**Docker**

Install Docker desktop, enable Hyper-V in windows 10 and install Linux kernel as per the docker pop window and Restart the computer.

Once sign in back to computer, go to command line and type

***docker –version***

Note: Hyper-V is a Windows 10 virtualization technology and is used to let us create multiple virtual machines and run different OSes in one single Windows 10 OS.

**Docker Registry:**

When we run the below command, we are downloading docker image from some remote location called *docker registry (*[*https://hub.docker.com*](https://hub.docker.com)*)*

***docker run in28min/todo-rest-api-h2:1.0.0.RELEASE***

Docker Registry contains lot of repositories and lot applications under each repository with different versions.

*What does the image contains?*

Images contains all the dependencies and software that are needed for our application to run. Like for example Java, all the dependent libraries.

An image is static and is just a set of bytes when it is present in the registry and even when it is downloaded to a local computer it is just an image but when it is running then it is called container.

When we run the below command and try to access the application in browser we will have issues.

***docker run in28min/todo-rest-api-h2:1.0.0.RELEASE***

So we are adding ***-p {Host Port}:{Container Port}***

***docker run -p 5000:5000 in28min/todo-rest-api-h2:1.0.0.RELEASE***

This is because the container that we run is part of docker bridge network (kind of docker’s internal network) and is not accessible directly until you expose this to the system where the container is running. Hence we adding above attribute to map internal port to host port.

Another handy attribute is detach, -d

When we put -d in the docker run command, we are actually detaching it from the terminal. So it will be running in the background. Once we add this and run, then we will get a container id.

***docker run -d -p 5000:5000 in28min/todo-rest-api-h2:1.0.0.RELEASE***

In order to see the logs, we use

***docker logs containerID***

here the containerId for logs can be part of full id. It doesn’t have to be full id. We can also add -f to follow the tail of logs.

***docker logs -f containerID***

In order to see what all the containers that are running, we can use below command

***docker container ls***

If we want to see images that are downloaded to our local computer

***docker images***

If we want to see all the containers that either running or stopped

***docker container ls -a***

In order to stop a container

***docker container stop containerID***

**Docker Architecture**



Docker Client

Docker demon

Containers

Local images

Image Registry

Mysql

Eurek

Your-app



When we install docker in our computer we will get both docker client and docker demon.



Docker client is where we type all the commands and these commands are sent to docker demon, and docker demon is the one that runs the command. This is also a client server architecture. Docker demon manages all the locally run containers, local images and pulling images from image registry or pushing locally build image to image registry.

**Why is Docker popular?**

**Playing with Docker**

We already know how to see the list of images in our local computer using docker command

***docker images***

Now if we want to change the tag/version of the image we can do that using

***docker tag in28min/todo-rest-api-h2:1.0.0.RELEASE in28min/todo-rest-api-h2:1.0.0.latest***

In order to pull any images then use the below command

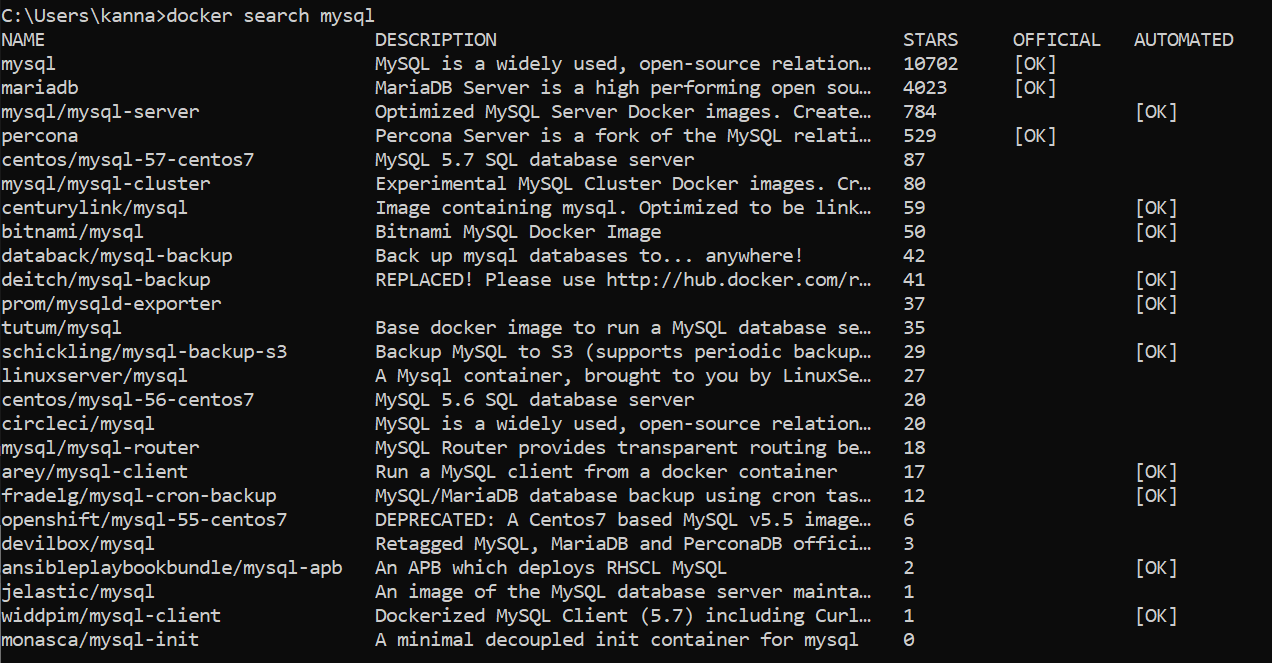
***docker pull imageName***

eg: docker pull mysql

here mysql is an official image

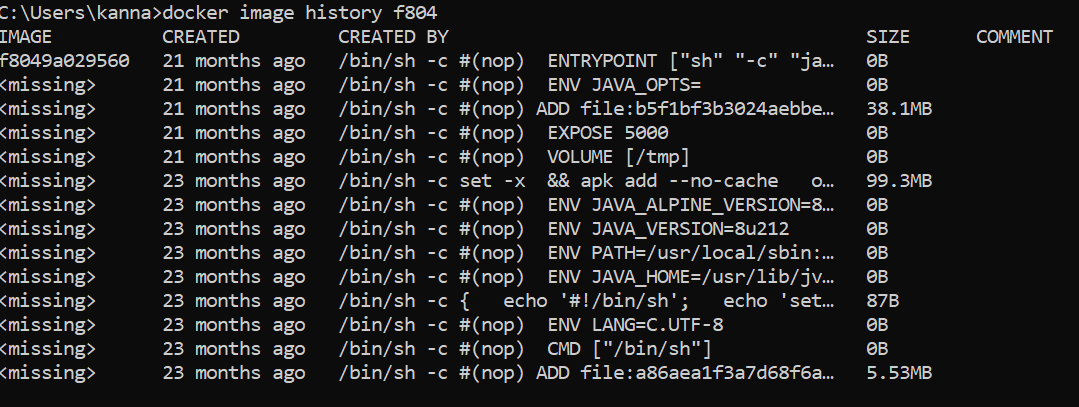
now if we want to search images we can use below command and make sure if it is official.

***docker search mysql***



If we need to get all the steps involved in creating an image then use the below command

***docker image history imageId***



If we want to see the configuration related to the image then following command

***docker image inspect imageID***

In order to remove the image from local memory, then use the below command

***docker image remove imageId***

**Playing with docker container**

Previously we were using the below command format to run a container..

***docker run …***

but we can also run a container by including or specifying container in the command

***docker container run……….***

We can **pause** and **unpause** the docker container

***docker container pause containerId***

***docker container unpause containerId***

We can inspect the docker container using below command.

***docker container inspect containerID***

Another interesting command is prune. This is used to remove all the stopped containers.

***docker container prune***

Earlier we have see docker stop command which gracefully shutsdown the docker container. This means demon will take sometime to gracefully shutdown container.

***docker container stop containerID***

We can use docker container kill command to kill the command immediately without giving it anytime. We generally won’t use this most of the times.

***docker container kill containerID***

Now there is a docker policy for restarting a container. If we have a container in docker before restarting a docker desktop, then based on the --restart policy, we can tell docker demon whether to bring specific containers up after it restarts.

***docker run -p 5000:5000 -d --restart=always in28min/todo-rest-api-h2:1.0.0.RELEASE***

Here restart can have two values **always** and **no.** The default value is no.

Now there is a command called ***docker events***, to see what is happening with the containers.

There is another interesting command called docker top, this one checks what is the top process which is running in a specific container.

***docker top containerID***

Another interesting command is docker stats. It give memory, cpu usage percentage of running containers.

***docker stats***

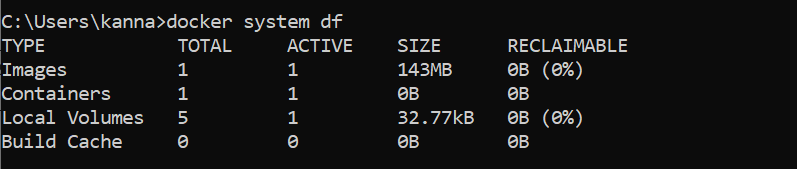
Now for a specific container if we want to allocate cpu and memory limits, we can do that using below command

***docker run -p 5000:5000 -m 512m --cpu-quota 5000 -d in28min/todo-rest-api-h2:1.0.0.RELEASE***

**Note**: 100,000 -100% CPU, 5000 – 5% CPU

Another interesting command is ***docker system df***

This command gives details of all the things docker demon manages.



Until now we have been playing with images that were already created. Now lets focus on creating our own images.

**Manual way of creating images**

Building image for hello world application

1. Build a jar file
2. Setup the prerequisites to run the jar file

openjdk:8-jdk-alpine (Alpine – A minimal docker image based on alpine Linux. Only bare essentials to run the jar)

1. Copy the jar
2. Run the jar

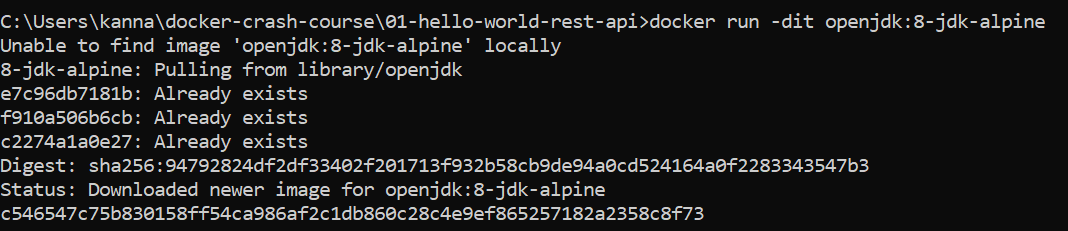
**Build a jar file**

In order build a jar file, right click on project and Run -> Maven Build, In goal field put “Clean Package”. Now a jar file will be generated and placed in target folder.

**Setup the prerequisites to run the jar file**

Now CD to the project folder and type the below command to run openjdk:8-jdk-alpine official image.

***docker run -dit openjdk:8-jdk-alpine***



Note: here we put -dit where d stands for detached mode

i interactive mode

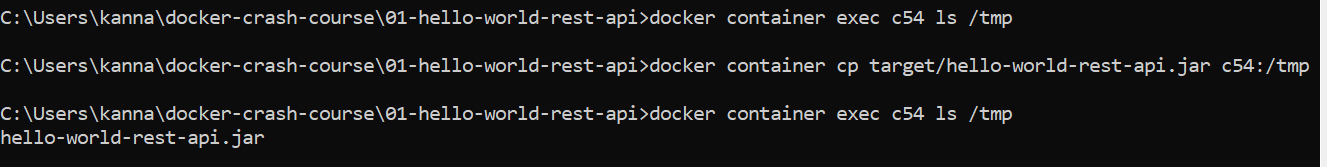
t -tty

the combination of -it is called interactive shell and it allows us to run few commands on running container.

***docker container exec containerName ls /temp***

Now we have to copy the jar file generated from original project to docker container using below command

***docker container cp target/ hello-world-rest-api.jar containerName:/temp***



Now we have to save the container with jar file as image using below command

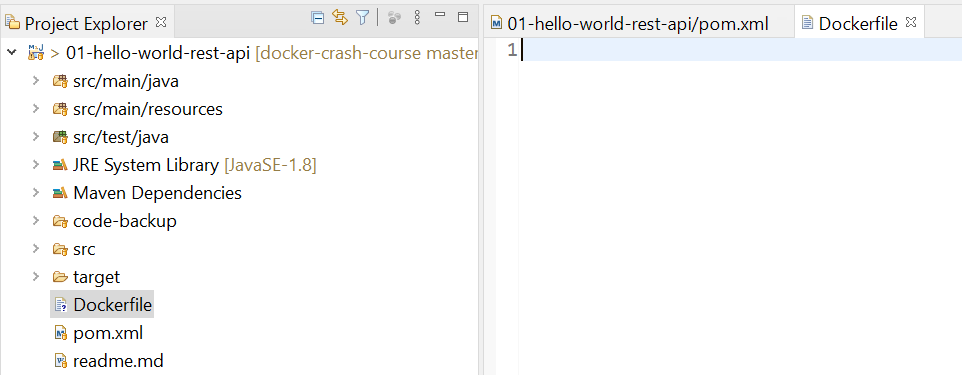
***docker container commit containerName in28min/todo-rest-api-h2:manual1***

Now if we run this image, it will not run because we haven’t specified that the jar has to be launched at startup. For that we have to use change command

***docker container commit --change=”CMD [\“java\”,\”-jar\”,\”hello-world-rest-api.jar\”]” containerName in28min/todo-rest-api-h2:manual1***

**Docker image creation using Dockerfile**

Create a Dockerfile without any extension under project folder.

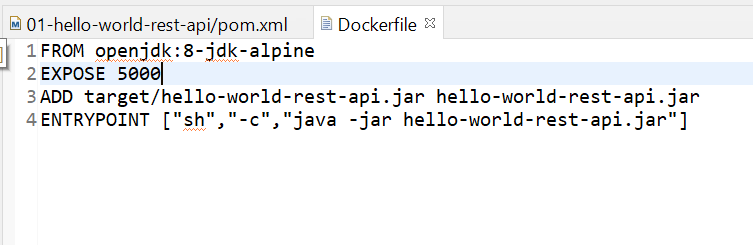


FROM openjdk:8-jdk-alpine

EXPOSE 5000

ADD target/hello-world-rest-api.jar hello-world-rest-api.jar

ENTRYPOINT ["sh","-c","java -jar hello-world-rest-api.jar"]



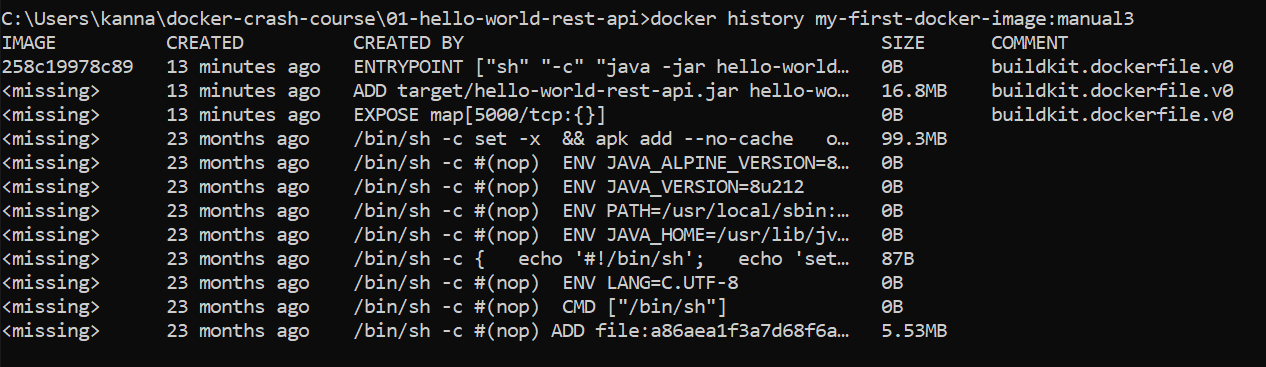
Now we have to build image using below command

***docker build -t my-first-docker-image:manual3***

**Image Layers, Caching and Dockerfile**

If we run the history command on an image that we created

***docker history my-first-docker-image:manual3***



Here we can notices the previous changes were done 23 months ago, basically that’s when any changes were added to jdk alpine image. Now 13 minutes ago we added a jar file, exposed a port and added entrypoint to run the jar file on startup.

Also Docker layers are cached. Say for example if we run the same build command with same tag name then it will mostly take from cache and only take modifications into effect.

**Spotify plugin to create docker image**

If we want to create images using just even easier, then we need to use maven plugins. One of the popular ones is spotify plugin.

* Don't do anything fancy. Dockerfiles are how you build Docker projects; that's what this plugin uses. They are mandatory.
* Make the Docker build process integrate with the Maven build process. If you bind the default phases, when you type mvn package, you get a Docker image. When you type mvn deploy, your image gets pushed.
* Make the goals remember what you are doing. You can type mvn dockerfile:build and later mvn dockerfile:tag and later mvn dockerfile:push without problems. This also eliminates the need for something like mvn dockerfile:build -DalsoPush; instead you can just say mvn dockerfile:build dockerfile:push.
* Integrate with the Maven build reactor. You can depend on the Docker image of one project in another project, and Maven will build the projects in the correct order. This is useful when you want to run integration tests involving multiple services.

Add the below plugin in maven plugins tags

<plugin>

<groupId>com.spotify</groupId>

<artifactId>dockerfile-maven-plugin</artifactId>

<version>1.4.10</version>

<executions>

<execution>

<id>default</id>

<goals>

<goal>build</goal>

</goals>

</execution>

</executions>

<configuration>

<repository>in28min/${project.name}</repository>

<tag>${project.version}</tag>

<skipDockerInfo>true</skipDockerInfo>

</configuration>

</plugin>

Now all we have to do to create image is Project right click -> Run as Maven Build -> Goal as ***package -DSkipTests***

Or just run the below command in cmd set to project folder

***mvn package -DSkipTests***

An image will be created automatically. Now we just have to run the new image like any other images in a container.

**Improving Caching of Docker Images by Adding Libraries in a Separate Step**

Have to see the udemy video again.

**JIB Maven plugin**

Jib is a [Maven](https://maven.apache.org/) plugin for building Docker and [OCI](https://github.com/opencontainers/image-spec) images for your Java applications. OCI is open containers initiative and its objective is to create and maintain the software shipping container image format spec. Docker is implementation of OCI specification.

With JIB plugin one thing is we don’t need Dockerfile. This will also takes care of adding libraries in a separate step by default.

<plugin>

<groupId>com.google.cloud.tools</groupId>

<artifactId>jib-maven-plugin</artifactId>

<version>1.6.1</version>

<configuration>

<container>

<creationTime>USE\_CURRENT\_TIMESTAMP</creationTime>

</container>

</configuration>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>dockerBuild</goal>

</goals>

</execution>

</executions>

</plugin>

<configuration>

<from>

<image>openjdk:alpine</image>

</from>

<to>

<image>in28min/${project.name}</image>

<tags>

<tag>${project.version}</tag>

<tag>latest</tag>

</tags>

</to>

<container>

<jvmFlags>

<jvmFlag>-Xms512m</jvmFlag>

</jvmFlags>

<mainClass>com.in28minutes.rest.webservices.restfulwebservices.RestfulWebServicesApplication</mainClass>

<ports>

<port>8100</port>

</ports>

</container>

</configuration>

By default jib doesn’t use openjdk:alpine image, so we have to use above configuration.

**Fabric8 Plugin**

This is little old plugin and uses Dockerfile or xml configuration below.

<plugin>

<groupId>io.fabric8</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>0.26.0</version>

<executions>

<execution>

<id>docker-build</id>

<phase>package</phase>

<goals>

<goal>build</goal>

</goals>

</execution>

</executions>

</plugin>

<properties>

...

<jar>${project.build.directory}/${project.build.finalName}.jar</jar>

</properties>

<plugin>

<groupId>io.fabric8</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>0.26.0</version>

<extensions>true</extensions>

<configuration>

<verbose>true</verbose>

<images>

<image>

<name>${project.artifactId}</name>

<build>

<from>java:8-jdk-alpine</from>

<entryPoint>

<exec>

<args>java</args>

<args>-jar</args>

<args>/maven/${project.build.finalName}.jar</args>

</exec>

</entryPoint>

<assembly>

<descriptorRef>artifact</descriptorRef>

</assembly>

</build>

</image>

</images>

</configuration>

<executions>

<execution>

<id>docker-build</id>

<phase>package</phase>

<goals>

<goal>build</goal>

</goals>

</execution>

</executions>

</plugin>

**Different commands in Dockerfile**

CMD – Default parameters. If you pass parameters from command line then your default

parameters will be overwritten. These params we pass when we run a container.

ENTRYPOINT: In this any parameters passed as arguments will not overwrite. (recommended)

COPY – this only allows copy from host computer to client. (recommended for simple copy)

ADD – this allows copy from host computer and also from a web url to client.

FROM tomcat:8.0.51-jre8-alpine

EXPOSE 8080

RUN rm -rf /usr/local/tomcat/webapps/\*

COPY target/\*.war /usr/local/tomcat/webapps/ROOT.war

CMD ["catalina.sh","run"]

/////////////////////////////////////////////////////////////////////////////////

FROM openjdk:8-jdk-alpine

EXPOSE 8080

ADD target/hello-world-rest-api.jar hello-world-rest-api.jar

ENTRYPOINT ["sh", "-c", "java -jar /hello-world-rest-api.jar"]

Pushing Images to docker hub

First create an account in docker hub. In Spotify plugin, specify the username in pom.xml



***docker login***

***docker push imageName:tagname***

Docker for mysql