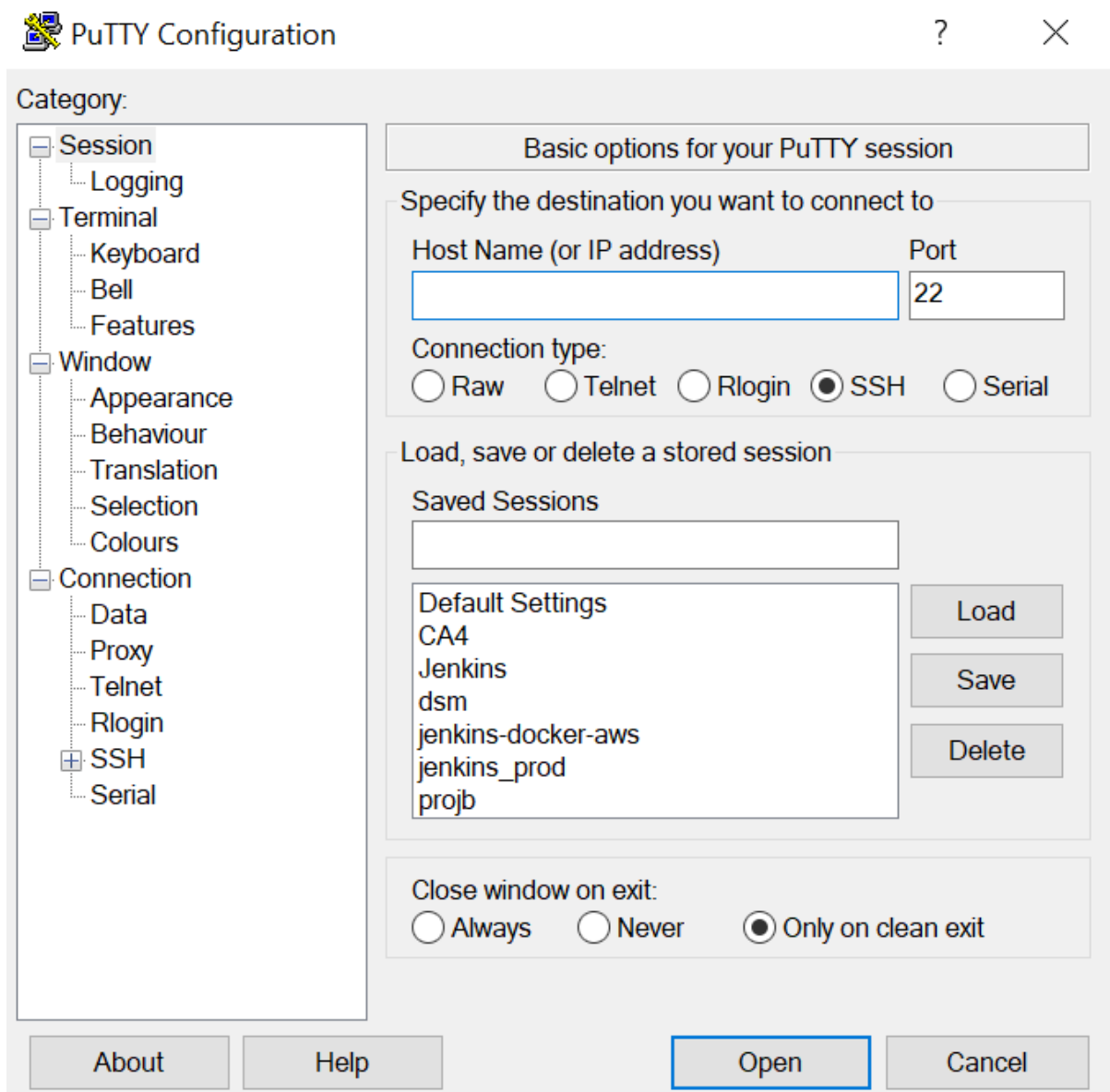
Firstly the user will need to determine the IP address of the newly created. This is done in linux by typing 'ip addr'. Take note of this and proceed to download Putty for Windows 64-bit. When this is downloaded, install it using the provided launch wizard and install it. Run the Putty program and you will be prompted with the UI shown in the image below. Please enter the IP address under Host Name and save the session for convenience. click 'Yes' to allow the connection and enter the user credentials for the VM that were created earlier. This concludes the first section. A Virtual Machine running CentOS has been initialised using VirtualBox and a user with admin privileges has been created. A connection to the virtual machine has been achieved using the SSH client Putty.

## 2 Setting up Docker and Jenkins

Now it is time to install Docker on the virtual machine. Docker and 'docker-compose' will be used to create and manage the various containers and the network between them. Installing docker differs depending on the distribution. In CentOS the following steps were found in the Docker documentation <https://docs.docker.com/install/linux/docker-ce/centos/>;



- Ensure that you are logged in as the user created earlier
- Ensure the yum utils and yum config utilities are installed :`sudo yum install -y yum-utils device-mapper-persistent-data lvm2`
- Use the next command to set up the repository: `sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo` NOTE: when downloading the repo, I had an error where the download kept timing out. This was resolved by first ensuring the time and date was correct in the machine using the `timedatectl` command. If this does not fix it try running a `sudo yum update -y`.
- Use the following command to install Docker using the stable repo previously downloaded; `sudo yum install docker-ce docker-ce-cli containerd.io`
- After this, use the following command to start the Docker service; `sudo systemctl start docker`
- Next the user can enable the docker service at boot using `sudo systemctl enable docker`
- Then we must add the user we created to the docker group in order to run docker commands from this account using the following command: `sudo usermod -aG docker username(replace with your username)`
- log out of the putty session and log back in to initialise this change.



Once docker is installed, its time to install docker-compose. Docker-compose makes the process of Installing and configuring our network of containers much easier than using the traditional docker build process. For the scale of this project, docker-compose is perfect. However, in larger scale, industry environments, tools like Docker Swarm would be better suited. As this laboratory is a simulation, docker-compose is perfect for what is needed.

- First we will download the current version of docker-compose using : `sudo curl -L "https://github.com/docker/compose/releases/download/1.24.0/docker-compose-(uname -s)-(uname -m)" -o /usr/local/bin/docker-compose`
- Then we give execution permissions to docker-compose using : `sudo chmod +x /usr/local/bin/docker-compose`
- Finally test it is working by entering : `docker-compose` and if it returns a list of possible commands, the installation has been successful.



The next section will cover the configuration of the Jenkins container, which will act as the CI server for the lab.

### 3 Configuring Jenkins in a Docker Container

This lab will utilize Docker to recreate a Continuous Integration/Delivery Pipeline at a smaller scale than a typical industry installation. Jenkins will be installed in a container and the docker-compose tool will be used to configure the containers that will interact with Jenkins in order to simulate a full CI/CD pipeline. The next stage of this lab configuration involves setting up Jenkins in a Docker container using docker-compose. Make sure you are signed in as the admin user created earlier and in the home directory for this user. OPTIONAL: This section will involve editing files within the VM. If you have a preferred text editor for Linux, now is the time to install it. I prefer to use the vim editor which I installed using the command: `sudo yum install vim`

- First find the official Jenkins image on Docker Hub: <https://hub.docker.com/r/jenkins/jenkins/>
- return to the command line of the VM and type in: `docker pull jenkins/jenkins` and wait for the image to download.
- Ensure the image was successfully downloaded by running: `docker images`. This should display the details of the new image.
- Next create a new directory using the command `mkdir jenkins-data`. The name of the folder is not significant.
- Within this folder we will create a YAML file which will define the configuration of our containers. The following command was used `vim docker-compose.yml` NOTE: The name of this file is significant. A rough guide to defining a Jenkins container using docker-compose can be found here: <https://docs.docker.com/compose/compose-file/>
- add the configurations shown in the image below.
- Press 'ESC' button and the type `:wq` to save the file.
- make a new folder to store the volume to persist the data in the jenkins container using `mkdir jenkins_home`. This directory will contain all of the files generated by our jenkins container as the lab progresses including the workspaces for the jenkins jobs that will be created.

The following image shows the initial configurations defined to get the Jenkins container up and running. Firstly the version of Docker being used is defined. Then under the services heading the name of the container is listed. This is useful in distinguishing the containers as the number of them grows. Then the image which is to be installed in the container is defined. The image which was downloaded earlier will be used. In order to connect to the front end of the Jenkins container, the ports which will be mapped to the container are then defined. Then the volume which will hold the data of the container is defined. The folder which was created in the last step is used. After this a network will be defined which will connect the Jenkins container to the various other components of the

CI/CD pipeline which will be defined in the future. Then the container is connected to this network by adding the '-net' to the container configuration. Once the docker-compose file has been created and the folder to store the data from the Jenkins container has been created, the Jenkins service can be started using docker-compose.

NOTE: Ensure that the `jenkins_home` folder is owned by the user that we created. Run an 'll' and if both the user and the group of this folder is not same as the user that was created, change ownership of the folder using `sudo chown userid:userid jenkins_home -R`. If you are unsure of the id of your user run the command 'id'. In the case of this lab it should be 1000

- The next step is to spin up the container using the following command: 'docker-compose up -d'
- In order to check to see if the container is up and running, run thi command: 'docker ps'. If not check that you have given the docker-compose.yml executable permissions and that the current user is the owner of the file.
- Next it is necessary to check the logs to obtain the initial password. This will allow us to sign into the Jenkins front end and create the initial admin user. Run the following command: 'docker logs -f jenkins'. Take note of the initial password.
- Go to your browser and type in the IP address of your VM followed by the ports you defined in the docker-compose file. It should look something like the following 0.0.0.0:8080
- You will be prompted to enter the password take from the logs. Enter this and create a user called 'admin' and create a password. You should be redirected to the Jenkins home screen.
- Click on install the suggested Plugins and allow some time for the installation process to take place.
- Create a user, ensuring to use a strong password.
- Proceed with the standard url provided by Jenkins. It should look something like 0.0.0.0:8080

Now that Jenkins is running within a Docker container, the next sections will cover installing the necessary plugins for the CI/CD pipeline and securing Jenkins using Role-Based authentication.

If you are setting up an AWS EC2 instance with docker and jenkins, ensure your instance has a security group associated with it with the following rules : <https://christopherstoll.org/2016/jenkins-docker-aws-ec2.html>

## 4 Remote Job Execution in Jenkins

This section will demonstrate how to set up remote job execution in Jenkins. A central part of a CI/CD pipeline is the ability to deploy the finished application to the production server.

- In the Jenkins GUI, click on 'Manage Jenkins'. Then click on 'Manage Plugins'

A terminal window with a dark background and light-colored text. The title bar at the top shows a user icon, the text 'jenkins@ip-10-0-76-18:~/jenkins-data', and standard window control buttons (minimize, maximize, close). The terminal content displays a Docker Compose configuration in YAML format. The configuration includes a 'version' field set to '3', a 'services' section with a 'jenkins' service, and a 'networks' section with a 'net' network. The 'jenkins' service is configured with 'container\_name: jenkins', 'image: jenkins/jenkins', 'ports' mapping '8080:8080', 'volumes' mapping '\$PWD/jenkins\_home:/var/jenkins\_home', and 'networks' mapping to 'net'. The 'net' network is defined in the 'networks' section. The cursor is positioned at the end of the 'net' line in the networks section.

```
jenkins@ip-10-0-76-18:~/jenkins-data
version: '3'
services:
  jenkins:
    container_name: jenkins
    image: jenkins/jenkins
    ports:
      - "8080:8080"
    volumes:
      - $PWD/jenkins_home:/var/jenkins_home
    networks:
      - net
networks:
  net:
```

- Next click on the 'Available' tag and type in 'SSH' into the search bar.
- Check the box beside the 'SSH' plugin.
- Click on Install without restart and wait for the download and installation process to complete.
- Check the box that says 'Restart Jenkins after installation'
- Verify that the installation was successful by returning to the 'Manage Plugin' screen and ensuring that SSH is now listed under the 'Installed' tab.

This next section can only be completed once a production VM has been spun up. This VM will act as the production environment, and will be where the built and tested app gets deployed to at the end of the pipeline. For the purpose of testing during the development process, this was initially set up as an open-ssh container and connected to the jenkins container on the development VM. This was later amended in order to more accurately simulate a real-world use case for the purpose of more accurate experiment results.

- Ensure Docker and docker-compose have been installed on the production VM.
- 

## 5 Securing Jenkins

Your fifth section. Change the header and label to something appropriate.

## 6 Email Notifications in Jenkins

This section will demonstrate how to install email plugin in Jenkins. Emails notifications are useful in keeping the admin/developers informed of failed builds. As in any professional environment, developers and administrators are not at their desks around the clock, this feature allows them to keep on top of the pipeline and react faster to any potential outages. Builds can fail due to failed tests, failed environment setups etc.

- First click on 'Manage Jenkins'
- Then 'Manage Plugins'
- Go to the 'Available' and search for 'Mail'
- install and restart the server
- add a post build action to any Jenkins called email notification and add your email address

## 7 Jenkins and Maven

Maven is a powerful build tool which allows us to build Java applications. <https://plugins.jenkins.io/maven-plugin/>. It can also be easily integrated into Jenkins using a plugin.

When Maven is integrated into Jenkins, Continuous Integration can be achieved. The idea being that when a developer commits new code to the SCM (GitLab in this case), the build stage is triggered in which Maven takes the Java source code from the SCM and builds a jar file. This JAR file then is tested and if it passes the requirements, the code is then deployed to the production environment.

- Click on Manage Jenkins and navigate to the Manage Plugins screen
- Click on available and search for Maven
- Check the box beside 'Maven Integration'
- Download and restart Jenkins
- Verify the installation was successful by checking that Maven Integration is now in the 'Installed tab'
- Also ensure that Git client and Git plugin are installed. If you installed suggested plugins during the initial installation of Jenkins, then this should already be done.

We will be utilising the sample Maven application found in the Jenkins docs on Github : <https://github.com/jenkins-docs/simple-java-maven-app.git> We will firstly test Maven by creating a new job which will pull the sample maven app from the git repo found on the link above, and then build a jar file from this and also run a couple of unit tests against the jar.

- Take note of the URL of the git repo. In this case it is <https://github.com/jenkins-docs/simple-java-maven-app.git>

- Go to the Jenkins GUI and click on 'New Item'.
- Specify a 'Freestyle Project' and click 'OK'
- under the 'SCM' tab check the 'Git' box
- in the URL box enter the URL of the github repo. The branch can be specified under the Advanced tab. It is set as master and for the purpose of this test this will be used.
- Click on 'Build Now'
- By checking the 'Console Output' of the build, you can verify that the code was downloaded to the workspace folder in the `jenkins_home` directory. Every job that is created in Jenkins has a workspace. Verify this by going to the VM command line and firstly entering into the jenkins container using : `docker exec -ti jenkins bash`  
Then `'cd /var/jenkins_home/workspace'` all of the jobs we create have a directory in here.
- cd maven-job to verify that the source code has been cloned to the job
- Go to the Jenkins GUI and then go to Manage Jenkins and then go to 'Global Tool Configuration.
- Under the 'Maven' tab , give the installation any name and choose the latest version in the drop down menu as shown below. Then click Save

**Maven**

Maven installations

**Add Maven**

Maven

Name

☒ Install automatically ?

**Install from Apache**

Version

**Add Installer** ▼

**Delete Installer**

**Delete Maven**

- Next return to the maven-job and click 'Configure' Under the build tab click on 'Invoke top-level Maven targets' in the dropdown. Specify the installation name we configured in the previous step.
- In the Goals field enter the following Maven code; `'-B DskipTests clean package'` found on <https://jenkins.io/blog/2017/02/07/declarative-maven-project/>
- Save this job and then click 'Build Now' again. The job will begin to install the necessary Maven packages and then build the jar file. The resulting jar is then saved within the workspace folder under the maven-job directory.

**Build**

**Invoke top-level Maven targets** X ?

Maven Version

Goals

**Advanced...**

**Add build step**

The newest version of the repository is downloaded. Because this is the first time Maven is invoked in the Jenkins installation, it will first download the Maven version that was specified in the global tools configuration. Then the jar is built. Now that Maven has been verified as working in conjunction with Jenkins, we can test the code using the minimal unit tests provided in the sample maven app.

- Go to Configure under the maven job.
- Add another step under the 'Build' tab
- Select Invoke top level maven target from the dropdown.
- Save the config and click Build Now again.
- Check the results in the Console Output

Now that the tests have run and the jar file has passed the test, it's possible to deploy the jar locally in order to examine the output of the application.

- Go back to 'Configure' the maven job
- Go to the Build Step and add another step. Select 'Invoke shell script' from the dropdown
- in the script add the following: `'java -jar /var/jenkins_home/workspace/maven-job/target/my-app-1.0-SNAPSHOT.jar'`
- Save the job and Build again
- Check the Console Output. The output of the build should show 'Hello World!'

The next section will cover configuring GitHub to trigger jobs automatically once new code is committed to the source code.

## 8 Jenkins and Git Source Code Management

In a CI/CD pipeline, the Source Code Management is the first link in the chain. When new code gets committed to the repository of an application, the pipeline is then triggered in which a jar is built and then tested and pushed to deployment. There are a couple of ways of doing this. For the purpose of this laboratory environment, we will be utilising Docker to spin up a local git server which will become the SCM for the pipeline build.

NOTE: If using AWS, GitHub can be used in the same way by utilising the webhooks feature to trigger builds.

The first part of this section involves spinning up a container with a gitlab image installed. This will be done by defining the service in the docker-compose file.

NOTE: This part of the lab involves spinning up a gitlab container in the VM. This is a fairly heavy install and the website recommends a minimum of 4GB of RAM and 2 cores in order for it to run correctly. Please refer to the following link.<https://docs.gitlab.com/ee/install/requirements.html> VMs in VirtualBox can be configured by clicking on the VM and then on Settings. Go to System and there you can modify the memory. Reboot the VM after modification

After this go to the jenkins-data folder ('cd jenkins-data'). Then return to the installation documentation to find the template for installing gitlab using docker-compose. <https://docs.gitlab.com/omnibus/docker/>

- First edit the docker-compose.yml: 'vim docker-compose.yml'
- then add a container name of your choosing.
- refer to the image name found in the gitlab docs.
- give the hostname
- define the ports through which the git server can be accessed
- define the volumes in which the git data will persist
- Finally define the network that the container will connect to. In this case its the one we defined earlier.
- Save the file and exit

After configuring this, its time to spin up the new git server using the following command; 'docker-compose up -d'. The git server will take a few minutes to initialise. This is due to the large size of the git-server. you can check the docker logs(docker logs -f to monitor the progress of the installation. Then go to IP-ADDR:8090(referring to the port we opened) in the browser. Once you do this you should be prompted to change the password. Do this and then set up the root user. The username will be 'root' and the password is whatever you have chosen. Now it is time to create the first repository.

- First create a group. The name is not significant. Define it as private.
- Click on 'New Project' next. The name should be 'maven'
- Next we can create a new user. use whatever name is easiest to remember. Then set a password.

```
jenkins@ip-10-0-76-18:~/jenkins-data
version: '3'
services:
  jenkins:
    container_name: jenkins
    image: jenkins/jenkins
    ports:
      - "8080:8080"
    volumes:
      - $PWD/jenkins_home:/var/jenkins_home
    networks:
      - net
  git:
    container_name: git-server
    image: 'gitlab/gitlab-ce:latest'
    hostname: 'gitlab.example.com'
    ports:
      - '8090:80'
    volumes:
      - '/srv/gitlab/config:/etc/gitlab'
      - '/srv/gitlab/logs:/var/log/gitlab'
      - '/srv/gitlab/data:/var/opt/gitlab'
    networks:
      - net
networks:
  net:
```

- Next got o the repo we created earlier. Go to settings and then project members.
- Then add the new user to the project.

Next we will upload the sample app we created earlier to the new repo. First install git using 'sudo yum -y git'

- Make sure your'e in the jenkins-data folder
- use the following command to clone the repo : 'git clone https://github.com/jenkins-docs/simple-java-maven-app.git'
- Next do a vim /etc/hosts and add the IP address followed by gitlab.example.com(possible change later)
- next clone the empty git repository we created earlier. make sure to pass the credentials into the url so it looks like: username:password@gitlab.example.com
- Next make sure your'e in the new maven folder.Now do a 'cp ../simple-maven-app/\* .'
- Git add . and then git commit -m "first commit" then git commit origin master.
- go back to the git server UI and reload to ensure he files have been uploaded.

The next exercise is integrating the Git server so that when new code is committed to the repository we created, an automated build is triggered. The maven job that was created earlier will be used to test the git integration, then later git will be integrated into the pipeline. This will be achieved using githooks or webhooks. This is the trigger that notifies Jenkins when code is pushed to the repository.



- First go to credentials. then add credentials. the add a username and password which os the same as that for our SCM(GitLab or Github)
- Now configure the Maven job we created earlier. Go to SCM section and change the repo URL to that of the GitHub/GitLab Repo for the source code. Then change the credentials to the ones created in the previous step.Save the job.Build the job to ensure the link was successful.
- login to the git server. docker exec -ti git-server bash
- cd /var/opt/gitlab/git-data/repositories/jenkins/maven.git.
- mkdir custom<sub>hooks</sub>
- <https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks> Then do a vi post-receive and refer to the sample post-receive hook found on the link above. A post receive hook is a server side which which can be used to notify the automation server when changes are made to the source code. This file is saved on the git server, but contains a crumb which calls pings the Jenkins server. We then set up our job to listen for this ping and trigger a build whenever it hears it. Please find below the file. NOTE plase replace the username and password with the ones which we created earlier.

```
#!/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 [ "$branch" == "master" ]; then
  crumb=$(curl -u "jenkins:1234" -s 'http://jenkins:8080/crumbIssuer/api/xml?xpath=concat(//crumbRequestField,"://crumb')
  curl -u "jenkins:1234" -H "$crumb" -X POST http://jenkins:8080/job/maven/build?delay=0sec

  if [ $? -eq 0 ]; then
    echo "*** Ok"
  else
    echo "*** Error"
  fi
fi
```

- this script is basically saying if the branch is equal to master, then trigger the crumb which sends a POST to the url of the maven job which, once referred to as a build step in the job, will trigger the job.
- give this file executable permissions : chmod +x post-receive
- change the owner of the file: chown git:git custom<sub>hooks</sub>/ - R
- exit the git server bash.
- test this git hook by modifying the source code. vi src/main/java/com/mycompany/app/App.java change the string to anything you like.
- vi src/main/java/com/mycompany/app/AppTest.java modify the test file to reflect the change to the main file.

- commit these changes to the git repo. `git add src/ git commit -m "first test of githooks" git push origin master`
- Go back to the jenkins front end in the browser and observe that the maven job has been triggered

## 9 Install Docker inside the Jenkins Container

In this section the instructions will be laid out to install Docker inside the Jenkins container. This is a necessary step in building the CI/CD pipeline. This is because Jenkins will be utilised to build docker images containing the sample application.

- The first step is creating a Dockerfile in which the specifications of the Docker installation will be defined. This tutorial was used <https://docs.docker.com/compose/gettingstarted> Prior to doing this ensure that the Jenkins Pipeline plugins are installed. If you installed the suggested plugins when installing Jenkins, these are already installed.
- In the VM, signed in as the user we created in section 1 and in the jenkins-data folder, create a directory; `mkdir pipeline`
- In this folder, create a file; `vim Dockerfile`. The file should have the same contents as the image below.

```

RUN pip install -U ansible

# Install Docker

RUN apt-get update && \
apt-get -y install apt-transport-https \
ca-certificates \
curl \
gnupg2 \
software-properties-common && \
curl -fsSL https://download.docker.com/linux/$(. /etc/os-release; echo "$ID")/gpg > /tmp/dkey; apt-key add /tmp/dkey && \
add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/$(. /etc/os-release; echo "$ID") \
$(lsb_release -cs) \
stable" && \
apt-get update && \
apt-get -y install docker-ce

# Compose

RUN curl -L "https://github.com/docker/compose/releases/download/1.22.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose && chmod +x /usr/local/bin/docker-compose

RUN usermod -aG docker jenkins

USER jenkins

```

- This file installs Ansible, and installs docker as per the instructions found on the docker website <https://docs.docker.com/install/linux/docker-ce/debian/>. The Jenkins container is a debian distribution so debian instructions are used. Docker compose is then installed. A user 'jenkins' is then added to the docker group.
- Go back to the jenkins data folder and modify the docker-compose.yml file.
- Change the context of the Jenkins container spec. Change the container name, and the build context so that the pipeline folder with the Dockerfile is referenced. Then add the docker.sock file to the container by referring to the folder which it is located in under the volumes heading. The file should look like the image below.

```

version: '3'
services:
  jenkins:
    container_name: jenkins
    image: jenkins/docker
    build:
      context: pipeline
    ports:
      - "8080:8080"
    volumes:
      - $PWD/jenkins_home:/var/jenkins_home
      - "/var/run/docker.sock:/var/run/docker.sock"
    networks:
      - net

```

- The next stage is to run the docker-compose file by using the command 'docker-compose build'
- Now launch the new jenkins container with docker installed by running the command 'docker-compose up -d'
- Next change ownership of the docker sock file so that the jenkins user in the container can execute docker commands. 'sudo chown 1000:1000 /var/run/docker.sock'

NOTE: If you are completing this on an EC2 instance, be advised that the user id '1000' is already taken up by the user which is created when you spun up the instance. This can have a variety of names depending on the distro. For RHEL it was 'ec2-user'. Therefore when you change ownership of the file, you will still get permission denied. The workaround I used was firstly, signing into the Jenkins container as root using 'docker exec -ti -u root jenkins /bin/bash'. Then change the id of the jenkins user to '1001' or whatever the id happens to be of the user you created in the VM. Use the following command; 'usermod -u 1001 jenkins'. To ensure you are selecting the right id, exit the container bash and type 'id'. This will show you the id you need to match within the VM.

- Verify that this has worked by going inside the Jenkins container and doing a 'docker ps' If this has worked you shouldnt get any errors.

## 10 Integrating everything to complete the CI/CD Pipeline

This section will integrate all of the tools and methodologies that have been set up in the previous section to build the CI/CD pipeline for this experiment. The pipeline will consist of four sections; Build, Test, Push and Deploy.

Git, Jenkins,JUnit, Docker and Maven are the 5 main technologies which will be utilised. Git is the source code management. Jenkins, which is triggered by any change to the SCM, will build the docker images containing the sample application. JUnit tests will then run against the sample application. This container will then be pushed to

DockerHub. Maven is the build tool which takes the Java source code and builds a JAR file from which the application is tested and deployed.

The first stage of this process will be to create a Jenkinsfile. This file will define the steps of the pipeline. It is the basis on which the CI/CD process is automated. Each of the scripts which will be developed is referenced in here in its corresponding stage in the pipeline. In the beginning the file will just return a string but as each stage is developed and tested, a script will be added to each stage.

- ensure you are in the pipeline folder created earlier. <https://jenkins.io/doc/book/pipeline/>
- 'vim Jenkinsfile' The file should look like the image below where the stages Build, Test, Push and Deploy are defined.

Next we will develop the pipeline. This stage will utilise bash scripts to automate packaging of the source code and the building of a Docker image of the generated Jar. Each script will then be referred to within Jenkinsfile under the corresponding stage.

- Create a new directory inside the pipeline folder called 'jenkins' mkdir jenkins
- inside this directory define four directories; Build, Test, Push and Deploy
- Within the build directory create two bash scripts; mvn.sh and build.sh. these will automate the process of building the JAR and containerising it.
- Within the test folder define a bash script called mvn.sh This will automate the Junit testing process.
- In the push directory create a file called push.sh. This will contain the code which pushes the application container to DockerHub registry
- define two scripts in the deploy folder. deploy.sh will Deploy the application container to the production VM. publish.sh will run the container in the production VM.

After these files are written then integrate them into the Jenkinsfile

After this ensure that all of the files have executable permissions. As seen in Figure 8; OWASP Dependency Checker is called through the Jenkinsfile which automates the security vulnerability scanning.

Now we commit all of this code to our git repo. This is where Jenkins will look for the code when it executes the pipeline. git status, git add spring-pipeline, git commit -m "Complete pipeline", git push -u origin master. Ensure the code has been committed successfully by checking the git server UI in the browser.

Now it is time to head to the jenkins UI in the browser and setup a job for the CI/CD pipeline. First Go to 'Manage Plugins' and install the OWASP Dependency Checker plugin. restart the server. Then we go to Global Credentials and create the credentials for our git server user so that jenkins can login to access the SCM Click on new item and call this job whatever you like and make sure you define it as a 'Pipeline Project'. Then click on 'Configure Job'. In the configuration ensure that the correct path is listed in th SCM which points to the git server. Remember that the Jenkins container and Git container are communicating through the internal Docker network that was set up in the docker-compose.yml file. This means we need to specify only the name which the git

server is referred to in this file. Then check that Jenkinsfile is listed in the correct place and click 'Build Now'. The pipeline build process will begin.

Finally we can see the resulting pipeline builds.

## 10.1 References

- <https://owasp.org/www-project-dependency-check/>
- <https://docs.docker.com/>
- <https://www.jenkins.io/doc/>
- <https://maven.apache.org/guides/index.html>
- <https://plugins.jenkins.io/>

```

stages {
    stage('Build') {
        steps {
            sh '''
                echo build
            '''
        }
    }

    stage('Test') {
        steps {
            sh 'echo test'
        }
    }

    stage('Push') {
        steps {
            sh 'echo push'
        }
    }

    stage('Deploy') {
        steps {
            sh 'echo deploy'
        }
    }
}
}

```

Figure 1: mvn.sh script

```

#!/bin/bash

echo "*****"
echo "***BUILDING JAR***"
echo "*****"

WORKSPACE=/home/jenkins/jenkins-data/jenkins_home/workspace/pipeline-maven-spring

docker run --rm -v $WORKSPACE/spring-maven-app:/app -v /root/.m2:/root/.m2 -w /app maven:3-alpine "$@"

```

Figure 2: build.sh script

```
#!/bin/bash

# Copy the new jar to the build location
cp -f spring-maven-app/target/*.jar jenkins/build/

echo "*****"
echo "** Building Docker Image **"
echo "*****"

cd jenkins/build/ && docker-compose -f docker-compose-build.yml build --no-cache
```

Figure 3: test script

```
#!/bin/bash

echo "*****"
echo "**TESTING THE CODE**"
echo "*****"

WORKSPACE=/home/jenkins/jenkins-data/jenkins_home/workspace/pipeline-maven-spring

docker run --rm -v $WORKSPACE/spring-maven-app:/app -v /root/.m2:/root/.m2 -w /app maven:3-alpine "$@"
```

Figure 4: push script

```
#!/bin/bash

echo "*****"
echo "**Pushing image **"
echo "*****"

IMAGE="spring-maven-app"

echo "***Docker Login**"
docker login -u cdeeg7293 -p $PASS
echo "***Tagging image**"
docker tag $IMAGE:$BUILD_TAG cdeeg7293/$IMAGE:$BUILD_TAG
echo "*****Pushing Image*****"
docker push cdeeg7293/$IMAGE:$BUILD_TAG
```

Figure 5: deploy script

```
#!/bin/bash

echo spring-maven-app > /tmp/.auth
echo $BUILD_TAG >> /tmp/.auth
echo $PASS >> /tmp/.auth

scp -i /opt/prod /tmp/.auth jenkins@192.168.1.1:/tmp/.auth
scp -i /opt/prod ./jenkins/deploy/publish jenkins@192.168.1.1:/tmp/publish
ssh -i /opt/prod jenkins@192.168.1.1 "/tmp/publish"
```

Figure 6: publish script

```
#!/bin/bash

export IMAGE=$(sed -n '1p' /tmp/.auth) #This navigates the /tmp/.auth file and locates the first line
export TAG=$(sed -n '2p' /tmp/.auth) # finds the second line
export PASS=$(sed -n '3p' /tmp/.auth) #finds the third line which is the password

docker login -u cdeeg7293 -p $PASS
cd ~/maven && docker-compose up -d
```

Figure 7: Jenkinsfile Part 1

```
pipeline {
    agent any

    environment {
        PASS = credentials('registry-pass')
    }

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

            post {
                success {
                    archiveArtifacts artifacts: 'spring-maven-app/target/*.jar', fingerprint: true
                }
            }
        }

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



Figure 8: Jenkinsfile Part 2

```
stage('OWASP Dependency-Check Vulnerabilities') {
    steps {
        dependencyCheck additionalArguments: '''
            -o "./"
            -s "./"
            -f "ALL"
            --prettyPrint'', odcInstallation: 'Default'
        dependencyCheckPublisher pattern: 'dependency-check-report.xml'
    }
}

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

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

Figure 9: Check the Git Server to ensure code was uploaded

The screenshot shows the GitLab web interface for the repository 'jenkins' under the project 'pipeline-maven spring'. The interface includes a sidebar with navigation links (Project, Repository, Files, Commits, Branches, Tags, Contributors, Graph, Compare, Charts) and a main content area displaying the commit history. The commit history table has columns for Name, Last commit, and Last update. The commit history shows three entries: 'jenkins' (adds post build action, 1 day ago), 'spring-maven-app' (Initial commit, 1 day ago), and 'Jenkinsfile' (Adds Dependency checks, 22 hours ago).

Name	Last commit	Last update
jenkins	adds post build action	1 day ago
spring-maven-app	Initial commit	1 day ago
Jenkinsfile	Adds Dependency checks	22 hours ago

Figure 10: Git Reference within the Jenkins Job

The screenshot shows the Jenkins Pipeline configuration interface. The 'Definition' is set to 'Pipeline script from SCM'. The 'SCM' is set to 'Git'. Under 'Repositories', the 'Repository URL' is 'http://git/jenkins/pipeline-maven-spring.git' and 'Credentials' is 'cdeeg1993/\*\*\*\*\*'. The 'Branches to build' section has a 'Branch Specifier (blank for \'any\')' set to '\*/master'. The 'Repository browser' is set to '(Auto)'. There are 'Save' and 'Apply' buttons at the bottom left.

Figure 11: Pipeline execution with Continuous Security

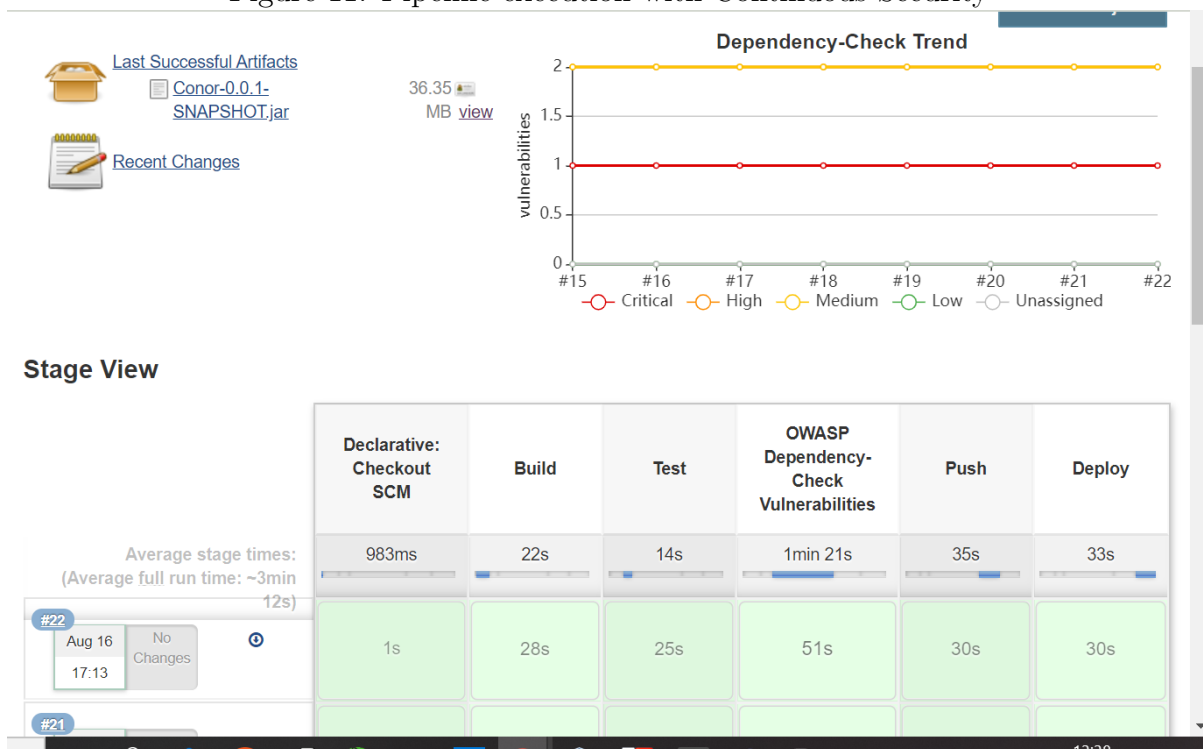
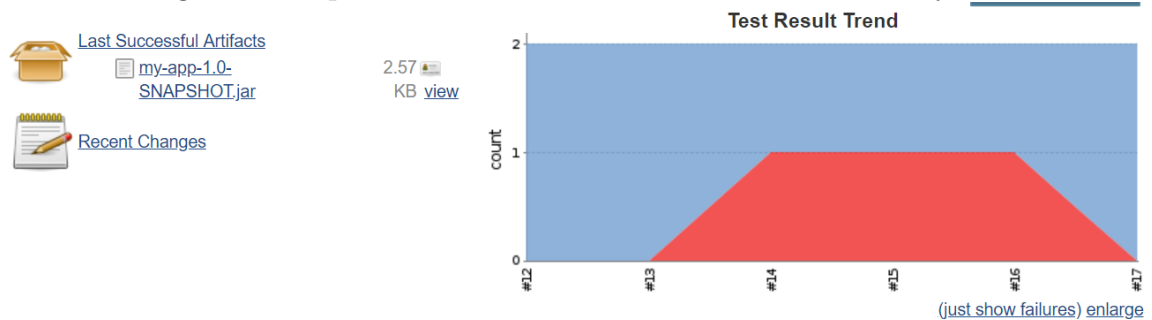


Figure 12: Pipeline execution without Continuous Security



## Stage View

