B - Developing dockerized applications

Run these exercises on the **Docker** VM.

1. Install nodejs on the Docker VM

```
$ cd; mkdir example-nodejs; cd example-nodejs
```

First, install the node version manager, then use it to install nodejs itself.

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.38.0/install.sh | bash
$ source ~/.bashrc        # in order to set some NVM variables
$ nvm install v14.17.6    # latest LTS version based on 14
```

2. Develop a Nodejs app

Write the application code

Write a toy web server based on the Express framework. It listens at port 8080 and replies *Hello World* when HTTP GET requests are sent to /.

Create `server.js` with this code:

```
$ cat > server.js <<EOF
'use strict';

const express = require('express');

// Constants
const PORT = 8080;
const HOST = '0.0.0.0';

// App
const app = express();
app.get('/', (req, res) => {
  res.send('Hello World\n');
});

app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
EOF
```

You need to install Express before you can run it. This is done in the next step.

Create package.json

Create a file `package.json` with the content below. It describes the application and its dependencies. Here, we need `express`, the standard Nodejs web framework.

```
$ cat > package.json <<EOF
{
  "name": "Node-web-app",
  "version": "1.0.0",
  "description": "Demo app for showing Nodejs on Docker",
  "author": "Me Myself <me.myself@myself.com>",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.16.1"
  }
}
EOF
```

Install dependencies and run the app

The `npm install` command installs dependencies as described by `package.json`. It installs Express in a new directory `node_modules`, and creates a `package-lock.json` file with a detailed description of dependencies.

Ignore the warnings.

```
$ npm install
```

Launch the app.

```
$ node server.js
```

From another terminal, access the server with `curl localhost:8080`. The curl command should output the server's response *Hello Word*. Instead of curl, you can use a browser window to access the app.

Note that the app responds with an error page when you send requests to a URL other than /.

3. Containerize the app

So far the app is running as an uncontained process. The traditional way of putting it in a container is via a Dockerfile.

Create the Dockerfile

In the `$HOME/example-nodejs` directory, create `Dockerfile` with the following content:

```
FROM node:14
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD [ "node", "server.js" ]
```

A Dockerfile contains directives to create a container image. In this example, the `FROM` directive uses an existing image named **node** as the foundation; it is tagged with 14, the nodejs version number. This image will be retrieved from the Docker hub.

Feel free to read the image's documentation on the Docker hub at https://hub.docker.com/_/node.

`WORKDIR` declares the application directory in the container as `/usr/src/app`.

`COPY` copies the application's package files to the work directory.

`RUN` executes `npm install` to add dependencies to the container.

The second `COPY` copies the entire app code to the WORKDIR.

`EXPOSE` documents the port at which the app is listening.

`CMD` defines the command that runs by default when a container is started from this image.

Create .dockerignore to tell the build process to ignore some content

You want the application code, but not the `node_modules` directory to be copied to the image.

Create `~/example-nodejs/.dockerignore` with this content:

```
node_modules
```

```
npm-debug.log
```

This file tells the Docker build process to ignore these files and directories.

Build the image

Below, the `-t` option specifies the *build target*, i.e. the name of the image to be created.

Don't forget the dot; it tells docker in which directory the Dockerfile and the code are located.

```
$ sudo docker build . -t my/nodeapp
Sending build context to Docker daemon  20.48kB
Step 1/7 : FROM node:14
14: Pulling from library/node
...
---> 8cf18d365a37
Step 2/7 : WORKDIR /usr/src/app
 ---> Running in 882ad88c1dd3
Removing intermediate container 882ad88c1dd3
 ---> be17f84feedf
Step 3/7 : COPY package*.json ./
 ---> 133a2862b81a
Step 4/7 : RUN npm install
 ---> Running in 2f4a2ad52982
...
Removing intermediate container 2f4a2ad52982
 ---> 5d374427754a
Step 5/7 : COPY . .
 ---> b5753dfbec4f
Step 6/7 : EXPOSE 8080
 ---> Running in 6d3aff89b439
Removing intermediate container 6d3aff89b439
 ---> 101d21701427
Step 7/7 : CMD [ "node", "server.js" ]
 ---> Running in 7653fdb63a58
Removing intermediate container 7653fdb63a58
 ---> 14f1db307794
Successfully built 14f1db307794
Successfully tagged my/nodeapp:latest
```

Have a look at the output. Each step corresponds to a directive in the Dockerfile. Each arrow indicates a layer that is added to the original image. Some steps require running a command in a temporary container, for example the `RUN npm install`.

Inspect and use the image

List all images. At the minimum, you should find `node:14` and the newly created `my/nodeapp:latest`.

```
$ sudo docker image ls
```

Inspect the image and find the work directory, the exposed port 8080, and the CMD from the Dockerfile. What volume information can you find?

```
$ sudo docker image inspect my/nodeapp
```

Run a container from this image, publishing its port.

```
$ sudo docker run -d -p 8080 my/nodeapp
$ sudo docker container ls
```

Using the above syntax, port 8080 was mapped to an arbitrary port. The PORTS column might contain this:

```
    PORTS
    0.0.0.0:49153->8080/tcp
```

Also use `sudo docker port CONTAINER_ID` to get similar output.

In this example, the published port is 49135. You can either access the app with the container's IP address and port 8080, or use the published port on the localhost:

```
$ sudo docker inspect container_id | grep IPAdd
$ curl IP_ADDRESS:8080
Hello World
$ curl localhost:49153
Hello World
```

To access the container from the browser on your Desktop, you need to open the port in the firewall first: `sudo firewall-cmd --add-port 49153/tcp`.

Instead of the random port, map the container to a port of your choosing, for example 8000:

```
$ sudo docker run -d -p 8000:8080 my/nodeapp
```

4. Develop containerized code

Develop code with a Docker image

Modify the code and rebuild the image. Make a chance in server.js, for example:

```
app.get('/', (req, res) => {
  res.send('There is nothing here\n');
});

app.get('/hello', (req, res) => {
  res.send('Hello World\n');
});
```

Then build the image again.

```
$ sudo docker build . -t my/nodeapp
Sending build context to Docker daemon  20.48kB
Step 1/7 : FROM node:14
 ---> 8cf18d365a37
Step 2/7 : WORKDIR /usr/src/app
 ---> Using cache
 ---> c7f77af795a8
Step 3/7 : COPY package*.json ./
 ---> Using cache
 ---> 249ae1d046ab
Step 4/7 : RUN npm install
 ---> Using cache
 ---> 7b679628e1fa
Step 5/7 : COPY . .
 ---> b47b39e543b9
Step 6/7 : EXPOSE 8080
 ---> Running in b25b896ed1bf
Removing intermediate container b25b896ed1bf
 ---> 7e4695d56958
Step 7/7 : CMD [ "node", "server.js" ]
 ---> Running in de712e436979
Removing intermediate container de712e436979
 ---> c9621445d81a
Successfully built c9621445d81a
Successfully tagged my/nodeapp:latest
```

This is much faster than the first time. Obviously, the image is not downloaded, because it still exists locally. But steps 2, 3 and 4 are also not performed, since the corresponding image layers have not changed: the workdir is the same, `package.json` is unchanged, and the `node_modules` directory (generated by `npm install`) has not been modified either. Step 5 is the first step that needs to be redone, and as a consequence, all later steps as well.

```
$ sudo docker image ls
```

```
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
my/nodeapp   latest    c9621445d81a   6 minutes ago    947MB
<none>       <none>    a18339c25592   18 minutes ago   947MB
```

`<none>` is the old version of the image. The `my/nodeapp` tag was "stolen" by the new image, and the old image has no tag anymore.

Test your code by launching another container from the image and accessing it with curl or the web browser.

Develop code on the container

It's not efficient to generate a new Dockerfile for each change you want to test. Instead, modify the code *directly on the container*. Since containers don't normally have a development environment, make the code changes on the host and bind-mount the code on the container.

Launch a container from the newest image and bind-mount the current directory to the container's work directory.

```
$ sudo docker run -d -p 8080:8080 -v $PWD/mycode:/usr/src/app my/nodeapp
```

Make a modification in server.js, for example add this route:

```
app.get('/howdy', (req, res) => {
  res.send('Are you having a good time?\n');
});
```

or by changing some of the text that is displayed in the browser.

You also need to restart the node process. A Javascript developer would use `nodemon` to perform the restart automatically whenever the code changes, but for simplicity, stop and start the container.

```
$ sudo docker stop CONTAINER_ID
$ sudo docker start CONTAINER_ID
```

Launch a local registry

Once you are satisfied with your code, create a final image and copy it to your local registry.

First, launch a local registry.

```
$ sudo docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

By default, Docker expects registries to be secured with TLS. To simplify this exercise, configure the docker daemon to allow insecure registries. Create a file `/etc/docker/daemon.json` and enter the following text:

```
{
  "insecure-registries" : ["localhost:5000"]
}
```

then restart the daemon with `sudo systemctl restart docker`.

Store the image in your local registry

The command sudo docker push my/nodeapp would upload the image to the Docker Hub. To upload it to the local registry, add a tag to the image that includes the registry's URL.

```
$ sudo docker tag my/nodeapp localhost:5000/nodeapp
$ sudo docker image ls
REPOSITORY               TAG       IMAGE ID       CREATED            SIZE
my/nodeapp               latest    a47a191d15ad   48 seconds ago     947MB
localhost:5000/nodeapp   latest    a47a191d15ad   48 seconds ago     947MB
```

The image is listed twice with its two names. Upload it.

```
$ sudo docker push localhost:5000/nodeapp
Using default tag: latest
The push refers to repository [localhost:5000/nodeapp]
```

Since you are happy with your code, build the image again, giving it a version tag. Upload it to the local registry.

```
$ sudo docker build . -t my/nodapp:v2
$ sudo docker tag my/nodeapp:v2 localhost:5000/nodeapp:v2
$ sudo docker image ls
$ sudo docker push localhost:5000/nodeapp:v2
The push refers to repository [localhost:5000/nodeapp]
5648e49e1396: Pushed
66972b63fd9d: Layer already exists
824be929dd32: Layer already exists
...
```

Most image layers are not uploaded. The registry stores hash values of layers, which allows it to detect that a pushed layer exists on registry already.

Use skopeo and podman list registry images

Docker has no commands to manage images on registries apart from pull and push. It does have a registry API, and skopeo and podman use it to access image information on registries and copy images.

Skopeo and podman require different library versions than Docker. You will therefore use them on the Desktop instead of the `c8-docker` VM.

First, open port 5000 on `c8-docker`.

```
$ sudo firewall-cmd --add-port 5000/tcp
```

Open a terminal window, and use podman to list all images on the local registry.

```
$ podman search 192.168.122.21:5000/ --tls-verify=false
INDEX         NAME                          DESCRIPTION  STARS   OFFICIAL  AUTOMATED
122.21:5000   192.168.122.21:5000/nodeapp                0
```

Install skopeo and use it to list all tags of the `nodeapp` image on the local registry.

```
$ sudo dnf install skopeo
$ skopeo list-tags docker://192.168.122.21:5000/nodeapp --tls-verify=false
{
    "Repository": "192.168.122.21:5000/nodeapp",
    "Tags": [
        "latest",
        "v2"
    ]
}
```

Use skopeo and podman to get information about images and copy them between registries

```
$ skopeo inspect docker://docker.io/wordpress
```

Find the location of the `wordpress` image on the Docker Hub and copy it to the local registry.

By default, podman searches the registries listed in the `registries` setting in `/etc/containers/registries.conf`. Select an AMD64 image from this list generated by the command below, then copy it.

```
$ podman search wordpress
$ skopeo copy docker://docker.io/amd64/wordpress docker://192.168.122.21:5000/wordpress:latest --tls-
verify=false
```

Check if this was successful.

```
$ podman search 192.168.122.21:5000/ --tls-verify=false
INDEX         NAME                            DESCRIPTION  STARS   OFFICIAL  AUTOMATED
122.21:5000   192.168.122.21:5000/nodeapp                  0
122.21:5000   192.168.122.21:5000/wordpress                0
```

5. Use source-to-image to develop the app

Although s2i bundled in OpenShift uses podman, the standalone version of s2i requires Docker.
Run this exercise on the `c8-docker` VM.

The point of this exercise is not to explore the standalone version of s2i, but to understand the meaning of "source-to-image".

Install s2i

Visit https://github.com/openshift/source-to-image/releases/latest, scroll to the *Assets* section at the end and download the Linux AMD64 tarball to `c8-docker` (be careful not to download the Darwin tarball).

Unpack the tarball. It contains a single file named `s2i` as well as a symbolic link to it, named `sti`. Copy `s2i` to a directory in your PATH such as `/usr/local/bin`.

Log on to the Red Hat registry

The node image from the previous exercise is not enabled for source-to-image. While you can find s2i images on the Docker hub, this exercise is based on Red Hat's software collection images. Before you can pull them, you need to authenticate with the registry. Use your developer account. If you don't have one, you can obtain a free account at https://developers.redhat.com/register (it needs to be renewed every year).

```
$ sudo docker login registry.redhat.io
Username: YOURACCOUNTNAME
Password:YOURPASSWORD
Login Succeeded!
```

Get the s2i node image

You will use an s2i-enabled nodejs image based on Red Hat's Universal Base Image **ubi8**. For information about s2i-enabled nodejs, consult its page on the Red Hat registry site, or its Github repo (the information on the two sites is more or less identical).

```
$ sudo docker pull registry.redhat.io/ubi8/nodejs-14
```

Use s2i to build the app and test it

Copy the application's code and `package.json` to another directory. Do not copy the Dockerfile or the `node_modules` directory.

```
$ mkdir ~/s2i-nodejs
$ cd ~/example-nodejs
$ cp package.json server.js ~/s2i-nodejs
```

Use s2i to build an image.
You specify the directory that contains the source code, the s2i-enabled image, and the name of the image you will create. Since s2i needs to communicate with Docker, use sudo to launch the command.

```
$ cd ~/s2i-nodejs
$ sudo s2i build . registry.redhat.io/ubi8/nodejs-14 my/nodeapp-s2i
---> Installing application source ...
---> Installing all dependencies
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN Node-web-app@1.0.0 No repository field.
npm WARN Node-web-app@1.0.0 No license field.

added 50 packages from 37 contributors and audited 50 packages in 2.881s
found 0 vulnerabilities

---> Building in production mode
---> Pruning the development dependencies
npm WARN Node-web-app@1.0.0 No repository field.
npm WARN Node-web-app@1.0.0 No license field.

audited 50 packages in 0.744s
found 0 vulnerabilities
```

```
/opt/app-root/src/.npm is not a mountpoint
---> Cleaning the npm cache /opt/app-root/src/.npm
/tmp is not a mountpoint
---> Cleaning the /tmp/npm-*
Build completed successfully
```

You created an image based on two files: *server.js* containing the code, and *package.json* containing a description of the dependencies. There was no need to run `npm install` or to create a Dockerfile.

Test the app as before.

```
$ sudo docker run -d -p 8888:8080 --name s2i-app my/nodeapp-s2i
$ curl localhost:8888/hello
Hello World
```

C - Podman and Buildah

This exercise is based on https://www.redhat.com/en/blog/introduction-ubi-micro. Conduct it on the Desktop, where Podman and Buildah are installed.

You will get a feeling for buildah, which proposes a slightly different way of building images than the docker command, and will use podman to manage containers.

1. Install the micro UBI

Create an image based on Red Hat's micro version of the Universal Base Image. While buildah can process Dockerfiles, you will use the equivalent of Dockerfile commands on the command line. You need to be root, since buildah will have to perform a mount.

```
$ sudo -i
```

Start with ubi-micro as a foundation.

```
# buildah from registry.access.redhat.com/ubi8/ubi-micro
```

Use various commands to confirm that the UBI was downloaded.

```
# buildah images
# podman image ls
# podman images
```

The next command shows that buildah also created a **container** from the image. This is a so-called "working container". Its purpose is to allow building your own image based on the UBI.

```
# buildah containers
```

The container is named `ubi-micro-working-container`. You will need this name (or the container's ID) shortly.

Only buildah can see this container.

```
# podman container ls -a
```

2. Add a web server

Mount the container's filesystem

Your goal is to install a web server in the working container. However, the micro UBI does not contain a package manager, as confirmed by this command:

```
# buildah run ubi-micro-working-container dnf -y httpd
```

You will now **mount** the container's filesystem on the host and use the host's package management tool to install the web server on the container.

```
# buildah mount ubi-micro-working-container
```

The command outputs the mountpoint. Save it to a shell variable.

```
# M=MOUNTPOINT
# ls -l $M
# du -sh $M
# cat $M/etc/yum.repos.d/ubi.repo
```

Install Apache

The host runs Centos 8, the container runs RHEL 8 and uses RHEL 8 repos. To install software to the RHEL container, you need Red Hat's public keys for checking package signatures. While you can ignore signatures with yum's `--nogpgcheck` option, this is not good practice. Better, copy the keys from the container to the host's `/etc/pki/rpm-gpg` directory.

```
# ls $M/etc/pki/rpm-gpg
# ls /etc/pki/rpm-gpg
# cp $M/etc/pki/rpm-gpg/* /etc/pki/rpm-gpg
```

The following command installs httpd to the container. Don't answer 'y'.

```
# yum install --installroot $M httpd
```

Don't answer "y", but view what yum wants to install: 111 packages (152 MB) including 8 weak dependencies, which may be useful on a regular server, but are not really required in a container. Although it's not easily visible, yum also installs documentation, which the container doesn't need either.

Answer "n" and add options to exclude unnecessary files.

```
# yum install --installroot $M --setopt install_weak_deps=false --nodocs httpd
```

Now, only 92 packages (99MB) will be installed. Answer "y". At the end of the installation, remove unnecessary yum data.

```
# yum clean all --installroot $M
```

Unmount the working container.

```
# buildah unmount ubi-micro-working-container
```

Add a web page, define an entrypoint, test the image

Add the web site's index file and set the container's entrypoint. The two buildah commands correspond to the Dockerfile ADD and ENTRYPOINTS directives.

```
# echo '<h1>Success!</h1>' > index.html
# buildah add ubi-micro-working-container index.html /var/www/html
# buildah config --entrypoint "httpd -D FOREGROUND" ubi-micro-working-container
```

The `entrypoint` command generates a warning that you can ignore.

Turn the working container into an image.

```
# buildah commit ubi-micro-working-container my-http-image
```

Test it.

```
# podman run -d -p 8080:80 my-http-image
# curl localhost:8080
<h1>Success!</h1>
```

Upload the image to the local registry and use it

```
# podman tag my-http-image 192.168.122.21:5000/my/http-s2i:v1
# podman image ls
REPOSITORY                                TAG     IMAGE ID      CREATED             SIZE
localhost/my-http-image                   latest  0f56da1495ff  About a minute ago  158 MB
192.168.122.21:5000/my/http-s2i           v1      0f56da1495ff  About a minute ago  158 MB
# podman push 192.168.122.21:5000/my/http-s2i:v1 --tls-verify=false
```

Try it.

```
# podman run -d -p 8001:80 192.168.122.21:5000/my/http-s2i:v1
Trying to pull 192.168.122.21:5000/my/http-s2i:v1...
  Get "https://192.168.122.21:5000/v2/": http: server gave HTTP response to HTTPS client
Error: Error initializing source docker://192.168.122.21:5000/my/http-s2i:v1: error pinging docker
registry 192.168.122.21:5000: Get "https://192.168.122.21:5000/v2/": http: server gave HTTP response to
HTTPS client
```

The local registry does not use HTTPS, and you need to configure podman to accept this.

```
# vi /etc/containers/registries.conf
```

In this file, replace

```
[registries.insecure]
registries = []
```

with

```
[registries.insecure]
registries = ["192.168.122.21:5000"]
```

No need for root from this point.

```
# exit
$ podman run -d -p 8001:80 192.168.122.21:5000/my/http-s2i:v1
$ curl localhost:8001
<h1>Success!</h1>
```