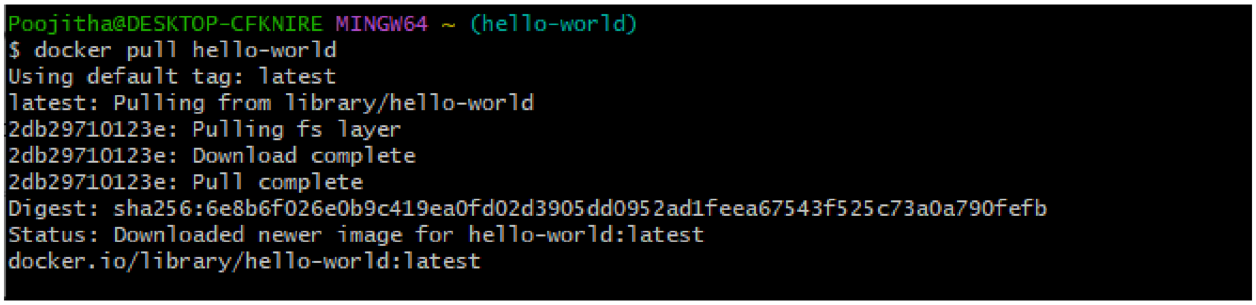
DevOps Assignment

Q1) Pull any image from the docker hub, create its container, and execute it showing the output.

Open a terminal window on your local machine.

Use the command to pull an image from Docker Hub:

docker pull hello-world

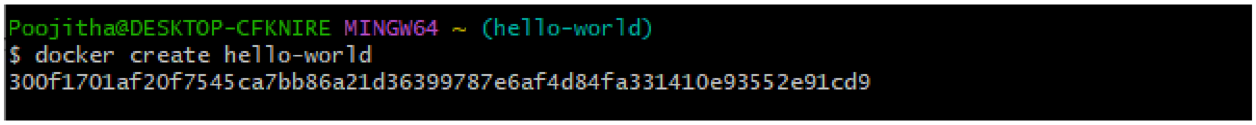


```
Poojitha@DESKTOP-CFKNIRE MINGW64 ~ (hello-world)
$ docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
2db29710123e: Pulling fs layer
2db29710123e: Download complete
2db29710123e: Pull complete
Digest: sha256:6e8b6f026e0b9c419ea0fd02d3905dd0952ad1feea67543f525c73a0a790fefb
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

This command will download the hello-world image from Docker Hub.

Now use the command to create a container from the image:

docker create hello-world



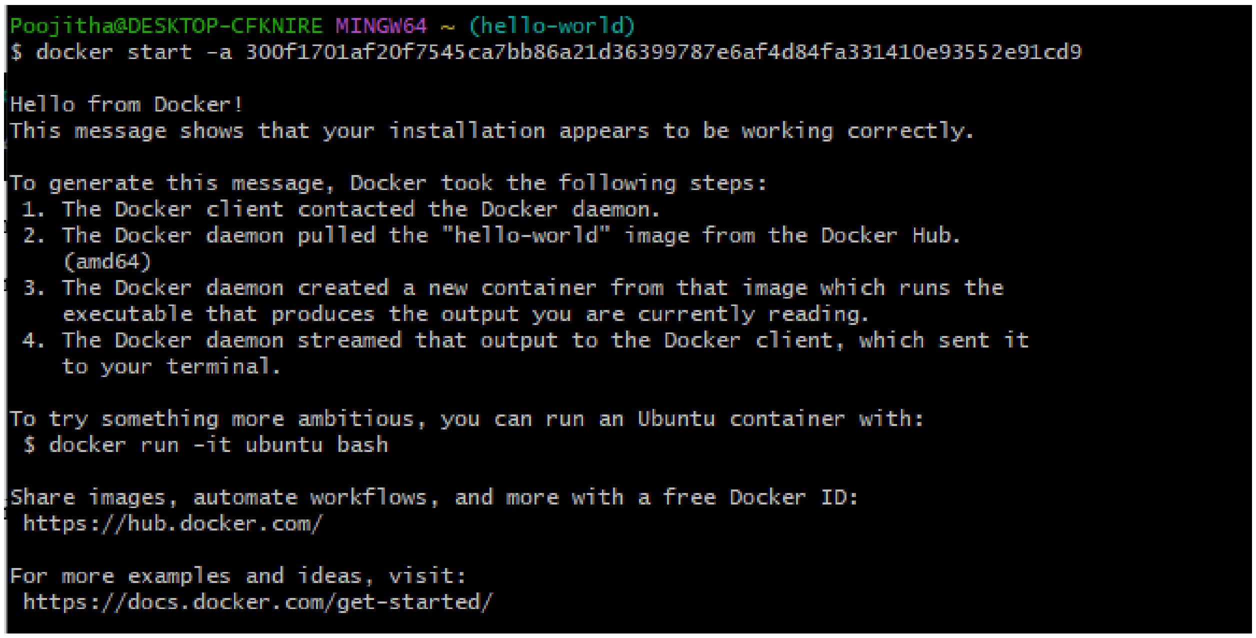
```
Poojitha@DESKTOP-CFKNIRE MINGW64 ~ (hello-world)
$ docker create hello-world
300f1701af20f7545ca7bb86a21d36399787e6af4d84fa331410e93552e91cd9
```

This command will help in creating a container from the hello-world image.

The following command is used to start the container:

docker start -a <container_id>

The output of the hello-world container will be displayed in the terminal window.



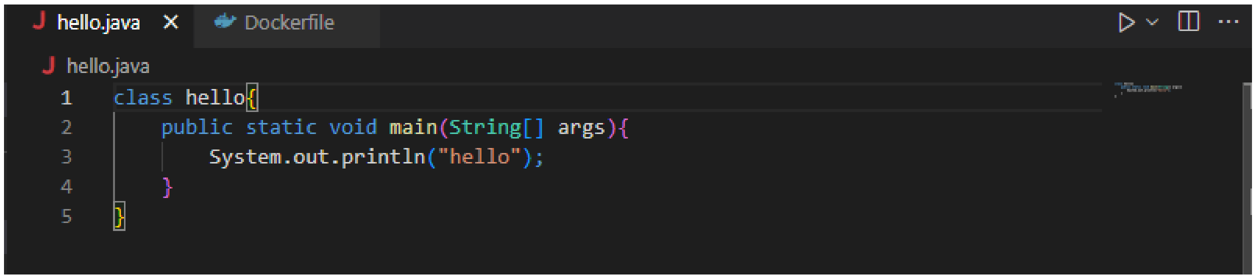
```
Poojitha@DESKTOP-CFKNIRE MINGW64 ~ (hello-world)
$ docker start -a 300f1701af20f7545ca7bb86a21d36399787e6af4d84fa331410e93552e91cd9

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Q2) Create the basic java application, generate its image with necessary files, and execute it with docker.

Create a Java application

Create a new directory for your Java application and navigate into it.

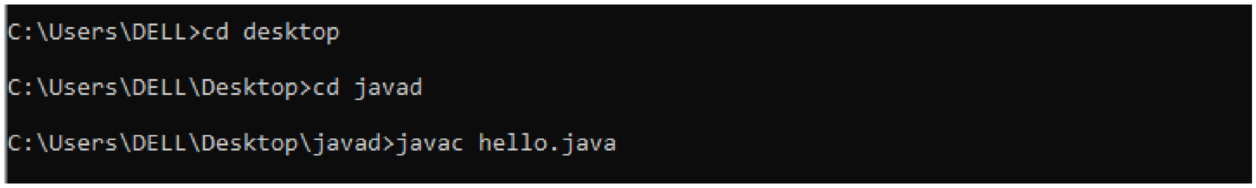
Create a file named hello.java with the following code:



```java
class hello{
    public static void main(String[] args){
        System.out.println("hello");
    }
}
```

Compile the Java application

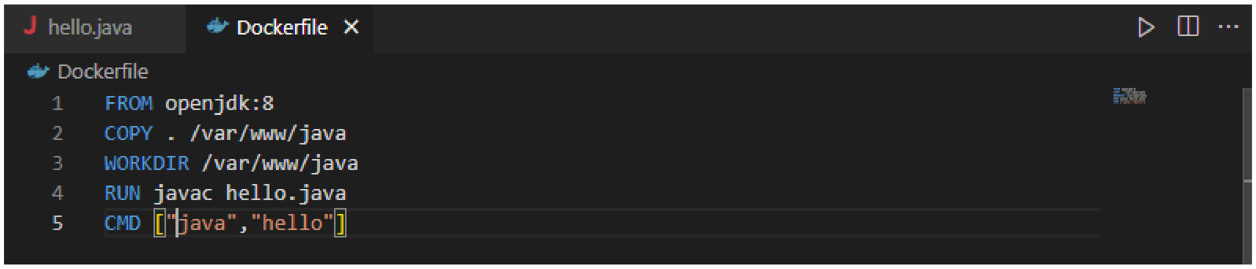
Compile the hello.java file by running the following command in your terminal or command prompt:



```
C:\Users\DELL>cd desktop

C:\Users\DELL\Desktop>cd javad

C:\Users\DELL\Desktop\javad>javac hello.java
```

Create a Dockerfile

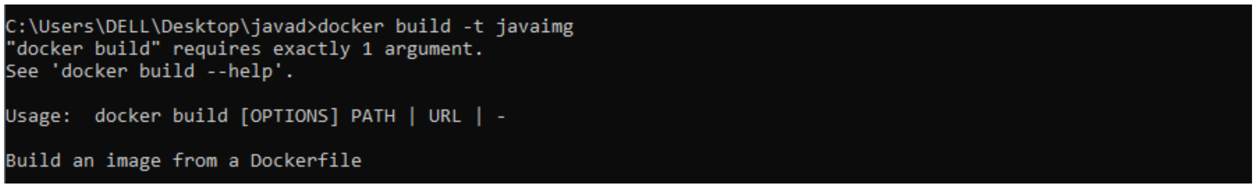
Create a new file named Dockerfile in your Java application directory with the following code:



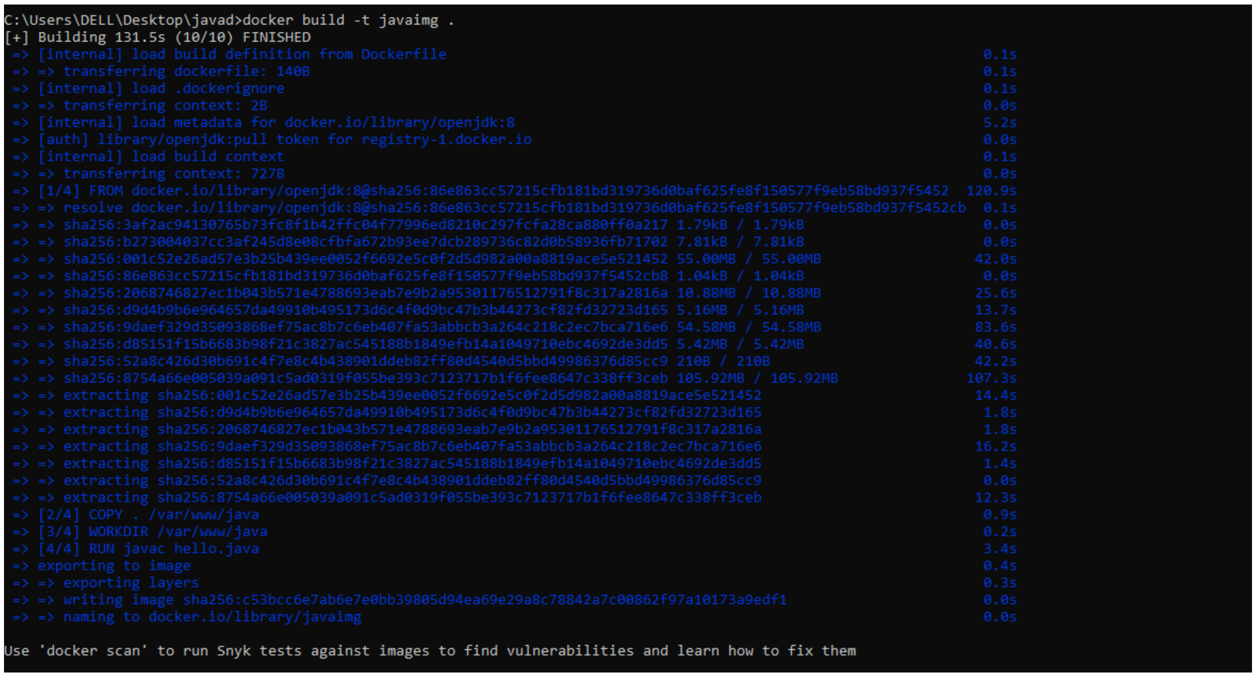
```
FROM openjdk:8
COPY . /var/www/java
WORKDIR /var/www/java
RUN javac hello.java
CMD ["java","hello"]
```

Now build the Docker image

Run the following command to build the Docker image:



```
C:\Users\DELL\Desktop\javad>docker build -t javaimg
"docker build" requires exactly 1 argument.
See 'docker build --help'.

Usage:  docker build [OPTIONS] PATH | URL | -

Build an image from a Dockerfile
```

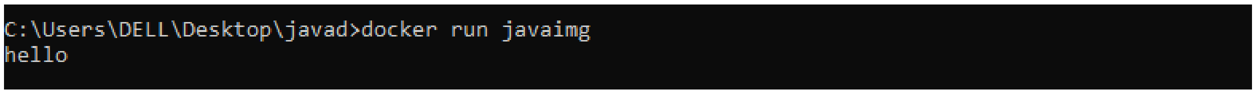


```
C:\Users\DELL\Desktop\javad>docker build -t javaimg .
[+] Building 131.5s (10/10) FINISHED
 => [internal] load build definition from Dockerfile                                                        0.1s
 => => transferring dockerfile: 140B                                                                        0.1s
 => [internal] load .dockerignore                                                                           0.1s
 => => transferring context: 2B                                                                             0.0s
 => [internal] load metadata for docker.io/library/openjdk:8                                                5.2s
 => [auth] library/openjdk:pull token for registry-1.docker.io                                              0.0s
 => [internal] load build context                                                                           0.1s
 => => transferring context: 727B                                                                           0.0s
 => [1/4] FROM docker.io/library/openjdk:8@sha256:86e863cc57215cfb181bd319736d0baf625fe8f150577f9eb58bd937f5452  120.9s
 => => resolve docker.io/library/openjdk:8@sha256:86e863cc57215cfb181bd319736d0baf625fe8f150577f9eb58bd937f5452cb  0.1s
 => => sha256:3af2ac94130765b73fc8f1b42ffc04f77996ed8210c297fcfa28ca880ff0a217 1.79kB / 1.79kB              0.0s
 => => sha256:b273004037cc3af245d8e08cfbfa672b93ee7dcb289736c82d0b58936fb71702 7.81kB / 7.81kB              0.0s
 => => sha256:001c52e26ad57e3b25b439ee0052f6692e5c0f2d5d982a00a8819ace5e521452 55.00MB / 55.00MB           42.0s
 => => sha256:86e863cc57215cfb181bd319736d0baf625fe8f150577f9eb58bd937f5452cb8 1.04kB / 1.04kB              0.0s
 => => sha256:2068746827ec1b043b571e4788693eab7e9b2a95301176512791f8c317a2816a 10.88MB / 10.88MB           25.6s
 => => sha256:d9d4b9b6e964657da49910b495173d6c4f0d9bc47b3b44273cf82fd32723d165 5.16MB / 5.16MB             13.7s
 => => sha256:9daef329d35093868ef75ac8b7c6eb407fa53abbcb3a264c218c2ec7bca716e6 54.58MB / 54.58MB           83.6s
 => => sha256:d85151f15b6683b98f21c3827ac545188b1849efb14a1049710ebc4692de3dd5 5.42MB / 5.42MB             40.6s
 => => sha256:52a8c426d30b691c4f7e8c4b438901ddeb82ff80d4540d5bbd49986376d85cc9 210B / 210B                 42.2s
 => => sha256:8754a66e005039a091c5ad0319f055be393c7123717b1f6fee8647c338ff3ceb 105.92MB / 105.92MB        107.3s
 => => extracting sha256:001c52e26ad57e3b25b439ee0052f6692e5c0f2d5d982a00a8819ace5e521452                  14.4s
 => => extracting sha256:d9d4b9b6e964657da49910b495173d6c4f0d9bc47b3b44273cf82fd32723d165                   1.8s
 => => extracting sha256:2068746827ec1b043b571e4788693eab7e9b2a95301176512791f8c317a2816a                   1.8s
 => => extracting sha256:9daef329d35093868ef75ac8b7c6eb407fa53abbcb3a264c218c2ec7bca716e6                  16.2s
 => => extracting sha256:d85151f15b6683b98f21c3827ac545188b1849efb14a1049710ebc4692de3dd5                   1.4s
 => => extracting sha256:52a8c426d30b691c4f7e8c4b438901ddeb82ff80d4540d5bbd49986376d85cc9                   0.0s
 => => extracting sha256:8754a66e005039a091c5ad0319f055be393c7123717b1f6fee8647c338ff3ceb                  12.3s
 => [2/4] COPY . /var/www/java                                                                              0.9s
 => [3/4] WORKDIR /var/www/java                                                                             0.2s
 => [4/4] RUN javac hello.java                                                                              3.4s
 => exporting to image                                                                                      0.4s
 => => exporting layers                                                                                     0.3s
 => => writing image sha256:c53bcc6e7ab6e7e0bb39805d94ea69e29a8c78842a7c00862f97a10173a9edf1                0.0s
 => => naming to docker.io/library/javaimg                                                                  0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

This will build a Docker image named javaimg using the Dockerfile in your Java application directory.

Run the Docker container

Run the following command to start a Docker container from the javaimg image:



```
C:\Users\DELL\Desktop\javad>docker run javaimg
hello
```

This will start a Docker container and execute your Java application. You will see "hello" printed to the console.

GitHub link- https://github.com/poojitha2803/Herovired