Lab: Introduction to Containers, Docker, and IBM Cloud Container Registry Using Java

Objectives

In this lab, you will:

- Pull an image from Docker Hub
- Run an image as a container using `docker`
- Build an image using a Dockerfile
- Push an image to IBM Cloud Container Registry

> **Note: Kindly complete the lab in a single session without any break because the lab may go on offline mode and may cause errors. If you face any issues/errors during the lab process, please logout from the lab environment. Then clear your system cache and cookies and try to complete the lab.**

Important:
You may already have an IBM Cloud account and may even have a namespace in the IBM Container Registry (ICR). However, in this lab **you will not be using your own IBM Cloud account or your own ICR namespace**. You will be using an IBM Cloud account that has been automatically generated for you for this exercise. The lab environment will *not* have access to any resources within your personal IBM Cloud account, including ICR namespaces and images.

Verify the environment and command line tools

1. Open a terminal window by using the menu in the editor: `Terminal > New Terminal`.

   > **Note: If the terminal is already opened, please skip this step.**

2. Verify that `docker` CLI is installed.

   ```
   docker --version
   ```

You should see the following output, although the version may be different:

3. Verify that `ibmcloud` CLI is installed.

```
ibmcloud version
```

You should see the following output, although the version may be different:

4. Change to your project folder.

Note: If you are already on the '/home/project' folder, please skip this step.

```
cd /home/project
```

5. Clone the git repository that contains the artifacts needed for this lab, if it doesn't already exist.

```
[ ! -d 'xiuph-Intro-Kubernetes-java' ] && git clone https://github.com/ibm-developer-skills-network/xiuph-Intro-Kubernetes-java.git
```

6. Change to the directory for this lab by running the following command. `cd` will change the working/current directory to the directory with the name specified, in this case **xiuph-Intro-Kubernetes-java/labs/1_ContainersAndDocker**.

   ```
   cd xiuph-Intro-Kubernetes-java/labs/1_ContainersAndDocker/
   ```

7. List the contents of this directory to see the artifacts for this lab.

   ```
   ls
   ```

Pull an image from Docker Hub and run it as a container

1. Use the `docker` CLI to list your images.

   ```
   docker images
   ```

You should see an empty table (with only headings) since you don't have any images yet.

2. Pull your first image from Docker Hub.

   ```
   docker pull hello-world
   ```

3. List images again.

   ```
   docker images
   ```

You should now see the `hello-world` image present in the table.

4. Run the `hello-world` image as a container.

   ```
   docker run hello-world
   ```

You should see a **‘Hello from Docker!’** message.

There will also be an explanation of what Docker did to generate this message.

5. List the containers to see that your container ran and exited successfully.

   ```
   docker ps -a
   ```

Among other things, for this container, you should see a container ID, the image name (`hello-world`), and a status that indicates that the container exited successfully.

6. Note the CONTAINER ID from the previous output and replace the **<container_id>** tag in the command below with this value. This command removes your container.

   ```
   docker container rm <container_id>
   ```

7. Verify that the container has been removed. Run the following command.

```
docker ps -a
```

Congratulations on pulling an image from Docker Hub and running your first container! Now let's try and build our own image.

Build an image using a Dockerfile

1. The current working directory contains a Java Spring Boot application that we will run in a container. The app will print a hello message along with the hostname. The following files are needed to run the app in a container:

- pom.xml defines the dependencies of the application.
- src/ contains the Java source code.
- Dockerfile defines the instructions Docker uses to build the image.

2. Use the Explorer to view the files needed for this app. Click the Explorer icon (it looks like a sheet of paper) on the left side of the window, and then navigate to the directory for this lab: `Intro-Kubernetes-java>labs>1_ContainersAndDocker`. Click Dockerfile to view the commands required to build an image.

3. The Dockerfile follows a multi-stage build process:

3.1. Build stage

- Uses `eclipse-temurin:21-jdk-jammy` as the base image.
- Installs Maven and fetches dependencies.
- Copies the source code and builds the application using `mvn package`.

3.2. Runtime stage

- Uses `eclipse-temurin:21-jre-jammy` as the base image.
- Copies the built `JAR` file from the build stage.
- Runs the application as a non-root user.
- Exposes port `8081` for the containerized application.

> **You can refresh your understanding of the commands mentioned in the Dockerfile below:**
>
> FROM initializes a new build stage and specifies the base image that subsequent instructions will build upon.
>
> COPY copies files from the local system into the container image.
>
> RUN executes commands inside the image during the build process.
>
> EXPOSE documents that the container listens on a specified port (but does not actually publish it).
>
> ENTRYPOINT specifies the command that runs when a container starts and allows arguments to be passed at runtime.

3. Run the following command to build the image:

```
docker build . -t myimage:v1
```

As seen in the module videos, the output creates a new layer for each instruction in the Dockerfile.

4. List images to see your image tagged `myimage:v1` in the table.

   ```
   docker images
   ```

Note that compared to the `hello-world` image, this image has a different image ID. This means that the two images consist of different layers—in other words, they're not the same image.

Run the image as a container

1. Now that your image is built, run it as a container with the following command:

   ```
   docker run -dp 8081:8081 myimage:v1
   ```

The output is a unique code allocated by Docker for the application you are running.

2. Run the `curl` command to ping the application as given below.

   ```
   curl localhost:8081
   ```

If you see the output as above, it indicates that **'Hello from Java and Kubernetes!'.**

3. Now to stop the container, we use `docker stop` followed by the container id. The following command uses `docker ps -q` to pass in the list of all running containers:

   ```
   docker stop $(docker ps -q)
   ```

4. Check if the container has stopped by running the following command:

   ```
   docker ps
   ```

Push the image to IBM Cloud Container Registry

1. The environment should have already logged you into the IBM Cloud account that has been automatically generated for you by the Skills Network Labs environment. The following command will give you information about the account you're targeting:

   ```
   ibmcloud target
   ```

2. The environment also created an IBM Cloud Container Registry (ICR) namespace for you. Since Container Registry is multi-tenant, namespaces are used to divide the registry among several users. Use the following command to see the namespaces you have access to:

   ```
   ibmcloud cr namespaces
   ```

You should see two namespaces listed starting with `sn-labs`:

- The first one with your username is a namespace just for you. You have full *read* and *write* access to this namespace.
- The second namespace, which is a shared namespace, provides you with only Read Access

3. Ensure that you are targeting the region appropriate to your cloud account, for instance `us-south` region where these namespaces reside as you saw in the output of the `ibmcloud target` command.

   ```
   ibmcloud cr region-set us-south
   ```

4. Log your local Docker daemon into IBM Cloud Container Registry so that you can push to and pull from the registry.

   ```
   ibmcloud cr login
   ```

5. Export your namespace as an environment variable so that it can be used in subsequent commands.

   ```
   export MY_NAMESPACE=sn-labs-$USERNAME
   ```

6. Tag your image so that it can be pushed to IBM Cloud Container Registry.

   ```
   docker tag myimage:v1 us.icr.io/$MY_NAMESPACE/hello-world:1
   ```

7. Push the newly tagged image to IBM Cloud Container Registry.

   ```
   docker push us.icr.io/$MY_NAMESPACE/hello-world:1
   ```

   Note: If you have tried this lab earlier, there might be a possibility that the previous session is still persistent. In such a case, you will see a **'Layer already Exists'** message instead of the **'Pushed'** message in the above output. We recommend you to proceed with the next steps of the lab.

8. Verify that the image was successfully pushed by listing images in Container Registry.

   ```
   ibmcloud cr images
   ```

Optionally, to only view images within a specific namespace.

```
      ibmcloud cr images --restrict $MY_NAMESPACE
```

You should see your image name in the output.

Congratulations! You have completed the second lab for the first module of this course.

Summary:

In this lab, you successfully containerized a Java application using Docker and IBM Cloud Container Registry. You built and deployed a custom Docker image, pushed it to the cloud, and tested its functionality.

Author

Sapthashree K S