CSC 346 - Cloud Computing

02 - SSH & Creating Docker Images

Docker Images

Docker Images

There are a few ways to make our own images

- Download from a docker image repository
  - This is what we've done so far with `docker run` commands.
- Using `docker commit` to save changes from a container to a new image.
  - Run a container, make some changes, then 'save' the changes
- Using a Dockerfile and the `docker build` command.
- Using `docker tag` to basically 'clone' and image and give it a new name.
  - This is not really creating a new image, it's just the same image with a different name

Docker Images

`docker commit`

- I've mentioned that images are immutable, and if you exit your container you'll loose all your changes unless you take special steps.
- The **`docker commit`** command is one of those special steps.
- First, let's make some changes.

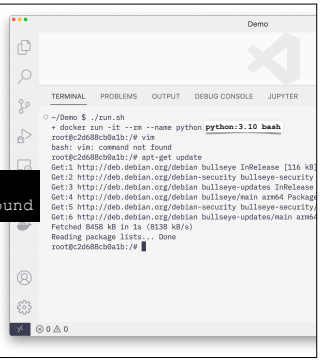
ubuntu

Installing software with `apt-get`

- Run our familiar python container
- See if the **`vim`** command exists

```
root@c2d688cb0a1b:/# vim
bash: vim: command not found
```

- Use apt-get update first to update the repository sources



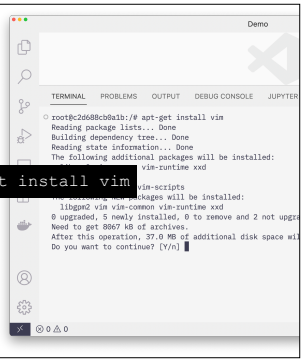
ubuntu

Installing software with `apt-get`

- Now use apt-get install to install **`vim`**

```
root@c2d688cb0a1b:/# apt-get install vim
```

- Type **`Y`** then enter to continue and install



ubuntu

Installing software with `apt-get`

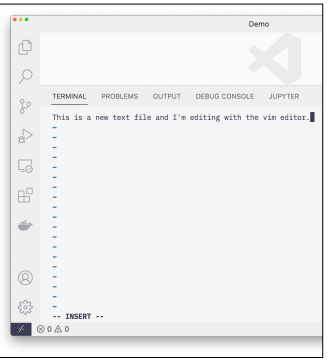
- After the installer finishes you can now use the vim command to create and edit text files.

```
root@c2d688cb0a1b:/# vim foo.txt
```

ubuntu

vim basics

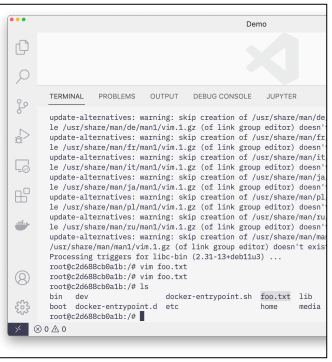
`i`	Enter insert mode
`esc`	Exit insert mode
`arrow keys`	Move the cursor around
`:w`	Save your changes (when not in insert mode)
`:wq`	Save your changes and exit vim (when not in insert mode)
`:q!`	Force quite vim and discard all changes (when not in insert mode)



ubuntu

Installing software with `apt-get`

- After saving changes and exiting vim the new file created is in our directory

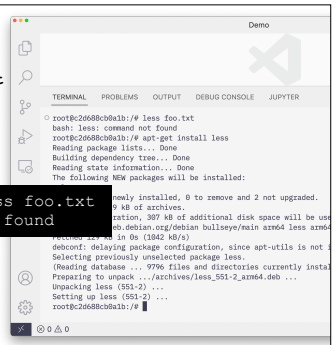


ubuntu

Installing software with `apt-get`

- **`less`** is also not installed in this container, let's install that too

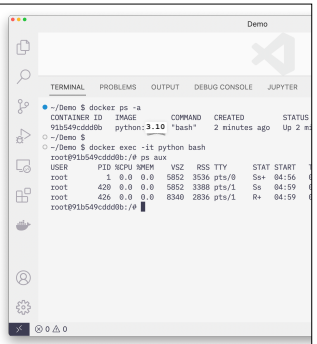
```
root@c2d688cb0a1b:/# less foo.txt
bash: less: command not found
```



Docker

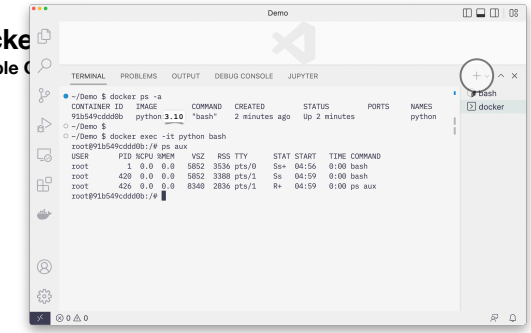
Multiple Container Connections

- When you use **`docker run -it`** you're creating a new container and making a shell connection to your container
- You can make more than one.
- You can use **`docker exec`** to run a command ***inside of an existing container that is running***.
  - Must be a running container



Docke

Multiple C

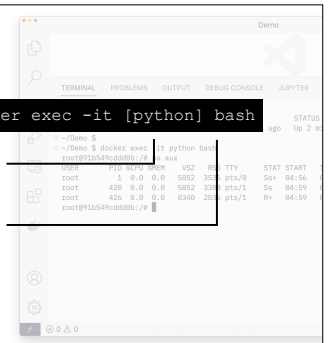


Docker

Multiple Container Connections

```
$ docker exec -it [python] bash
```

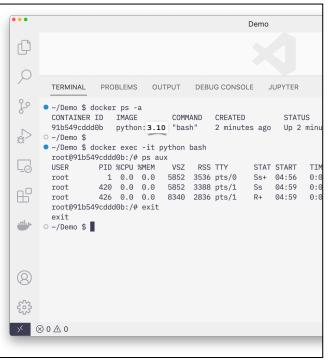
- You need to specify the name or ID of the running container
- You need to specify the command you want to execute in the new container
  - In most cases, you want a new bash shell



Docker

Multiple Container Connections

- You can exit from this second connection and it won't kill the container
  - There's still the first bash process running

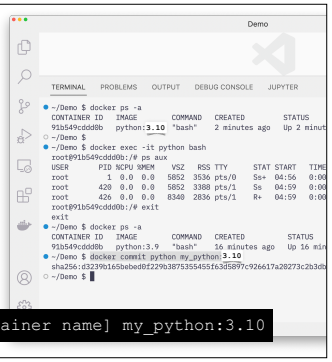


Docker

Saving with `docker commit`

- From the second terminal with the container still running we can use the **`docker commit`** command so save the current container to a new image.
- Container can be running or stopped.
- All '**`docker`** …' are run from outside of the container.

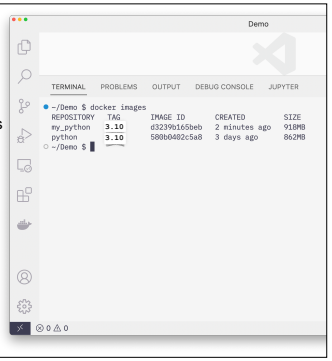
```
$ docker commit [container name] my_python:3.10
```



Docker

Saving with `docker commit`

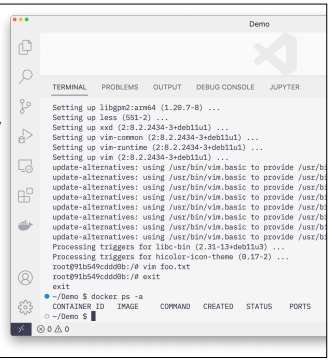
- Now you can use the docker images command to see our newly created image



Docker

Saving with `docker commit`

- With our image "saved" we can now finally exit our other bash session in the other terminal, and exit the container
- Remember we ran the container with the --rm option, so it will be removed upon exit



Docker

Saving with `docker commit`

- We can now run a new container based off of our new image

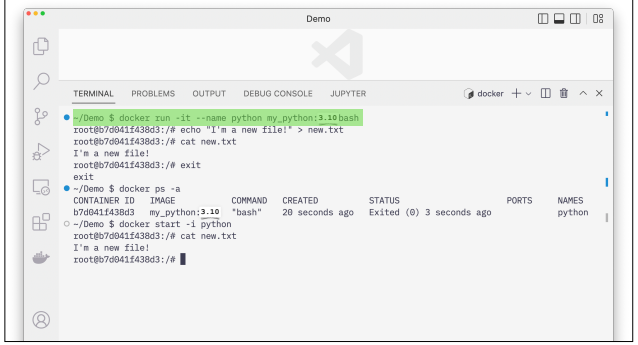
```
$ docker run -it my_python:3.10 bash
```

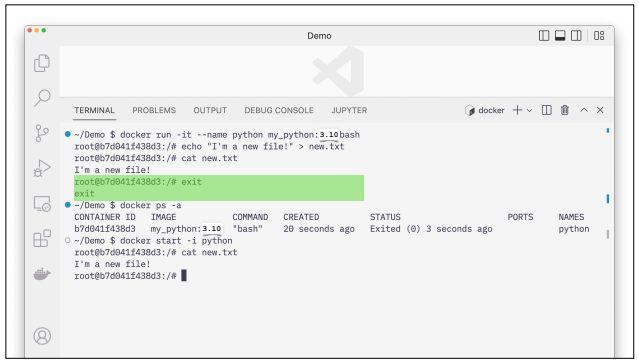
- Our `foo.txt` file is still there.

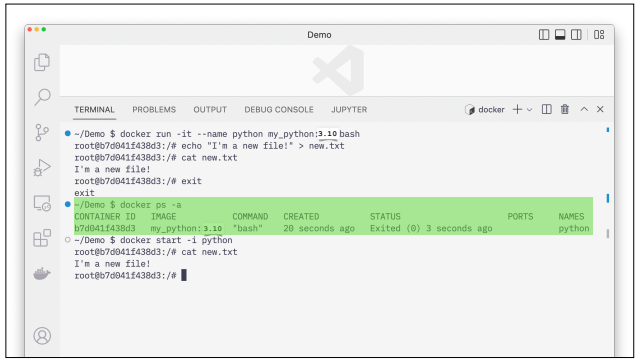
Docker

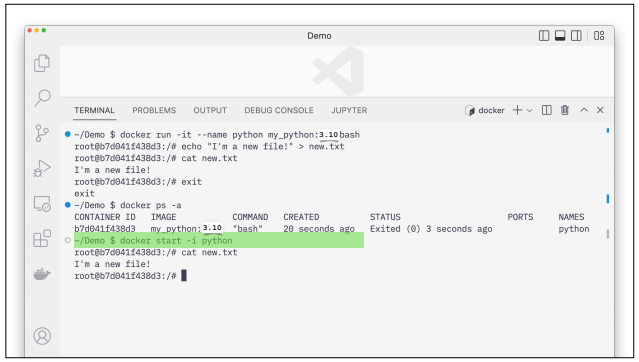
Stopping and Starting a container

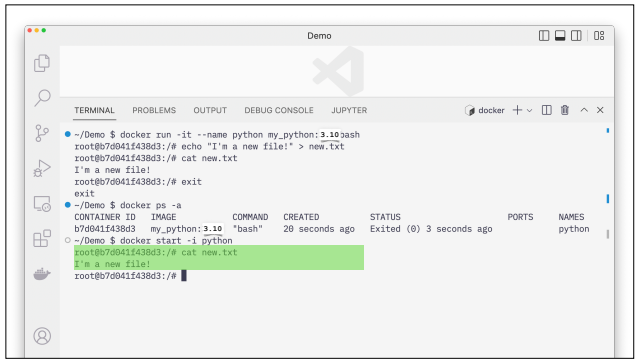
- You don't have to throw away your container when you exit
- Without the `--rm` option, when you exit the container, it remains in an exited state
- You can re-start this container
- This is fine for prototyping, but don't depend on that stopped container. It's easy to accidentally remove it.




Demo
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER
docker
~/Demo $ docker run -it --name python my_python:3.10 bash
root@b7d041f438d3:/# echo "I'm a new file!" > new.txt
root@b7d041f438d3:/# cat new.txt
I'm a new file!
root@b7d041f438d3:/# exit
exit
~/Demo $ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
b7d041f438d3 my_python:3.10 "bash" 20 seconds ago Exited (0) 3 seconds ago python
~/Demo $ docker start -i python
root@b7d041f438d3:/# cat new.txt
I'm a new file!
root@b7d041f438d3:/#


Demo
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER
docker
~/Demo $ docker run -it --name python my_python:3.10 bash
root@b7d041f438d3:/# echo "I'm a new file!" > new.txt
root@b7d041f438d3:/# cat new.txt
I'm a new file!
root@b7d041f438d3:/# exit
exit
~/Demo $ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
b7d041f438d3 my_python:3.10 "bash" 20 seconds ago Exited (0) 3 seconds ago python
~/Demo $ docker start -i python
root@b7d041f438d3:/# cat new.txt
I'm a new file!
root@b7d041f438d3:/#

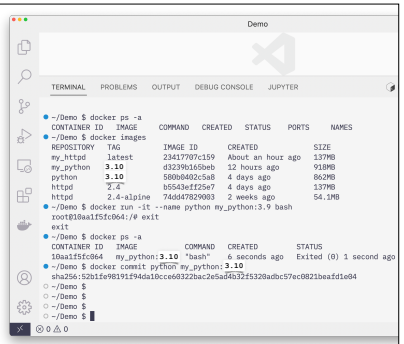

Demo
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE JUPYTER
docker
~/Demo $ docker run -it --name python my_python:3.10 bash
root@b7d041f438d3:/# echo "I'm a new file!" > new.txt
root@b7d041f438d3:/# cat new.txt
I'm a new file!
root@b7d041f438d3:/# exit
exit
~/Demo $ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
b7d041f438d3 my_python:3.10 "bash" 20 seconds ago Exited (0) 3 seconds ago python
~/Demo $ docker start -i python
root@b7d041f438d3:/# cat new.txt
I'm a new file!
root@b7d041f438d3:/#



Docker

Using docker commit on a stopped container

- You can also use **`docker commit`** on a stopped container that hasn't been removed yet
- You can either give this commit a new image name and tag, or you can overwrite an existing one



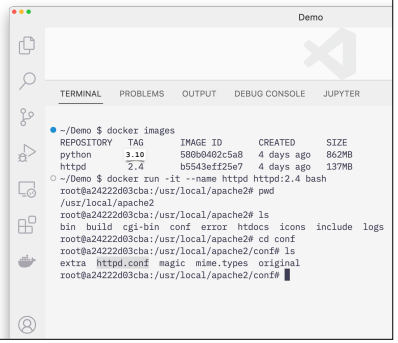
Docker

Moving files into and out of a container

- You can copy files into and out of a running or stopped container.
- Only works with ***containers***, not ***images***.
- Let's say we want to work with the apache web server image for **`httpd`**.
- If we want to modify the default config file from the image, it would be helpful to copy the default one out of the container and then change it.

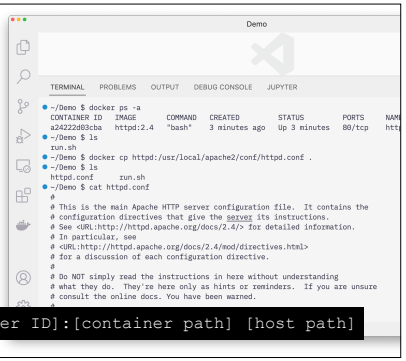
Docker
Copying Files

- Run a new container using the **`httpd:2.4`** image
- Look at the default directory we start in
- Change to the **`conf`** directory
- Look for the **`httpd.conf`** file



Docker
Copying Files

- Open a new Terminal
- use the **`docker cp`** command to copy from inside the container to the current directory
- The special "`.`" directory means "the directory I'm in"




```
docker cp [container ID]:[container path] [host path]
```

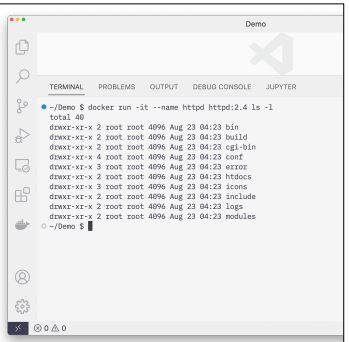
Docker
Copying Files

- This works the other way too. You can copy files from your host into a running or stopped container. Just reverse the order of the arguments

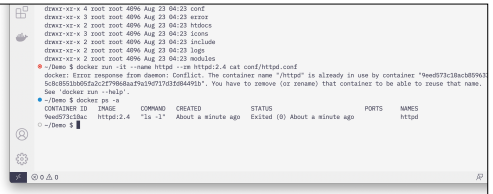
```
docker cp [host path] [container ID]:[container path]
```

Docker
Other Container Commands

- You don't have to just run a new bash shell inside of a container.
- We can just run the **`ls`** command
- Or just a **`cat`** command.



Docker
Run Errors



- Hmm what happened to cause our error?
- We tried to run a new container with a name of **`httpd`**, but we did not remove the first one
- You can't have two containers with the same name on a host at the same time

Docker
Other Container Commands

- After we remove the old image, you can run the command successfully.
- By including the **`--rm`** option we can make sure these ephemeral commands don't leave old exited containers around

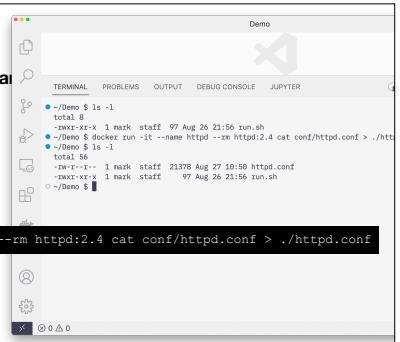
Docker

Other Container Commands

- On macOS, Linux, and Windows with **`wsl2`** setup, you can use redirection on the host to capture the output of your docker commands

```
docker run -it --name httpd --rm httpd:2.4 cat conf/httpd.conf > ./httpd.conf
```

- Gets us the same result as **`docker cp`** in a different way



Docker

Volume Mounting

- Copying files back and forth from a container is tedious
- Having to commit your changes to an image each time you're done is error prone
- We can avoid both of these problems by mounting a directory from your host computer inside the running container
- This is done with the **`-v`** or **`--volume`** option to the **`docker run`** command

```
--volume [host path]:[container path]
```

Docker

Volume Mounting

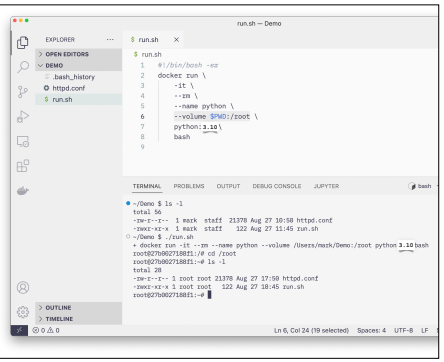
- The host path must be a full absolute path
  - Many times you want to mount your current directory, or something in it
  - Can use the **`$PWD`** environment variable on macOS, Linux, and WSL2
  - Can use the **`%cd%`** environment variable in PowerShell
- The following two commands are equivalent

```
docker run --volume $PWD:/root python:3.10
```

```
docker run --volume /Users/mark/Demo:/root python:3.10
```

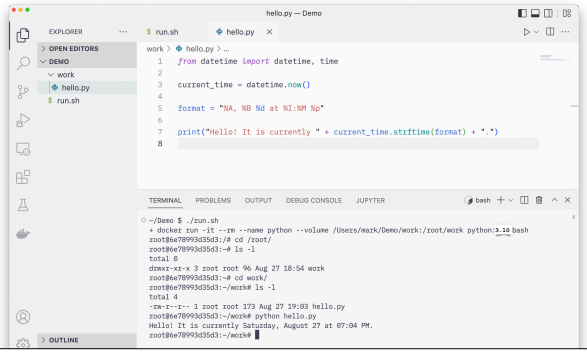
Docker
Volume Mounting

- Inside the /root directory in our container you can see the same files from our host.
- This is a live two way mapping. Changes are available in both places.



Docker
Volume Mounting

- This is really useful
- Lets us get files into a container without having to copy them each time
- Changes made inside the container to those files are reflected on the host
  - Note they're not copied, its the same file in both places. Filesystem magic!
- Changes made outside the container to the files are reflected inside the container
- Let's us work on the files in our GUI, but run them inside the container



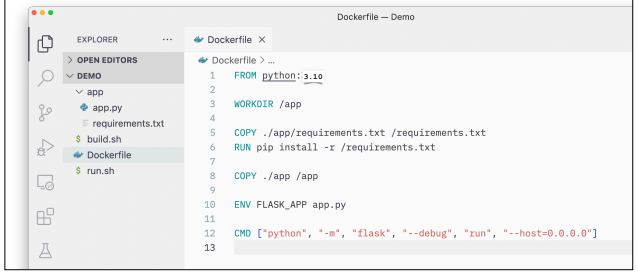
Demo

Creating Images
The Dockerfile

- Probably the most common ways we create Docker images for our projects are with a **`Dockerfile`** and the **`docker build`** command.

https://docs.docker.com/engine/reference/builder/

Creating Images
The Dockerfile



Creating Images

The Dockerfile

Dockerfile — Demo

EXPLORER

OPEN EDITORS

DEMO

- app
  - app.py
  - requirements.txt
- build.sh
- Dockerfile
- run.sh

Dockerfile > ...

```
FROM python:3.10

WORKDIR /app

COPY ./app/requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

COPY ./app /app

ENV FLASK_APP app.py

CMD ["python", "-m", "flask", "--debug", "run", "--host=0.0.0.0"]
```

The FROM keyword specifies which base image we're building on top of.

Creating Images

The Dockerfile

Dockerfile — Demo

EXPLORER

OPEN EDITORS

DEMO

- app
  - app.py
  - requirements.txt
- build.sh
- Dockerfile
- run.sh

Dockerfile > ...

```
FROM python:3.10

WORKDIR /app

COPY ./app/requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

COPY ./app /app

ENV FLASK_APP app.py

CMD ["python", "-m", "flask", "--debug", "run", "--host=0.0.0.0"]
```

The WORKDIR keyword sets the default directory for the container

Creating Images

The Dockerfile

Dockerfile — Demo

EXPLORER

OPEN EDITORS

DEMO

- app
  - app.py
  - requirements.txt
- build.sh
- Dockerfile
- run.sh

Dockerfile > ...

```
FROM python:3.10

WORKDIR /app

COPY ./app/requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

COPY ./app /app

ENV FLASK_APP app.py

CMD ["python", "-m", "flask", "--debug", "run", "--host=0.0.0.0"]
```

The COPY keyword lets us copy a files from our working directory to inside the image we're building

Creating Images

The Dockerfile

Dockerfile — Demo

EXPLORER

OPEN EDITORS

DEMO

- app
  - app.py
  - requirements.txt
- build.sh
- Dockerfile
- run.sh

Dockerfile > ...

```
FROM python:3.10

WORKDIR /app

COPY ./app/requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

COPY ./app /app

ENV FLASK_APP app.py

CMD ["python", "-m", "flask", "--debug", "run", "--host=0.0.0.0"]
```

The RUN keyword executes the rest of the line as a command inside the build environment

Creating Images

The Dockerfile

Dockerfile — Demo

EXPLORER

OPEN EDITORS

DEMO

- app
  - app.py
  - requirements.txt
- build.sh
- Dockerfile
- run.sh

Dockerfile > ...

```
FROM python:3.10

WORKDIR /app

COPY ./app/require
RUN pip install -r

COPY ./app /app

ENV FLASK_APP app.py

CMD ["python", "-m", "flask", "--debug", "run", "--host=0.0.0.0"]
```

COPY our application files into the image

Creating Images

The Dockerfile

Dockerfile — Demo

EXPLORER

OPEN EDITORS

DEMO

- app
  - app.py
  - requirements.txt
- build.sh
- Dockerfile
- run.sh

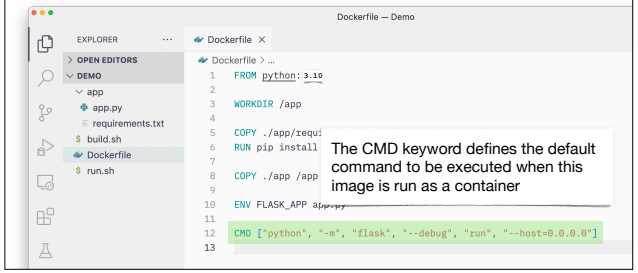
Dockerfile > ...

```
FROM python:3.10

WORKDIR /app

COPY ./app/require
RUN pip install -r

COPY ./app /app

ENV FLASK_APP app.py

CMD ["python", "-m", "flask", "--debug", "run", "--host=0.0.0.0"]
```

The ENV keyword defines an environment variable which will be accessible to the running container

Creating Images

The Dockerfile



Creating Images

The Dockerfile

- The **`docker build`** command is what turns our Dockerfile into an image

```
docker build --tag [image name]:[tag] [location]
```

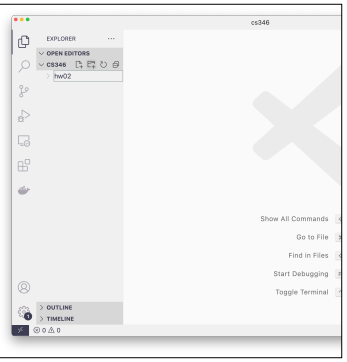
```
docker build --tag my_app:latest .
```

```
https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
```

Creating Images

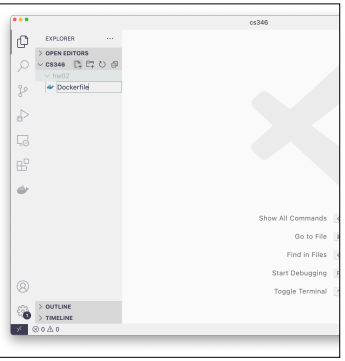
Getting Started Basics

- When working on any project, such as an application or homework assignment, the first step is often to create a new directory to hold all the stuff relating to the project.
- So to start with, figure out where on your laptop you want to keep all your work for this class, and make a new folder in there. I'm going to call mine **`hw02`**.



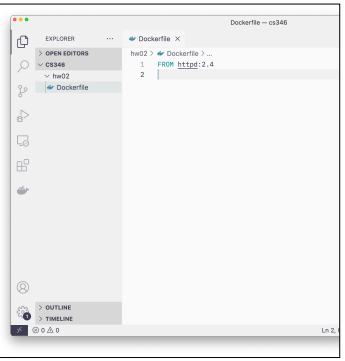
Creating Images
Getting Started Basics

- Next we need to create a new empty text file inside our project folder, and name it **`Dockerfile`**



Creating Images
Getting Started Basics

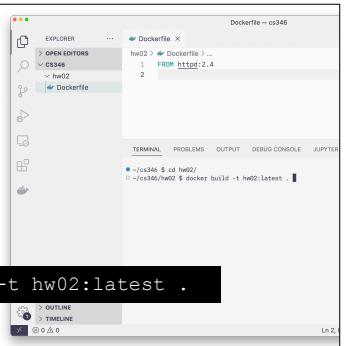
- With our newly created Dockerfile open in an editor, we can start with the most basic directive, and just have a **`FROM httpd:2.4`** line in our file.



Creating Images
Getting Started Basics

- Don't forget to **`save`** your **`Dockefile`** before you build it!
- Make sure your terminal session is currently in your project folder.
- Build our new image with the **`docker build`** command:

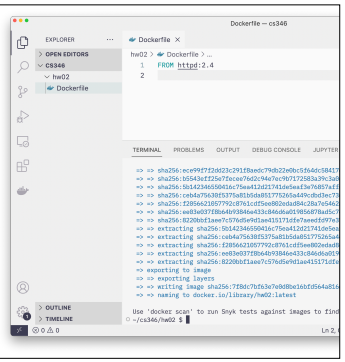
```
docker build -t hw02:latest .
```



Creating Images

Getting Started Basics

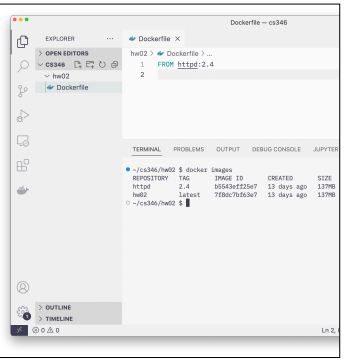
- If your image builds successfully, you won't see any errors, and you'll be returned to your laptop's command prompt.



Creating Images

Getting Started Basics

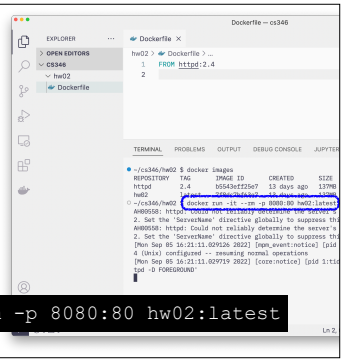
- You can see your newly created image with the **`docker images`** command on your laptop.
- You may see more or fewer images depending on when you last pruned your docker system.



Creating Images

Getting Started Basics

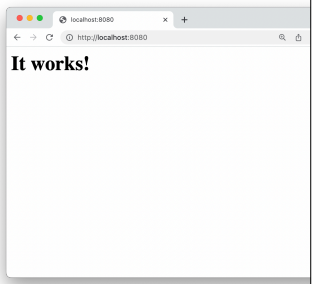
- We can now run our basic image to make sure everything is working so far.
- Because this container's purpose is to run a web server, we need to make sure to map our host and container ports.




```
docker run -it --rm -p 8080:80 hw02:latest
```

Creating Images
Getting Started Basics

- If everything worked out, you should be able to open a new browser tab and go to **`http://localhost:8080`** and see the default web page served up by the **`httpd:2.4`** container.

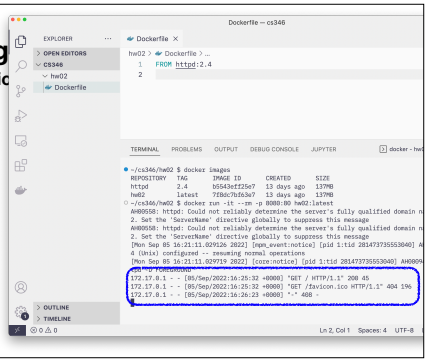


Creating Images
Getting Started Basics

- Remember, everything we did here was done ***from the host computer*** (i.e. your laptop). We aren't building or running anything from *inside* of a container.
- With the exception of certain automated build environments you'll likely never run any **`docker`** … command from *inside* of a container.

Creating Imag
Getting Started Basic

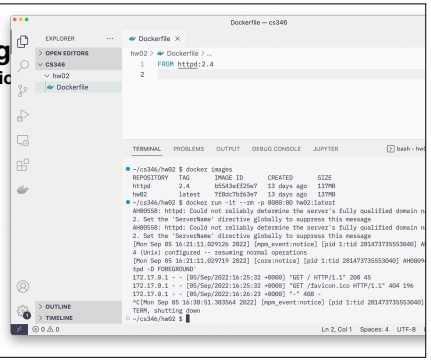
- You can see the logs from the web server in your terminal window
- This shows you exactly what your browser requested from the web server



Creating Imag

Getting Started Basic

- To exit the container, press the **`control`** and **`c`** key together.
- This is often abbreviated as just **`ctrl-c`** or **`^C`**
- You can see the **`^C`** in the screenshot before the shutdown line



Demo

SSH Basics

Connecting to Remote Hosts

`ssh` - The Secure Shell

- "Back in my day" we connected to remote unix hosts with the **`telnet`** command
  - Plain text network traffic
  - No encryption
  - It's horribly insecure!
- Can still be useful, but is often not installed by default anymore
  - Did I mention it's *horribly insecure*?

Connecting to Remote Hosts

`ssh` - The Secure Shell

- The ssh program is better
  - End-to-end encryption
  - Can use passwords or public keys
  - ssh + public keys is very secure

```
ssh [username]@[hostname]
```

```
ssh [username]@[IP Address]
```

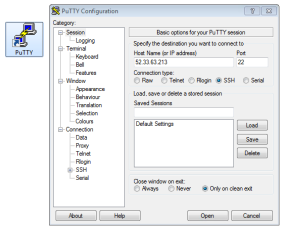
Connecting to Remote Hosts

`ssh` - The Secure Shell

- The ssh program is installed by default on macOS, Linux desktops, recent version of Windows, and the Windows Subsystem for Linux 2 (WSL2).
- If you prefer GUI apps on Windows, Putty is the default go-to

Connecting to Remote Hosts

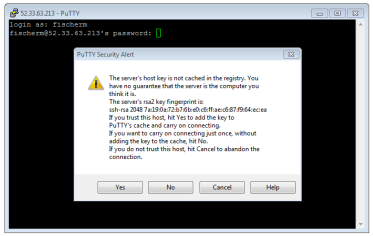
Putty



http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

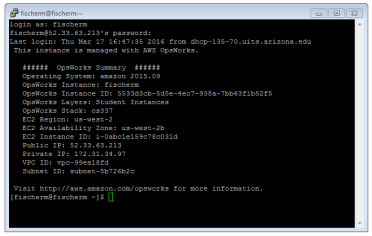
Connecting to Remote Hosts

Putty



Connecting to Remote Hosts

Putty



Lectura

Shared Computer Science Host

- Our department hosts a shared UNIX server, named lectura.
- Before logging in, create/reset your password:
  - https://helpdesk.cs.arizona.edu/selfservice
  - Your username will be same as NetID But your password can be different

```
ssh netid@lectura.cs.arizona.edu
```

next up: The HTTP Protocol and Networking