```
docker run hello-world
```

Docker Containerization

Rob Fatland, Kristen Finch
UW IT Research Computing
2-DEC-2025

Hypothesis refinement: AWS Q Developer coding assistant `docker run hello-world`

The Lab Sequence: Optional “code along”

1. Run a hello-world Docker Image (this page)
2. Access workshop content (copy with `git clone` or read along)
    - `git clone https://github.com/robfatland/docker-workshop`
3. Simplest possible Container–from–Dockerfile
4. Bind mount
5. Summary + Context
6. Build a Prime Checker web application on Localhost: C–C
7. Run ResNet-50 as a web app: From HuggingFace to Localhost
8. Summary + Conclusion

Part 2. Clone the repo for this Docker Container workshop

```
git clone https://github.com/robfatland/docker-workshop
```

Clone this repository for localhost material; or you can simply visit the above website

The repo includes documents that go further into what why and how.

- Workshop instructions: Follows this procedural
- Docker Q&A: Learning by asking questions
- Docker Reference: Learning by reading themes (includes a K-means walkthrough)
- This Google Slides deck as a pdf

The repo includes three project folders used later in the workshop

Part 3. Dockerfile

```
FROM python:3.11-slim
```

- This single line of text is the simplest possible `Dockerfile`
- python:3.11-slim is an existing image
  - Implicitly Debian Linux comes along for the ride
    - Ubuntu is based on Debian
- `-slim` is a smaller/faster-to-load version of Python (with no C compiler)
- Version number of Python (3.11): Compatibility, stability
  - For example: Run `python --version`
- Put this `Dockerfile` in project folder `/fu` to work:
  - Use `docker build` to build a corresponding Image
  - Use `docker run` to create the corresponding Container and run it
  - After we verify it is ok: Delete everything except the `Dockerfile`

Part 3. Dockerfile, sequence 1

- `cd ~`
- `mkdir fu`
- `echo FROM python:3.11-slim > ~/fu/Dockerfile`
- `cd ~/fu && ls`
- `docker build -t fu-image .`
- `docker images`
- `docker ps -a`
- `docker run -it fu-image bash`
- <In the Container terminal: Try some Linux, run Python… then **`quit()`** and **`exit`**
- `docker ps -a`
- `docker container prune`
- `docker ps -a`
- `docker images`
- `docker rmi fu-image`
- `docker images`

notice period at the end

Part 3. Dockerfile sequence 2

Repeat with a difference: Create an artifact during the visit to the Container; then leave; then go back: Still there?

- `cd ~/fu`
- `docker build -t fu-image .`
- `docker images` ← confirm the image is present
- `docker run -it fu-image bash`

- (From within the Container, on the command line)
  - `pwd`
  - `ls`
  - `echo curious > thiswasnotherebefore`
  - `exit`
- `docker ps -a` ← take note of the Container name…
- `docker start -ai crazy_bullwinkle` ← …use it to start the Container again

- (From within the Container, on the command line)
  - `pwd`
  - `ls` ← is the artifact still there?
  - `cat thiswasnotherebefore`
  - `exit`
- `docker rmi fu-image`
- `docker container prune`

Part 4: Bind Mount

This section continues to use “fu” from Part 3.

```
ls ~/fu

cat ~/fu/Dockerfile

echo WORKDIR app >> ~/fu/Dockerfile

cd fu

docker build -t fu-image .

mkdir ~/dwdata

echo This is the host machine > ~/dwdata/host.txt

cat ~/dwdata/host.txt

docker run -it -v ~/dwdata:/data fu-image bash
```

Part 4: Bind Mount 2

We are in the Container: A pseudo-terminal bash shell

Notice where we are: **`pwd`**

We can navigate to **`/data`** and confirm there is a text file here from the host

We can add a new text file from the Container

Then we leave to go see if this appears in **`~/dwdata`**

Part 4: Bind Mount 2

```
pwd

ls

cd /data

ls

cat host.txt

echo This is from the Container > container.txt

exit

cd ~/dwdata

ls

cat container.txt
```

Part 5. Summary so far and Context

- Part 1 (hello-world) invisibly retrieved an Image from Docker Hub
    - Docker Hub is a big feature of the Docker ecosystem. It is Home Depot.
- Part 2 just **`git clone`**
- Part 3 **`fu`** and the Docker workflow D > I > C
- Part 4 Bind Mount gets at Container Duality
    - A running Container is an isolated execution space…
    - …but it needs to extend into the real world to be relevant

Now we’ll step through some context to try and expand / solidify our picture of how Containers might be useful.

Read Later: Communication C ⇐⇒ C and U ⇐⇒C

Flask is a lightweight web framework that translates between Python functions and network API calls.

C–C uses a user-defined bridge virtual network that Containers use to pass information. The virtual network exists in the Linux kernel's network namespace managed by the Docker daemon. We can say this network exists inside the kernel of Docker Desktop's hidden VM.

We can inspect this network using **`docker network inspect prime-net`**.

This produces a JSON rundown of the network. Note that bridge networks like this support C–C for many Containers all on the same host. A different type of network called an Overlay Network spans multiple Docker hosts.

Miscellaneous Tactics

- Compute anywhere: Supported by the Docker ecosystem / framework

- Kristen Finch presented containerization using AppTainer on Tillicum
  - Including automatic translation from Docker Containers
- Naomi Alterman teaches Docker in MSE544
- The Carpentries features a Docker tutorial

- Why two (or more) flavors of Container?
- Why ‘Docker Desktop’ + the docker command?

Read Later: Docker Containers in a nutshell

There are three central nouns in the Docker ecosystem: ***`Dockerfile`***, ***Image***, and ***Container***.

`Dockerfile` is a text file we edit and read. Our aim is to create a Container that executes some code-based task; either continuously as with a web app; or episodically as in analysis of a dataset. Web apps are punchier for demos; analysis is more what a STEM person would use in practice.

Repeating: **`Dockerfile`**, Image, Container. **`Dockerfile`** is the starting point of a recipe for the Container. We create it within a dedicated project folder along with other components. We want to run say k-Means on a dataset. We write some Python code for this; and we also need SciKit-Learn Library with its k-Means method. So we have three files in our project folder: **`Dockerfile`**, Python code in **`app.py`**, and a **`requirements.txt`** file that lists scikit-learn, a necessary Library.

Data access we saw in the Bind Mount, part 4: Containers need to connect to the wide world.

So the **`Dockerfile`** behaves as an organizing set of instructions; and the Container executes a task when it runs. The Image is an intermediate template. It does not execute. It is a *snapshot* that includes everything needed to create and run the Container inside the Docker VM.

We built an Image using the command **`docker build`**. The subsequent command **`docker run`** creates a Container from the Image and then runs it. The image remains unchanged so the process can be repeated.

Crossing the fine line into too much detail: An Image is actually a layered filesystem. If we change only the Python code and do a new build: Docker will rebuild just that layer, not the entire Image: Faster development.

Docker Vocabulary

- Docker Desktop
- Docker CLI $\longrightarrow$ `docker <task name> <qualifiers>`
- Dockerfile
- Image
- Container
- Docker workflow   pull → create → run → exit
- Docker hub
- Docker Engine
- Docker client
- Docker daemon
- Docker root
- Docker Compose

Container *Duality*

- On one hand: Containers function like hermetic automaton VMs (that *borrow* the OS)...
    - "I'm ***not*** a VM!!! …but I *am* ***isolated***" -the Container

- On the other hand: Containers need connectivity to the outside world
    - Container to terminal or browser (port-wise)
    - Container to disk (volume-wise)
    - Container to Container (network-wise)

- Today's plan / Container-learner's plan
    - Get into the Container headspace
    - Maximize awareness, skills
    - Minimize the rabbit hole

Docker Containers need connectivity to the outside world in three ways (setting aside terminal access): ports, volumes and networks. Port connectivity is for Container to Host port of course. Volume connectivity is for filesystem overlap, i.e. data / results from the Container are visible in the Host. Networks exist within the Docker VM and facilitate communication between Containers.

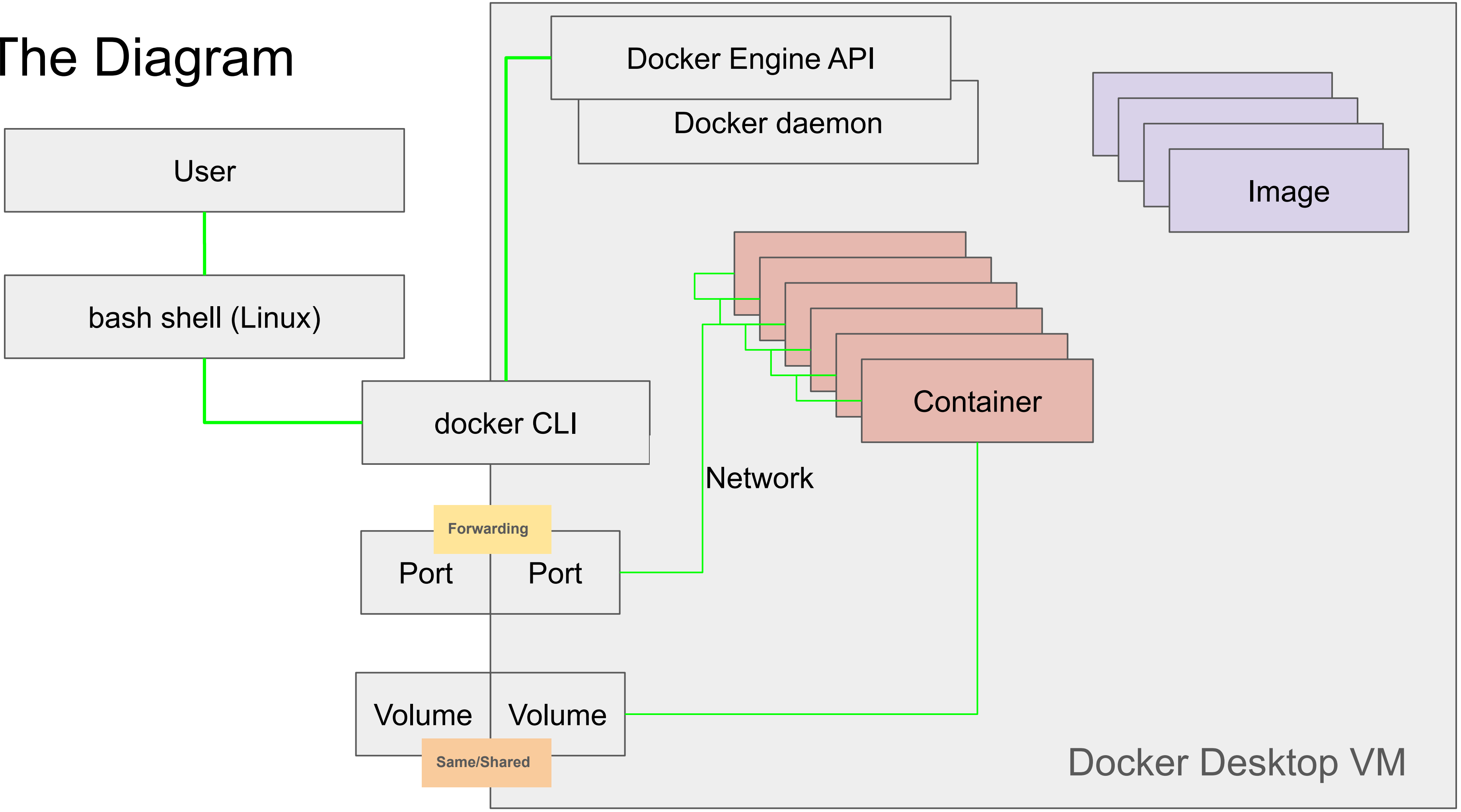
Philosophical Concerns

Containerization – we will find – follows from some design principles

- isolation	no collisions / overlap with host
- abstraction	Images, Containers are “somewhere”
- multiplication	From one Image: Many Containers
- ~~derision~~ resilience	Container modified → auto-saves
	(and more resilience to follow)

Separation of Concerns: Map pieces of a complex workflow to Containers

The Diagram



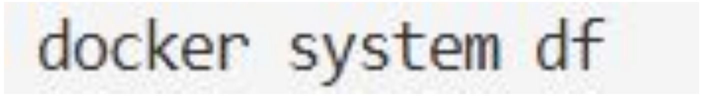
User
bash shell (Linux)
docker CLI
Docker Engine API
Docker daemon
Image
Container
Network
Forwarding
Port
Port
Volume
Volume
Same/Shared
Docker Desktop VM

Docker Images and Containers

‘Don’t Worry About It: *Abstraction’*

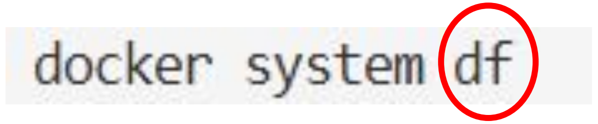
- In the bash shell: We run the command **`docker`** with arguments
- **`docker run hello-world`** checks locally, then scoots to docker hub
- There it gets an **`image`** and converts that into a **`container`**
- The container runs, does what it does, halts

We never worry about *where* the containers and images are on **`localhost`** because we are managing them via the **`docker`** command. This is a key design feature called *abstraction*.

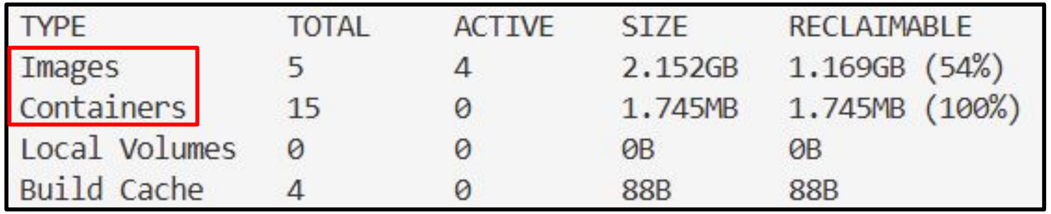


```
docker system df
```

No worrying: Linux ‘disk free’ (df) command



```
docker system df
```



```
TYPE            TOTAL     ACTIVE    SIZE       RECLAIMABLE
Images          5         4         2.152GB    1.169GB (54%)
Containers      15        0         1.745MB    1.745MB (100%)
Local Volumes   0         0         0B         0B
Build Cache     4         0         88B        88B
```

this df command is running inside the Docker VM

…so now we turn to the file/image/container ecosystem…

Ports, Volumes, Networks

- Reminder: Containers must connect to the outside world
    - Port mapping: localhost to Container ‘View a web app’
        - Resolve: Does this include the -ai attach?
    - Volumes: Preserve stable information
        - Named volumes (inside the Docker VM)
        - Bind mounts (alias for a host folder)
    - Networks: Container ←→ Container

`docker run` versus `docker start`

- We run the docker CLI from a terminal: **`docker <action> <etc>`**
- **`docker run`**: Create a container from an image and run the container
  - default behavior: Attach the terminal to the container’s standard output

…contrast with…

- **`docker start`**: Container already exists; so start it up
  - default behavior: This container is old news so terminal is not connected to standard output

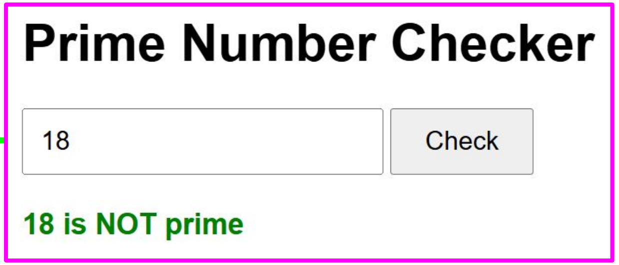
We can override the **`start`** default behavior using **`-a`** for “attach”

This wires up the Container output to the terminal display

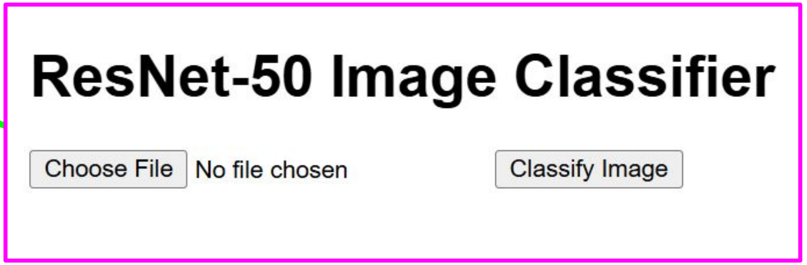
End of Part 5 Summary + Context

Remaining:

- Two Container build, test
- ResNet build, test
- Summary & Conclusion



18 is NOT prime



Part 6: Two Containers Working Together

Container Duality (isolation ~ connection) provides for creating networks defined within the Docker VM. Result: Multiple Containers on a single host can intercommunicate.

This example: One Container acts as a frontend (User-oriented) piece of a web application. Second Container is the backend: Evaluates a positive integer as prime or composite.

Both Containers use the Flask web framework (from Python)

We run both Containers on our host. Test by copying this URL into the browser: http://localhost:8080. Prep commands:

- Create the network
- Build the front-end Image
- Build the back-end Image
- Run the front-end Image → front-end Container
- Run the back-end Image → back-end Container
- **`docker ps`** (to verify the Containers are running)

Part 6: Two Containers Working Together

```
docker network create prime-net
docker build -t prime-checker .
docker build -t prime-frontend .
docker images
docker run -d --name prime-api --network prime-net prime-checker
docker run -d --name prime-web --network prime-net -p 8080:8080
prime-frontend
docker ps
```

Test: Browser tab address bar: `http://localhost:8080`

Read Later: *Prime Check* localhost web app narrative

We have two directories: **`prime-frontend`** and **`prime-checker`**. These include respective Dockerfiles used to build two Images called **`prime-frontend`** and **`prime-checker`**.

These Images were used to create and run two Containers called **`prime-web`** and **`prime-api`**.

We created a Docker network called **`prime-net`**.

We associate **`http://localhost:8080`** with the proxy server: The Container called **`prime-web`**. This association was established between the host machine port 8080 and the Container port 8080 (where Flask is listening for traffic) using

```
docker run -d --name prime-web --network prime-net -p 8080:8080 prime-frontend
```

This **`docker`** command uses **`run:`** Converts the named Image to a Container and runs that Container.

Uses the **`-d`** flag to run in detached mode, as a background task. (We get the cursor back right away.)

The **`-p`** flag indicates a port mapping: **`host port:Container port`**.

We use **`docker run`** to start **`prime-api`** as well, also with **`--network prime-net`**

Localhost browser tab: Enter port **`8080:`** Translates to port **`8080`** inside the Container (> Flask). Sends back the HTML for the web app.

The User enters a number, say 17, and clicks Check: A different message (a prime query for 17) goes to **`prime-web`**.

`prime-web` in turn connects to **`prime-api`** using the network **`prime-net`**. The evaluation comes back (17 is prime) and this result is passed back to the browser.

The browser never interacts with the backend **`prime-api`** Container.

Part 7: Running ResNet in a Container

ResNet is a **Res**idual Neural **Net**work invented circa 2015.

We will use an instance that fits in a modest Container.

It has already been trained to recognize 1000 categories of objects (ImageNet dataset)

It has 50 neuronal layers; and 26 million parameters.

The image classifier is not so much the point here…

- …rather: a concluding look at `docker build + run`
- …particularly relating to huggingface

```
cd ~/docker-workshop/resnet-classifier
docker build -t resnet-classifier .
docker run -d --name resnet -p 8081:8080 resnet-classifier
```

browser tab address bar: `http://localhost:8081`

```
WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

RUN python -c "from transformers import AutoModelForImageClassification;
AutoModelForImageClassification.from_pretrained('microsoft/resnet-50')"

COPY app.py .

EXPOSE 8080

CMD ["python", "app.py"]
```

`docker run` → from Image to (running) Container

RUN → What happens to create the Image
CMD → Dockerfile: What runs in the Container

WORKDIR “where home is in the Container” `/app`
COPY → from project folder to home

Part 8. Summary and Conclusions

Once these atomic Containers become second nature: A next topic is getting multiple Containers working in concert on a collective task. This is known as Container orchestration.

On a smaller scale you can use **`docker compose`**.

On a larger scale (across many host machines) you have Kubernetes and so on.

Part 8. Summary and Conclusions

- `docker build`	`Dockerfile (.) to Image`
- `docker create`	`Image to Container (no run, rare)`
- `docker run`	`Image to running Container`
- `docker start`	`Stopped Container to running Container`

- The UW IT RC workshop: November, KF: AppTainer + Tillicum
    - Bridge to Docker via subtle auto-convert: Docker Hub > Tillicum HPC
- Docker (et al) key design principles
    - Rapid development, minimal support infrastructure
    - Isolation from host
    - Portability
    - Persistence (Containers remember changes)
    - From SPF/Research paradigm → Research Software Engineering paradigm
- There may (will) be bumps along the way
    - Localhost ARM may not translate to HPC AMD processor architecture
    - One approach is Multi-Platform Building
    - Another approach: Build where you will run

Conclusion

- Docker is a Container ecosystem → learning curves
- This workshop attempted ‘from the ground up’
- Note existence: Tillicum / AppTainer workshop
- Docker Hub: Free, public Container space
- GitHub: Free, public Project space
- HuggingFace: Free, public Data + Models space
- Coding Assistants (Copilot, Q etc)
- Orchestration
- Duality
- UW IT Research Computing + eScience Institute

Center Title Left Text Template