Intro to Jenkins
Lab 4

1 Lab setup

Ensure that you can access the Jenkins UI in the usual manner.

In all the labs that follow, `userx` refers to your assigned username in the lab guide.

For some of the Docker CLI commands that are provided in these labs that run on their own, these can be executed on the local Docker installation on your Windows machine. However, most of the commands will be run on the Docker platform installed on the Main Server in AWS Cloud since we need to work on the project code base that we already have there. In that case, make sure you precede all your container names with `userx` to distinguish them from the containers started by other users on the same platform.

You should type your Docker commands into the Powershell 7 CLI windows (there may be issues sometimes with using Powershell 5.1).

2 Working with Docker in a CI / CD pipeline

Docker offers many advantages for use in a CI/CD pipeline, but probably the most important advantage it offers is environment standardization across the entire pipeline. A Docker container provides a reproducible, isolated environment that can be easily duplicated and / or reproduced via a DockerFile or Docker Compose YAML configuration files. This provides a standardized environment for build, testing and deploying across different external servers / host machines in a CI / CD pipeline and helps prevent unexpected errors occurring in a fully automated CI/CD pipeline.

https://docs.docker.com/build/ci/

https://dev.to/kodekloud/the-role-of-docker-in-devops-1con

DockerHub is the public Docker registry which hosts millions of images. Many of these images are official images, which are reviewed and vetted by the Docker team for security and reliable.

https://docs.docker.com/docker-hub/official_images/

If you login to your DockerHub account and click on Explore you can use a filter to view only these official images.

You have official images for the main Linux distros (e.g. Ubuntu/Debian and CentOS/Fedora)

https://hub.docker.com/_/debian

https://hub.docker.com/_/ubuntu

https://hub.docker.com/_/centos

https://hub.docker.com/_/fedora

For Windows, there is an image for variants of Windows Server

https://hub.docker.com/_/microsoft-windows

https://learn.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/container-base-images

There are also official images which contain an installation of popular programming languages (Java, Python, C#, JavaScript, etc) on top of a number of well known base OS images:

For OpenJDK (Java), a popular official repo is Eclipse Temurin:
https://hub.docker.com/_/eclipse-temurin

For Python:

https://hub.docker.com/_/python

For ASP.Net Core / C#

https://mcr.microsoft.com/en-us/catalog?search=dotnet

For Mono (open source implementation of .NET framework)

https://hub.docker.com/_/mono

For PHP

https://hub.docker.com/_/php

For Node.JS (JavaScript)

https://hub.docker.com/_/node

Similar, you will official images for popular applications: databases (for e.g. mySQL, postgresql) , web servers (Nginx, Apache Server, Tomcat), etc

https://hub.docker.com/_/mysql
https://hub.docker.com/_/postgres

https://hub.docker.com/_/nginx
https://hub.docker.com/_/httpd
https://hub.docker.com/_/tomcat

Let's examine the official Eclipse Temurin repository which provides official images for OpenJDK binaries
https://hub.docker.com/_/eclipse-temurin

Typically when we develop production grade Java apps, we will develop using a LTS version of Java. Currently there are 3 major LTS versions in use: 8, 11, 17, 21
https://blogs.oracle.com/javamagazine/post/java-long-term-support-lts
https://www.oracle.com/java/technologies/java-se-support-roadmap.html

The tags for the various images are classified based on this info:

- The LTS release version (8, 11.x.x., 17.x.x.)
- Whether it's a JDK or JRE
- The OS that Java is installed on (for e.g. CentOS, NanoServer (Windows), Alpine (light weight OS), Ubuntu (Jammy/Focal): https://wiki.ubuntu.com/Releases

Let's assume that your organization has decided to standardize that all Java apps should be compiled on and executed using the latest version of Java 17 on an Ubuntu OS (Jammy). This means that this particular environment will be used in development, testing / staging and on the production servers of a CI / CD pipeline

Looking through the official Oracle Java site, we can see that the latest version for Java 17 is 17.0.6
https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html

As part of the development team, we need to look for a JDK image for 17.0.6 for Jammy (we need JDK because we are also compiling and the JDK provides tools such as `javac` for compiling). If we do a search on 11.0.17, we will eventually find an image: `17.0.6_10-jdk-jammy`

We can download this to our local Docker registry by copying and pasting the command provided there:

```
docker pull eclipse-temurin:17.0.6_10-jdk-jammy
```

You can check that this image has been downloaded successfully with:

```
docker image ls -a
```

Then we can run a container from this image and open an interactive Bash shell into it at the same time with:

```
docker container run --name userx-firstjava -it eclipse-temurin:17.0.6_10-jdk-jammy /bin/bash
```

Notice you are now established in a Bash shell in the `root` account (the root account is equivalent to the Administrator account in Windows, it has complete privilege to perform anything).

Now open another Powershell via a new SSH connection to the Main Server (or use a new terminal in MobaXTerm) and type:

```
docker container ls -a
```

This shows the list of all containers on the Docker platform. You should see the name of the image that the container is based on, the process that is actively running in the container and its name. On the Main Server, you will see all the containers started up by all the other users that are logged into the Main Server.

Back in the Bash shell in the container, type:

```
java -version
```

Verify that we indeed have the correct version of Open JDK installed (17.0.6) on this container.

To check the OS of this container, type

```
cat /etc/*-release
```

This confirms to us that have the desired OS installed on this container as well.

Type:

```
echo $JAVA_HOME
```

This gives us the home installation directory of Java on this container.

Let's test whether we can successfully execute the JAR file that we had generated earlier in `simple-java-app`

Lets navigate back to the root folder of this project:

```
cd ~/userx/githubprojects/simple-java-app
```

First, lets check the version of Java on the Main Server:

```
java -version
```

Notice that it is different version (11.0.17) from the one in the container (17.0.6).

We are now going to test whether the JAR we generated earlier (which was successfully generated and executed with Java 11.0.17) can also be successfully executed with Java 17.0.6

In the Bash shell in the container, let's first create a folder that we can copy the JAR file from the Main Server (the host) so that we can execute it. Since we are logged in as root account in the container, we have full admin privilege and can create directories anywhere we want. Let's switch back to our home directory and create a directory app there:

```
cd
pwd
mkdir app
ls -l
```

Now in the Powershell window, generate the JAR file by running the standard Maven goal:

```
mvn package
```

Now we will copy the generated JAR into the `/root/app` folder in the running container with:

```
docker cp target/my-app-1.0-SNAPSHOT.jar userx-firstjava:/root/app
```

In the Bash shell in the container, verify that the JAR files has been successfully copied to this location with:

```
ls -l app
```

Now attempt to execute it with:

```
java -jar app/my-app-1.0-SNAPSHOT.jar
```

It should execute perfectly fine.
Although this is very trivial example, it illustrates a very important point:
Using Docker allows you to build, test and deploy your project in multiple environments WITHOUT the need to change the configuration and software on the development, staging / testing and production machines.

So for e.g. you could have:

- development machine (MacOS with JDK 11 installed)
- testing / staging server (Windows 11 and JDK 8 installed)
- production server (Windows Server 2019 and JDK 20 installed)

but as long as all of these machines use the same image (`eclipse-temurin:17.0.6_10-jdk-jammy`) in the CI/CD pipeline, there should not be any issues.

Exit from the Bash shell in the container:

```
exit
```

You can remove this container with:

```
docker container rm userx-firstjava
```

Let's repeat this example with Maven, the build tool.

We can see the entire Maven release history here

https://maven.apache.org/docs/history.html

Lets check the version of Maven on the Main Server:

```
mvn -v
```

We can see that this is the latest version: 3.9.0

Let's assume again that the organization has decided to standardize on using an earlier version of Maven (3.8.7) on Ubuntu (Jammy).

Again, we can repeat the exercise again, but this time to search and find a suitable Maven Docker image at:
https://hub.docker.com/_/maven

The tag contains info regarding the following

- Maven version
- OpenJDK distribution alternatives (`amazoncorretto`, `eclipse-temurin`, etc)
- JDK LTS release versions: 8, 11, 17, 21
- OS base image (alpine, focal) - note that if the final part of the tag does not include an OS base image name, then by default it is the latest version of Ubuntu (at the moment this is jammy).

If we do a search on 3.8.7, we will eventually find an image that most closely matches the JDK image that we already using: `3.8.7-eclipse-temurin-17`

Again we pull this image to our local registry with:

```
docker pull maven:3.8.7-eclipse-temurin-17
```

You can check that this image has been downloaded successfully with:

```
docker image ls -a
```

Then we can run a container from this image and open an interactive Bash shell into it at the same time with:

```
docker container run  --name userx-firstmaven -it maven:3.8.7-eclipse-temurin-17 /bin/bash
```

In the other Powershell CLI, type:

```
docker container ls -a
```

This shows the list of all containers on the Docker platform. You should see the name of the image that the container is based on, the process that is actively running in the container and its name.

Back in the Bash shell in the container, type:

```
mvn -v

java -version
```

Verify that we indeed have the correct version of Maven (3.8.7) and Open JDK installed (17.0.6) on this container.

To check the OS of this container, type

```
cat /etc/*-release
```

This confirms to us that have the desired OS installed on this container as well.

Type:

```
echo $MAVEN_HOME
```

This gives us the home installation directory of Maven on this container.

We will repeat again what we did previously for the case of the Java container, here we will copy the entire project folder: simple-java-app into a suitable directory in the container (we can use the root account home directory `/root`).

In the Powershell window, move to the correct parent directory and perform this copy operation:

```
cd ~/userx/githubprojects
```

Now we will copy the generated JAR into the `/root` folder in the running container with:

```
docker cp simple-java-app userx-firstmaven:/root
```

Back in the Bash shell in the container, change to the home directory of the root account and verify that the copy was successful with:

```
cd
ls -l
cd simple-java-app
```

Now let's run and verify that this particular version of Maven is able to build our project code successfully:

```
mvn clean package
```

You will notice the downloading of a lot of dependencies related to the plugins used in the Maven pom.xml. This is because the local cache of the root account in the container is currently empty (our previous Maven commands were executed on the Main Server and so the dependencies have already been stored on the cache there).

Finally, we execute the JAR artifact in the usual fashion.

```
java -jar target/my-app-1.0-SNAPSHOT.jar
```

So again we confirm that although we originally performed our development and build process using Maven 3.9.0, we verified that it can still work using an earlier version of Maven 3.8.7.

In a real life scenario, we would not actually even need to have Maven or Java installed on our development machine ! All we need is just an IDE of our choice to work on the source code. Then once we need to perform a build and / or execution, we use the image that the Dev / Ops team has standardized on in the way just demonstrated.

A slightly more efficient way to perform all the activity that we have just done is to mount the project directory on the host (the Main Server) into a directory in the container, and then just execute the relevant Maven goals in the root of that project directory.

Exit the Bash shell in the container:

```
exit
```

Remove the container you just created with:

```
docker container rm userx-firstmaven
```

In one of the Windows Powershell, switch over to the `simple-java-app` folder

```
cd ~/userx/githubprojects/simple-java-app
```

Then issue this command:

```
docker container run --rm -it --name userx-mavenproject -v "$(pwd)":/usr/src/mymaven -w /usr/src/mymaven maven:3.8.7-eclipse-temurin-17 mvn clean package
```

Here, we mount our current directory `simple-java-app` into `/usr/src/mymaven` in the container, execute the `mvn clean package` goal on it, and the resulting JAR is placed back in our local directory because of the volume mapping. The `--rm` also deletes the container after we are done with this operation. In effect, we are "borrowing" the specific Maven / Java environment within the container to perform the build for us. As mentioned before, this saves us the hassle of installing a different version of Maven / Java on our system, and in fact, we don't even **NEED TO HAVE** Maven / Java installed on our system. All we need is Docker 😊

Let's repeat the same process on the generated JAR. Assume we now want to execute it and verify the result using the JDK from the image we pulled down earlier: `eclipse-temurin:17.0.6_10-jdk-jammy`

We can issue this command:

```
docker container run --rm -it --name userx-javaproject -v "$(pwd)":/usr/src/mymaven -w /usr/src/mymaven eclipse-temurin:17.0.6_10-jdk-jammy java -jar target/my-app-1.0-SNAPSHOT.jar
```

Before moving on, lets clean up the project folder by using the Maven image again but this time only with the `clean` goal:

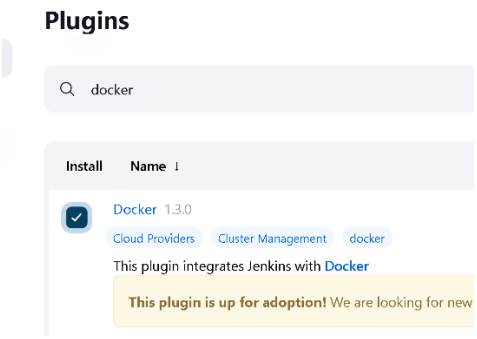
```
docker container run --rm -it --name userx-mavenproject -v "$(pwd)":/usr/src/mymaven -w /usr/src/mymaven maven:3.8.7-eclipse-temurin-17 mvn clean
```

3 Using Docker in a Freestyle project

We are now going to automate everything we have done so far using Docker in a Jenkins freestyle project. Again the idea is the same: we standardize on the same image to be used at all stages: build, testing / staging and deployment.

The action below of installing a plugin only needs to be performed once by a user with Admin Privilege.

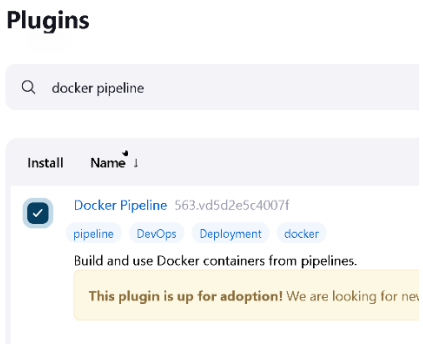
From the main Jenkins Dashboard, Manage Jenkins -> Manage Plugins -> Available Plugins, look for Docker and select the item.



Click Install without Restart.

After download progress is complete, check the box: Restart Jenkins when installation is complete and no jobs are running. Wait for Jenkins to restart.

Repeat this for another plugin: Docker Pipeline

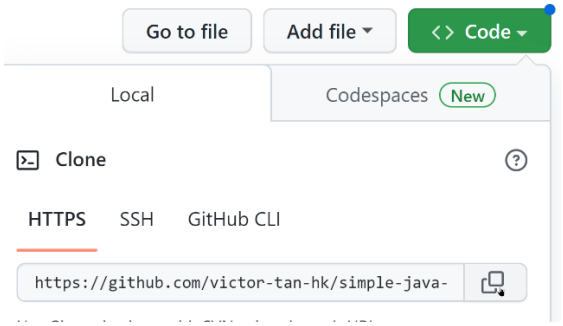


Once Jenkins has restarted, confirm again that both plugins have been installed correctly

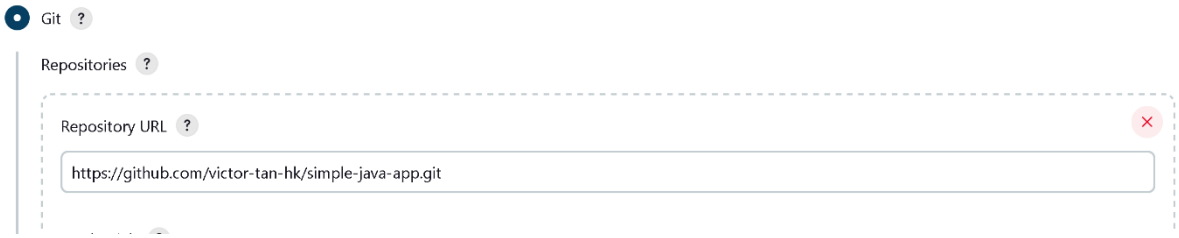
From the main Jenkins dashboard, create a new Item and name it: *userx*-Docker-GitHubJob
Make this a Freestyle Project and select Ok.

In the Configure section, go to Source Code Management and select Git

Return to the main page of your GitHub repo, click on the Green Code button and copy the HTTPs URL:



Paste the URL into the Repository URL on the Jenkins configuration page.



Click on Save and perform a build.

The Console Output for this build attempt will show that the remote repo contents have been cloned into the workspace folder for this current job (*userx*-Docker-GitHubJob).

In the main area for the job, select Configure, go to Build Steps and add a Build Step -> Execute Shell:

```
docker container run --rm --name userx-mavenproject -v "$(pwd)":/usr/src/mymaven -w /usr/src/mymaven maven:3.8.7-eclipse-temurin-17 mvn clean package
```

This performs the build stage.

Click on Save and perform a build.

The build may take some time to complete because the newly created container will again need to download all the dependencies it needs. When the build completes successfully, the Console output should show the log related to all these dependencies being downloaded, with the JAR artifact being generated successfully at the end.

You can check in the Workspace item from the navigation pane to verify the final JAR artifact was generated successfully.

In the main area for the job, select Configure, go to Build Steps and add another Build Step -> Execute Shell:

```
docker container run --rm --name userx-javaproject -v "$(pwd)":/usr/src/mymaven -w /usr/src/mymaven eclipse-temurin:17.0.6_10-jdk-jammy java -jar target/my-app-1.0-SNAPSHOT.jar
```

Click Save and run a build again. Verify from the Console Output that the JAR was executed successfully with its corresponding output.

Again, keep in mind that we are using the containers generated from the respective Docker images to perform the Maven build goals and execute the JAR. We are NOT using the locally installed Maven / Java of Jenkins.

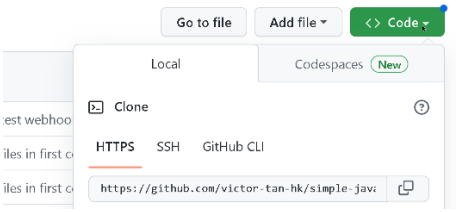
4 Using Docker in a Pipeline project

We will replicate the functionality we performed just now in a pipeline project.

From the main Jenkins dashboard, create a new Item and name it: *userx*-Pipeline-Docker-GitHub
Make this a Pipeline and select Ok.

Select the Pipeline section and copy and paste the script from: `pipeline-docker-github`

Modify the script to specify the HTTPS URL for the GitHub repo for `simple-java-app` that you had created earlier:



Run a build and check the Console Output as well as the Workspaces link that the JAR artifact has been generated and executed correctly.

Some points to note:

- The global variable `docker` here is actually an implicit shortcut for the various steps in the Docker pipeline plugin
  https://www.jenkins.io/doc/pipeline/steps/docker-workflow/
- Using `docker` in the `agent` section for each individual stage of the build to allows us to specify a different container to be used at each stage.
- Any container created here will not persist after the job build completes (i.e. it has the same effect to performing a `docker container run --rm` ….). This is different from the Docker CLI commands used in the `sh` step of a free style project.
- When running a build involving the use of `docker`, you will get a message to this effect: `Jenkins does not seem to be running inside a container`. This is a basic informational message and is not an error because typically Jenkins itself is often started inside a container

https://hub.docker.com/r/jenkins/jenkins

When you use docker to create a container for either a single stage or all stages of a pipeline, the workspace job folder is mounted (bind mount) to a folder with an identical name in the container. This is functionally identical to what we explicitly specified in the freestyle project with:

```
docker container run --rm --name userx-javaproject -v "$(pwd)":/usr/src/mymaven -w /usr/src/mymaven eclipse-temurin:17.0.6_10-jdk-jammy java -jar target/my-app-1.0-SNAPSHOT.jar
```

This essentially means we are using the container to provide us access to specific versions of executables (e.g. Java, Maven) that are typically used in build activities, with the build action being performed on files in the job workspace folder through the bind mount.