Containers

Images Overview

In this lab you will get a chance to work with building images and using image registries. Registries are network based services for saving, looking up and retrieving container images. Docker Hub is the default registry for Docker based systems. Private registries can also be used to save and retrieve images within an organization. All modern public clouds also offer hosted registries for their clients. In the steps below you will run a private registry server and push/pull images to/from it. We will run the registry application itself in a container.

1. Docker Registry operations

Docker supplies several subcommands to interact with registries, including:

- login
- logout
- push
- pull

Setting up and running a simple service on a normal Linux system is often a daunting task. Even setting up a simple web server can take more time than you would like to spend; install packages, mess with config files, fix dependencies, test, change things some more, etc. Docker can reduce or even eliminate this repetitive work. Run the following command to start a container image registry server on your Docker host:

```
~$ docker container run -p 5000:5000 -d --name registry registry:2

Unable to find image 'registry:2' locally
2: Pulling from library/registry
c926b61bad3b: Pull complete
5501dced60f8: Pull complete
e875fe5e6b9c: Pull complete
21f4bf2f86f9: Pull complete
98513cca25bb: Pull complete
Digest:
sha256:0a182cb82c93939407967d6d71d6caf11dcef0e5689c6afe2d60518e3b34ab86
Status: Downloaded newer image for registry:2
8180d8e6c24dd40f4755a372eacb08d4bf8021abd79f9749a87a2004f4b71602

~$
```

List the containers on your system:

```
~$ docker container ls

CONTAINER ID   IMAGE        COMMAND                  CREATED
STATUS         PORTS                                      NAMES
8180d8e6c24d   registry:2   "/entrypoint.sh /etc…"   10 seconds ago   Up 9
seconds   0.0.0.0:5000->5000/tcp, :::5000->5000/tcp   registry

~$
```

One Docker command and seconds later you have a running image registry. The container almost always works because it brings its own filesystem, configuration and environment with it. The only things we need to configure when we run a container are the external aspects of the service, which programs outside the container will make use of, like the network port to use and so on.

Registry servers generally listen on TCP port 5000, however, at present, only software running on the Docker host can reach our container via their IP address. For this reason the Docker run command we used provided the `-p` switch to map the 5000 container port to the host so that we can contact the registry from the outside world using the host network interface. The `-d` switch runs the container detached (in the background).

Lookup the official “registry” repository on Docker Hub.

- How many images are available through the “registry” repository?
- How many tags are defined in the “registry” repository?
- Which tag identifies the newest image in the repository?
- Which image does the “latest” tag refer to?
- Which OS Distribution and Version are the “registry” images based on?

The registry service, like most Docker services, exposes its interface using a REST API. We can use the `curl` or `wget` tools to test our registry service:

```
~$ curl -i http://localhost:5000

HTTP/1.1 200 OK
Cache-Control: no-cache
Date: Thu, 18 Jan 2024 00:30:59 GMT
Content-Length: 0

~$
```

Our service is running and responds with "200 OK". Note that because we have mapped port 5000 on the host to port 5000 in the container, we do not need to know the IP address of the container on the container network.

2. Use the registry

Imagine we want to create a stable, repeatable environment to test some software in. We could create a single container image with all of our tools installed that we could use on our desktop, in our CI system and in development environments.

To illustrate we'll create a simple dev container with git installed. Run the following commands:

```
~$ docker container run -it --name client ubuntu:20.04 /bin/bash

Unable to find image 'ubuntu:20.04' locally
20.04: Pulling from library/ubuntu
06d39c85623a: Pull complete
Digest:
sha256:24a0df437301598d1a4b62ddf59fa0ed2969150d70d748c84225e6501e9c36b9
Status: Downloaded newer image for ubuntu:20.04

root@f36deaf920b6:/#
```

In the container:

```
root@f36deaf920b6:/# apt update

...

root@f36deaf920b6:/# apt install -y git

...

root@f36deaf920b6:/# exit

exit

~$
```

This is of course a trivial example, our container includes an Ubuntu 20.04 base and the Git version control system. In a real scenario we might install many other tools (valgrind, cmake, g++, java, maven, ant, scala, Play, ruby, Rails, python 2/3, Sinatra, etc.)

Now that we have our example container prepared let's create an image from the container that we can launch over and over again. Execute the following `commit` subcommand to create an image from your container:

```
~$ docker container commit client mydev/devenv

sha256:e4f768d5906b6c4c70838faafe4886bbcdd121b349fc3bb1dbf32390827174b5

~$
```

Now display the images on your docker system with the `image ls` subcommand:

```
~$ docker image ls

REPOSITORY     TAG       IMAGE ID       CREATED          SIZE
mydev/devenv   latest    e4f768d5906b   13 seconds ago   223MB
ubuntu         20.04     f78909c2b360   5 weeks ago      72.8MB
registry       2         909c3ff012b7   6 weeks ago      25.4MB

...

~$
```

We can push our devenv image to a registry to make it easy to access from any other Docker system on the network. Our image has a userId/repository formatted name. Pushing such a repository image will cause it to be sent to the Docker Hub. To push the image to a different repository we need to change the first part of the repository string to represent the URL for the local repository. We can create multiple names for a single image ID.

Use the `tag` subcommand to add a second name for the image created above and display the results:

```
~$ docker image tag mydev/devenv localhost:5000/devenv

~$ docker image ls | grep devenv

mydev/devenv            latest    e4f768d5906b   39 seconds ago   223MB
localhost:5000/devenv   latest    e4f768d5906b   39 seconds ago   223MB

~$
```

Note that the image IDs are identical; the tag is simply a pointer to the same set of image bits on disk identified by the image ID.

Use the following command to push the image to your local registry:

```
~$ docker image push localhost:5000/devenv

Using default tag: latest
The push refers to repository [localhost:5000/devenv]
5500e19d29c0: Pushed
3a03f09d2129: Pushed
latest: digest:
sha256:024337eb3ac3e0f07a91b562ca2254caa52f20b3b5c558b9d8289f2b05c1ed8f
size: 741

~$
```

When docker is asked to push the image it examines the image name to identify the registry to connect to, in our case, localhost:5000.

We can confirm the registry has our image using curl:

```
~$ curl -X GET localhost:5000/v2/_catalog

{"repositories":["devenv"]}

~$
```

3. Pull an image from the registry

To fully test our registry we should try to pull images from it. Remove the two image names you created above:

```
~$ docker image rm mydev/devenv

Untagged: mydev/devenv:latest

~$ docker image rm localhost:5000/devenv

Untagged: localhost:5000/devenv:latest
Untagged:
localhost:5000/devenv@sha256:024337eb3ac3e0f07a91b562ca2254caa52f20b3b5c55
8b9d8289f2b05c1ed8f
Deleted:
sha256:e4f768d5906b6c4c70838faafe4886bbcdd121b349fc3bb1dbf32390827174b5
Deleted:
sha256:dca5ee06f6253cdd2ab94f461e4a55698dea01cbc2c0f0597fbfc0311ca83288

~$
```

Because both tags referred to the same image only the second `image rm` subcommand actually deleted the image. In the example above two image layers are deleted. Under certain circumstances Docker creates unnamed intermediate images. When you delete a named image, Docker automatically removes any unused unnamed intermediate images in the ancestry.

Run a `docker image ls` command to be sure the images have been removed, then try to pull your image back down to the Docker host from the registry:

```
~$ docker image ls

REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
ubuntu       20.04     1c5c8d0b973a   2 weeks ago   72.8MB
nginx        latest    904b8cb13b93   3 weeks ago   142MB
registry     2         0d153fadf70b   5 weeks ago   24.2MB
```

```
...

~$
```

Now let's try to retrieve out image from the registry server:

```
~$ docker image pull localhost:5000/devenv

Using default tag: latest
latest: Pulling from devenv
521f275cc58b: Already exists
06a14fa56863: Pull complete
Digest:
sha256:024337eb3ac3e0f07a91b562ca2254caa52f20b3b5c558b9d8289f2b05c1ed8f
Status: Downloaded newer image for localhost:5000/devenv:latest
localhost:5000/devenv:latest

~$
```

Success! Note the image layer(s) in the output above state `Already exists` even though we deleted our two `devenv` images, why do you think that is? (Hint: what base image is the devenv built on? Is it still cached on your Docker host?)

List your local images to verify the download:

```
~$ docker image ls

REPOSITORY              TAG       IMAGE ID       CREATED              SIZE
localhost:5000/devenv   latest    e4f768d5906b   2 minutes ago   223MB
ubuntu                  20.04     1c5c8d0b973a   2 weeks ago
72.8MB
nginx                   latest    904b8cb13b93   3 weeks ago
142MB
registry                2         0d153fadf70b   5 weeks ago
24.2MB

...

~$
```

- Is the image you downloaded from the registry the same exact set of bits that you had before?
    - What makes you think so or not?

4. Dockerfiles

So far we have run a Docker container *from* an image and used the commit command to snapshot a container *into* an image.

There is a way to build images automatically by reading the instructions in a file known as a Dockerfile. A Dockerfile is a text file that contains all of the commands you would normally execute manually to configure an application artifact. Scripting the image build makes it repeatable, allows it to be committed to source code control, audited, reverted, etc.

In this lab step we'll begin by creating a "build" container image that builds and runs a simple Go Language "hello" service. The build image will include the Go compiler and linker, all of the source code for our application and the libraries it uses, all of the Go standard libraries and a lot of other bits and pieces.

Later we will create a second image to release to production with only the necessities required to run our service.

4.1. Create a basic Docker image

Create a new directory for your Dockerfile project:

```
~$ mkdir ~/goapp && cd $_

~/goapp$
```

We will create a simple "hello world" microservice in Go to package into our container image. The program listens on port 8080 and responds to requests on the root route with a "hello world" message.

First create the source file:

```
~/goapp$ nano hello.go && cat $_
```

```
package main

import (
    "fmt"
    "net/http"
)

func helloHandler(w http.ResponseWriter, r *http.Request) {
    response := "Hello World!"
    fmt.Fprintln(w, response)
    fmt.Println("Processing hello request.")
}

func listenAndServe(port string) {
    fmt.Printf("Listening on port %s\n", port)
    err := http.ListenAndServe(":"+port, nil)
    if err != nil {
        panic("ListenAndServe: " + err.Error())
    }
}
```

```
func main() {
    http.HandleFunc("/", helloHandler)
    port := "8080"
    go listenAndServe(port)
    select {}
}
```

```
~/goapp$
```

Now create a Dockerfile to build an image using the code above:

- We will use a preexisting image that has Golang already in it (`FROM golang:1.20`)
- Copy the source file into the image using the Dockerfile COPY command (`COPY ...`)
- Run the program (`CMD ["go","run","hello.go"]`)

```
~/goapp$ nano Dockerfile && cat $_
```

```
FROM docker.io/golang:1.20
WORKDIR /go/src/hello
COPY ./hello.go /go/src/hello
RUN go mod init example.com
CMD ["go","run","hello.go"]
```

```
~/goapp$
```

We use the `image build` subcommand to create the image using our Dockerfile:

```
~/goapp$ docker image build -t hello-build:v1 .

[+] Building 34.3s (9/9) FINISHED
docker:default
 => [internal] load build definition from Dockerfile
0.0s
 => => transferring dockerfile: 174B
0.0s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/golang:1.20
2.4s
```

```
 => [1/4] FROM
docker.io/library/golang:1.20@sha256:bfc60723228b88180b1e15872eb435cf7e6d8
199eae9be77c8dfd  28.6s
 => => resolve
docker.io/library/golang:1.20@sha256:bfc60723228b88180b1e15872eb435cf7e6d8
199eae9be77c8dfd8  0.0s
 => =>
sha256:30d85599795460b2d9d24c6b87c53ec60555b601705cc83bea31632240500980
64.14MB / 64.14MB           2.4s
 => =>
sha256:bfc60723228b88180b1e15872eb435cf7e6d8199eae9be77c8dfd8f8079343df
2.36kB / 2.36kB             0.0s
 => =>
sha256:0812e50d2e826e2feb01f7e5ef2a08978fc3acd398141d7940bfd7ef066347fd
1.58kB / 1.58kB             0.0s
 => =>
sha256:69b75e52a296dce3b9e8a8361bb7418b3e8744c46edd59be58384a33022ac533
6.86kB / 6.86kB             0.0s
 => =>
sha256:1b13d4e1a46e5e969702ec92b7c787c1b6891bff7c21ad378ff6dbc9e751d5d4
49.56MB / 49.56MB           1.6s
 => =>
sha256:1c74526957fc2157e8b0989072dc99b9582b398c12d1dcd40270fd76231bab0c
24.05MB / 24.05MB           1.1s
 => =>
sha256:7d90f81b68902900900a88202c27c541efbe650a06472cb0151c54917620eb6e
92.37MB / 92.37MB           3.6s
 => => extracting
sha256:1b13d4e1a46e5e969702ec92b7c787c1b6891bff7c21ad378ff6dbc9e751d5d4
4.4s
 => =>
sha256:01aa0e3863b47c0af24d0056fd011b9a6a25d1f4e7b4f5839b9635a4f61d90a7
100.52MB / 100.52MB         5.1s
 => =>
sha256:b84861d468c4174be6db8f75a28ed02188abc5e4ebab42fe75c17c2c00e4a72e
155B / 155B                 3.6s
 => => extracting
sha256:1c74526957fc2157e8b0989072dc99b9582b398c12d1dcd40270fd76231bab0c
1.1s
 => => extracting
sha256:30d85599795460b2d9d24c6b87c53ec60555b601705cc83bea31632240500980
4.9s
 => => extracting
sha256:7d90f81b68902900900a88202c27c541efbe650a06472cb0151c54917620eb6e
4.6s
 => => extracting
sha256:01aa0e3863b47c0af24d0056fd011b9a6a25d1f4e7b4f5839b9635a4f61d90a7
7.9s
 => => extracting
sha256:b84861d468c4174be6db8f75a28ed02188abc5e4ebab42fe75c17c2c00e4a72e
0.0s
 => [internal] load build context
0.0s
 => => transferring context: 533B
```

```
0.0s
 => [2/4] WORKDIR /go/src/hello
2.6s
 => [3/4] COPY ./hello.go /go/src/hello
0.0s
 => [4/4] RUN go mod init example.com
0.4s
 => exporting to image
0.1s
 => => exporting layers
0.1s
 => => writing image
sha256:eff67c22e949ecdbeca754fb78471947e6ede4a53f986684b6e89d117176ef07
0.0s
 => => naming to docker.io/library/hello-build:v1
0.0s

~/goapp$
```

Run the container so we can see that it works:

```
~/goapp$ docker container run -d --name mybuild hello-build:v1

9518f4f8fbe23952e7f4a774ef6eecf3f13a24b3f41519ebda7a83f6beabbcf6

~/goapp$
```

To see our container running, we can use `docker container ls`:

```
~/goapp$ docker container ls

CONTAINER ID   IMAGE            COMMAND                  CREATED
STATUS         PORTS                                     NAMES
9518f4f8fbe2   hello-build:v1   "go run hello.go"        9 seconds ago
Up 8 seconds                                             mybuild
8180d8e6c24d   registry:2       "/entrypoint.sh /etc…"   5 minutes ago
Up 5 minutes   0.0.0.0:5000->5000/tcp, :::5000->5000/tcp   registry

~/goapp$
```

When we're running many containers, the `ls` output can get difficult to read, but because our container is based on a particular image "`hello-build:v1`", we can filter (`--filter`) using the "ancestor" filter so that Docker shows us containers based on our image only:

```
~/goapp$ docker container ls --filter ancestor=hello-build:v1

CONTAINER ID   IMAGE            COMMAND             CREATED
```

```
STATUS          PORTS     NAMES
c06af59cc070   hello-build:v1   "go run hello.go"   14 seconds ago   Up 14
seconds             mybuild

~/goapp$
```

To discover information about our container we can examine its metadata using the `inspect` subcommand; `inspect` can tell us things like its IP address, environment variables, process ID, etc.

```
~/goapp$ docker container inspect mybuild | head -20

[
    {
        "Id":
"9518f4f8fbe23952e7f4a774ef6eecf3f13a24b3f41519ebda7a83f6beabbcf6",
        "Created": "2024-01-18T00:36:12.628795003Z",
        "Path": "go",
        "Args": [
            "run",
            "hello.go"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 20490,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2024-01-18T00:36:13.047899829Z",

~/goapp$
```

This too is *a lot* of information to sort through; fortunately the `inspect` hierarchy can be parsed using the go template language! By using what we know about the manifest's hierarchy, we can extract only the information we need with the `--format` argument.

Examine your container's inspect hierarchy and try a few queries; *replace the example ID below with the ID of your own container!*

```
~/goapp$ docker container inspect mybuild --format '{{.State.Pid}}'

20490

~/goapp$ docker container inspect mybuild --format '{{.Id}}'

9518f4f8fbe23952e7f4a774ef6eecf3f13a24b3f41519ebda7a83f6beabbcf6
```

```
~/goapp$ docker container inspect mybuild --format '{{.Config.Cmd}}'

[go run hello.go]

~/goapp$
```

Combining the previous `ls` and `inspect` commands we can:

- Target the container image with: `docker container ls --filter ancestor=hello-build:v1` in a nested shell, adding the "quiet" flag (`-q`) so that only the short ID is returned (try it on its own to see what it returns)
- Use `docker container inspect --format '{{.NetworkSettings.IPAddress}}'` to reveal your hello-build container's IP
- Assign the final output (the container's IP address) to an environment variable so that we can reuse it

Try it:

```
~/goapp$ IP=$(docker container inspect --format
'{{.NetworkSettings.IPAddress}}' mybuild) && echo $IP

172.17.0.3

~/goapp$
```

Now run a client container to test the service; adding the `--rm` flag below deletes the container when the command completes since we only need to use our client container temporarily:

```
~/goapp$ docker container run --rm busybox wget -qO- $IP:8080 && echo

Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
a307d6ecc620: Pull complete
Digest:
sha256:ba76950ac9eaa407512c9d859cea48114eeff8a6f12ebaa5d32ce79d4a017dd8
Status: Downloaded newer image for busybox:latest
Hello World!

~/goapp$
```

The app is successfully running but you may have noticed that it took more than a moment to retrieve the golang image.

If we check the size of our `hello-build` image, we notice it is quite sizeable:

```
~/goapp$ docker image ls hello-build

REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
hello-build   v1        eff67c22e949   2 minutes ago   846MB

~/goapp$
```

Let's fix that!

5. Multistage Builds

One of the most challenging things about building images is keeping the image size down. Each instruction in the Dockerfile adds a layer to the image, and it is important to clean up any artifacts not needed before moving on to the next layer/instruction.

It *was* common to have one Dockerfile to use for development and builds (which contained everything needed to build your application), and a slimmed-down one to use for production, which only contained your application and exactly what was needed to run it. This concept was known as the “builder pattern”.

With multi-stage builds, we can implement the builder pattern in a single Dockerfile. Each FROM instruction can use a different base, and each of them begins a new stage of the build. We can selectively copy artifacts from one stage to another, leaving behind everything we don’t want in the final image.

A typical multistage microservice build will build the service in the first stage and then package it for distribution in the second stage. In this step we will use the Docker multi-stage build functionality to create a static binary in the first stage which uses the golang image. In the second stage we will extract the executable service from the first stage.

This final container will have a small footprint, making it quick to download and also minimizing its attack surface.

To begin create a new context:

```
~$ mkdir ~/multi-build && cd $_

~/multi-build$
```

Now that we have a new directory to work in, create a Dockerfile with the following features:

- Just as before, use the Golang image (`FROM golang:1.20`)
- Add source code to the image (`COPY ...`)
- Run a build this time (`RUN ["go","build","-tags","netgo"]`)

> N.B. The "-tags netgo" flag causes Go to statically link the executable, making it free of dependencies.

In the second image:

- Using a second `FROM` instruction, define the second build from scratch
- Copy the binary from the first build to the second image build `COPY --from=build-env`
- Expose ports where our app is listening
- Set the entrypoint of the image as our app

```
~/multi-build$ cp ~/goapp/hello.go .

~/multi-build$ nano Dockerfile && cat $_
```

```
FROM docker.io/golang:1.20 AS build-env
WORKDIR /go/src/hello/
COPY ./hello.go /go/src/hello/
RUN ["go","mod","init","example.com/hello"]
RUN ["go","build","-tags","netgo"]

FROM scratch
COPY --from=build-env /go/src/hello/hello hello
EXPOSE 8080
ENTRYPOINT ["./hello"]
```

```
~/multi-build$
```

Now build the image with the same repo name `hello-build` with a `v2` tag:

```
~/multi-build$ docker image build -t hello-build:v2 .

[+] Building 26.7s (11/11) FINISHED
docker:default
 => [internal] load build definition from Dockerfile
0.0s
 => => transferring dockerfile: 309B
0.0s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [internal] load metadata for docker.io/library/golang:1.20
0.6s
 => [build-env 1/5] FROM
docker.io/library/golang:1.20@sha256:bfc60723228b88180b1e15872eb435cf7e6d8
199eae9  0.0s
 => [internal] load build context
0.0s
 => => transferring context: 533B
0.0s
```

```
 => CACHED [build-env 2/5] WORKDIR /go/src/hello/
0.0s
 => [build-env 3/5] COPY ./hello.go /go/src/hello/
0.0s
 => [build-env 4/5] RUN ["go","mod","init","example.com/hello"]
0.4s
 => [build-env 5/5] RUN ["go","build","-tags","netgo"]
25.3s
 => [stage-1 1/1] COPY --from=build-env /go/src/hello/hello hello
0.1s
 => exporting to image
0.1s
 => => exporting layers
0.1s
 => => writing image
sha256:163ebaa35f52dc1a9e64b0e92dd7b9976286c773f1b8269a57725131f7af486f
0.0s
 => => naming to docker.io/library/hello-build:v2
0.0s

~/multi-build$
```

List our hello-build images:

```
~/multi-build$ docker image ls hello-build

REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
hello-build   v2        163ebaa35f52   2 minutes ago   6.63MB
hello-build   v1        eff67c22e949   5 minutes ago   846MB

~/multi-build$
```

Look at the difference; our v1 hello-build is just under 1GB but our v2 is under 10MB. Talk about improvement!

Run a new container from our new image:

```
~/multi-build$ docker container run -d hello-build:v2

7bd4ee8c9fc14dc1c7d91fa814c76e8af7b7333436da4df672484e79f57d84ef

~/multi-build$
```

Store its IP in another environment variable, *making sure you change the ancestor argument to filter for the v2 image this time*:

```

```

```
~/multi-build$ IP2=$(docker container inspect --format
'{{.NetworkSettings.IPAddress}}' \
$(docker container ls -q --filter ancestor=hello-build:v2)) && echo $IP2

172.17.0.4

~/multi-build$
```

Run another client container to test if it works:

```
~/multi-build$ docker container run --rm busybox wget -qO- $IP2:8080 &&
echo

Hello World!

~/multi-build$
```

Success! The app is the same, but we have eliminated all of the build tools from the image, making it more secure and lighter weight.

Use the `exec` command to take a look inside this lightweight container:

```
~/multi-build$ docker container exec -it $(docker container ls -q --filter
ancestor=hello-build:v2) /bin/sh

OCI runtime exec failed: exec failed: container_linux.go:380: starting
container process caused: exec: "/bin/sh": stat /bin/sh: no such file or
directory: unknown

~/multi-build$
```

What happened?

We didn't include anything in our app image filesystem other than the application binary itself. It is impossible to run `/bin/sh` or anything else other than our app! This is great for security but can make the container harder to debug.

6. CHALLENGE: add a Linux distro with a filesystem

Add a Linux distribution filesystem to the `hello-world:v2` image so that users can `exec` a sh in a running container. Build the image, tagging the new image version `hello-world:v3`; then run a container from the image and ensure you can exec a shell inside the container.

7. CHALLENGE: tagging images

Add an appropriate tag to your production go program image and push it to your local registry. Verify your registry has stored your image.

8. Clean up

Stop and remove all of the containers on your system.

```
~/multi-build$ docker container rm $(docker container stop $(docker
container ls -qa))

8cd929133fb8
7728345754b9
c06af59cc070
f36deaf920b6
7839f47563b5

~/multi-build$ cd

~$
```

Congratulations, you have completed the lab!