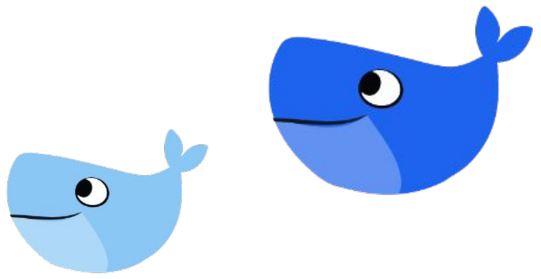
dockercon 23

Getting Started with Docker

Michael Irwin

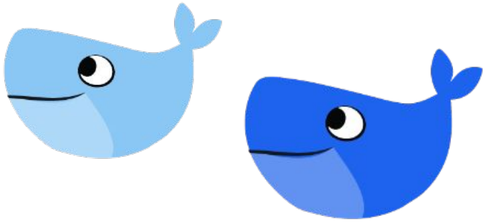
Developer Relations | Docker



Michael Irwin

Developer Relations | Docker

Interact with me @mikesir87



Group selfie!

#dockercon

Workshop agenda

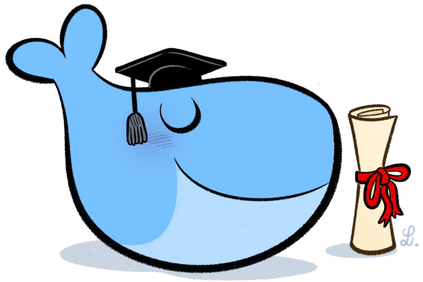
8:00-8:40	Container and image intro
8:40-9:15	Using containers for development
9:15-9:30	Break
9:30-10:10	Building your own images
10:10-11:00	Multi-service applications
11:00-11:15	Break
11:15-12:00	Intro to Docker Compose

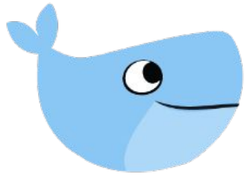
01

Container and image intro

This section's learning objectives

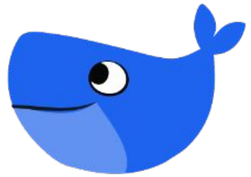
- By the end of this section, you'll be able to answer these questions:
  - What is a container? Image?
  - How's a container different than a VM?
  - What is Docker's *current* role in the container ecosystem?
  - Why should I care about this container movement?
  - How do I run a simple container? See what's running? View logs?





Docker’s vision

Increase the time **developers** spend on innovation
and decrease the time they spend on everything else

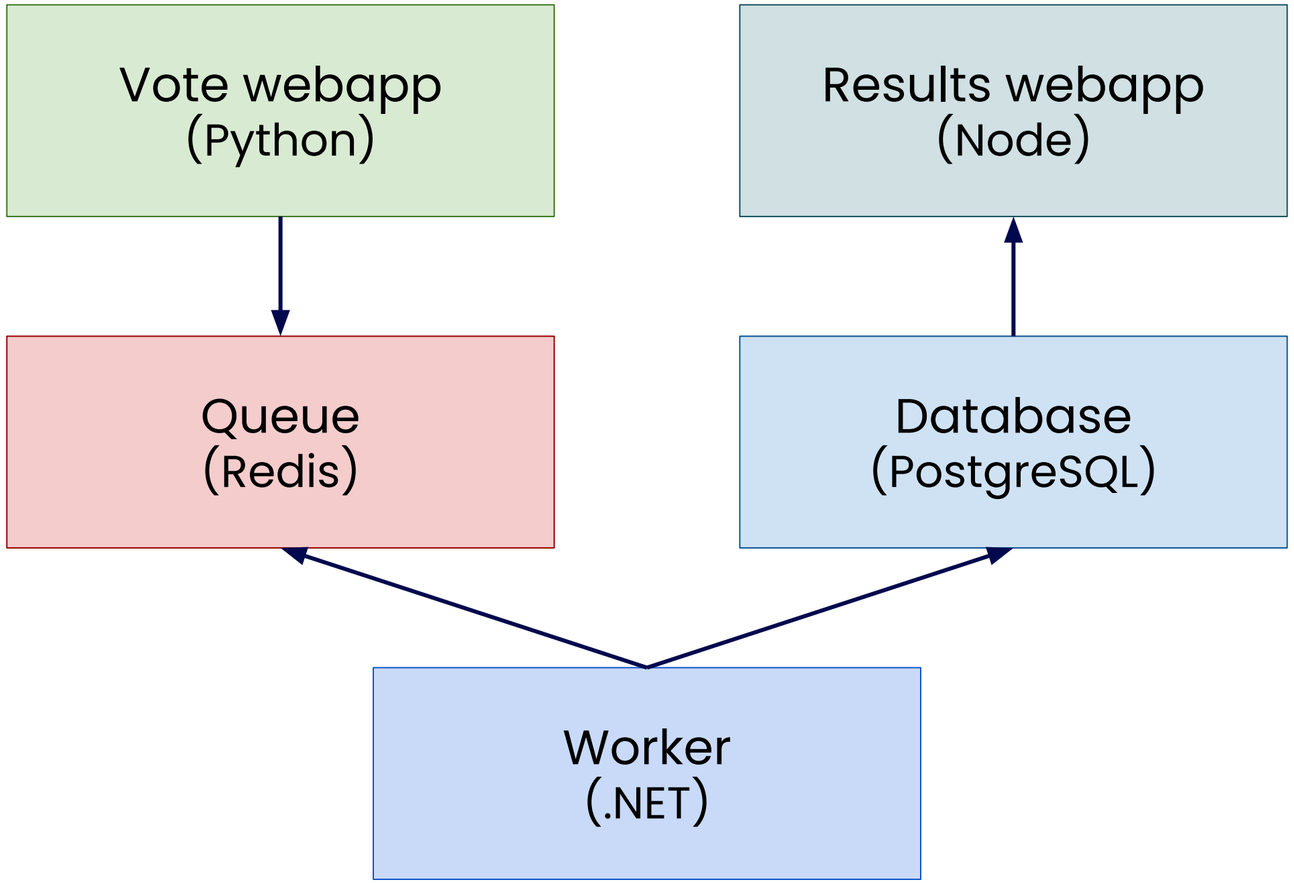



Welcome! Glad to have you on the team!
We already assigned you an easy first task. Challenge - see if you can knock it out by lunch!


Welcome! Glad to have you on the team!
Clone the repo, use README to install env. Spend the first week updating the docs and setting up your machine. Then work on the task

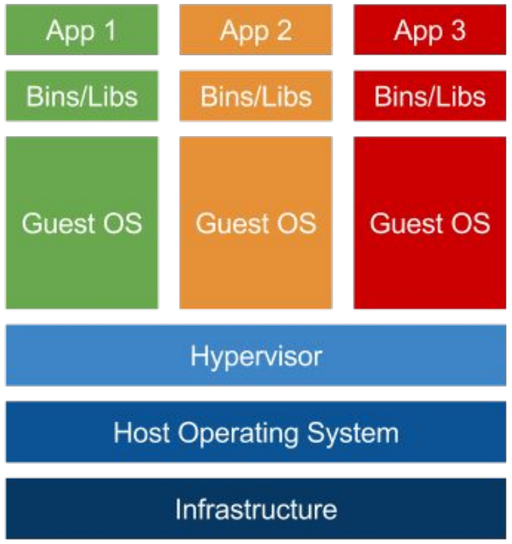
Wouldn’t it be nice if we could just *run* software, without installing or configuring it?

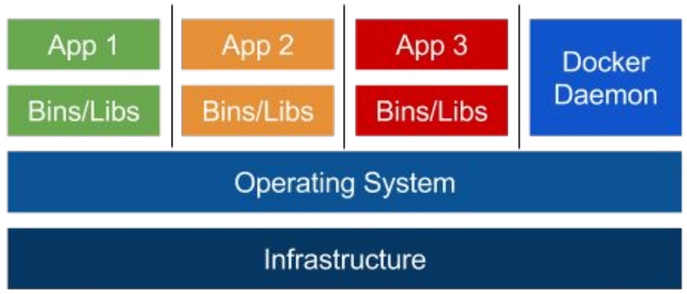
Demo time!



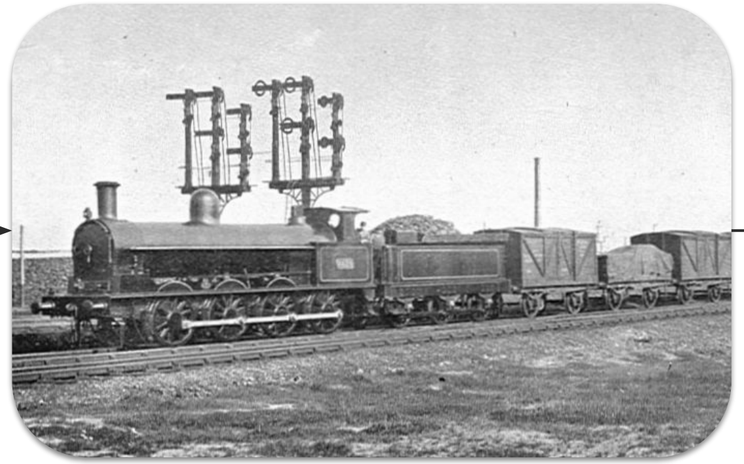
Example voting app: github.com/dockersamples/example-voting-app

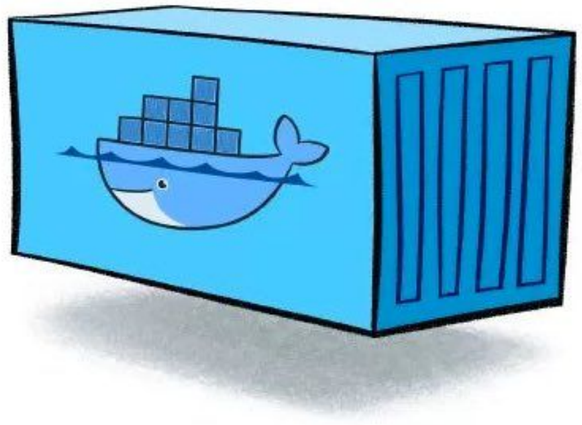
Containers vs VMs





- Containers are…
  - Simply isolated processes, not full VMs
  - Much more portable than VMs
  - Using the same kernel as the host
  - Able to limit CPU/memory
- VMs are…
  - A full OS with kernel
  - A deeper level of isolation
  - Able to provide storage and networking limits

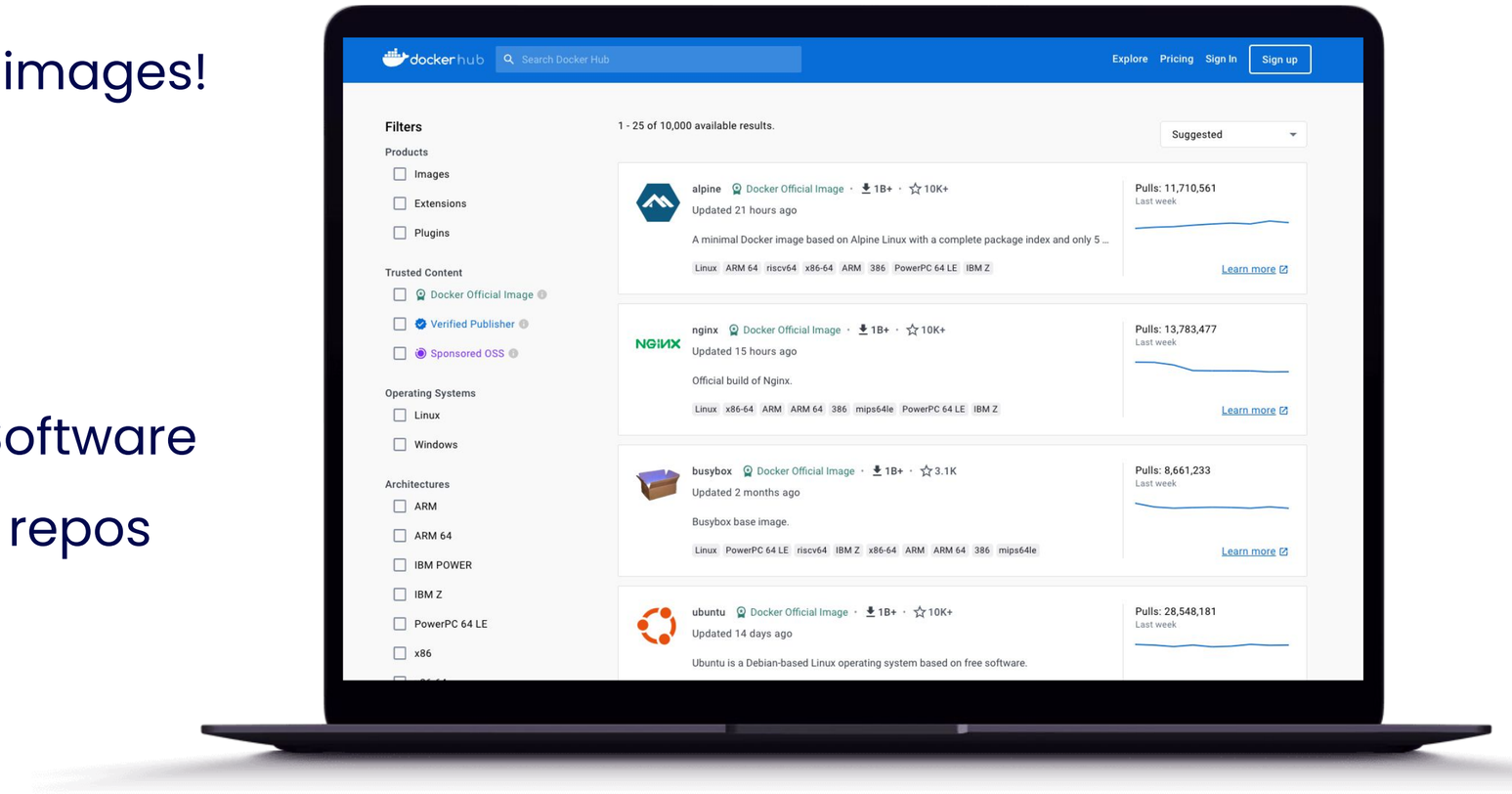




Are containers *just* for microservices?

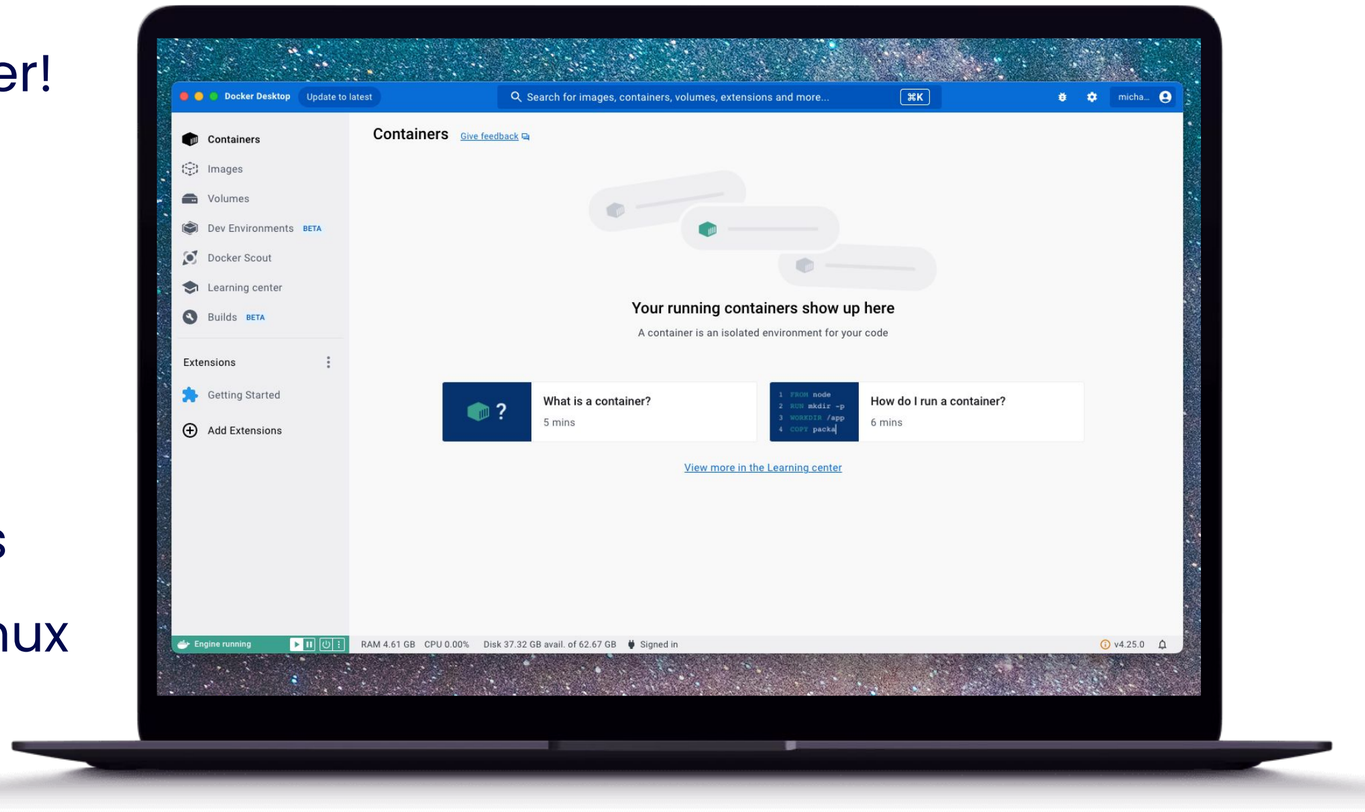
Docker Hub

- The default marketplace for images!
- Trusted Content
  - Docker Official Images
  - Verified Publishers
  - Sponsored Open Source Software
- Personal and organizational repos



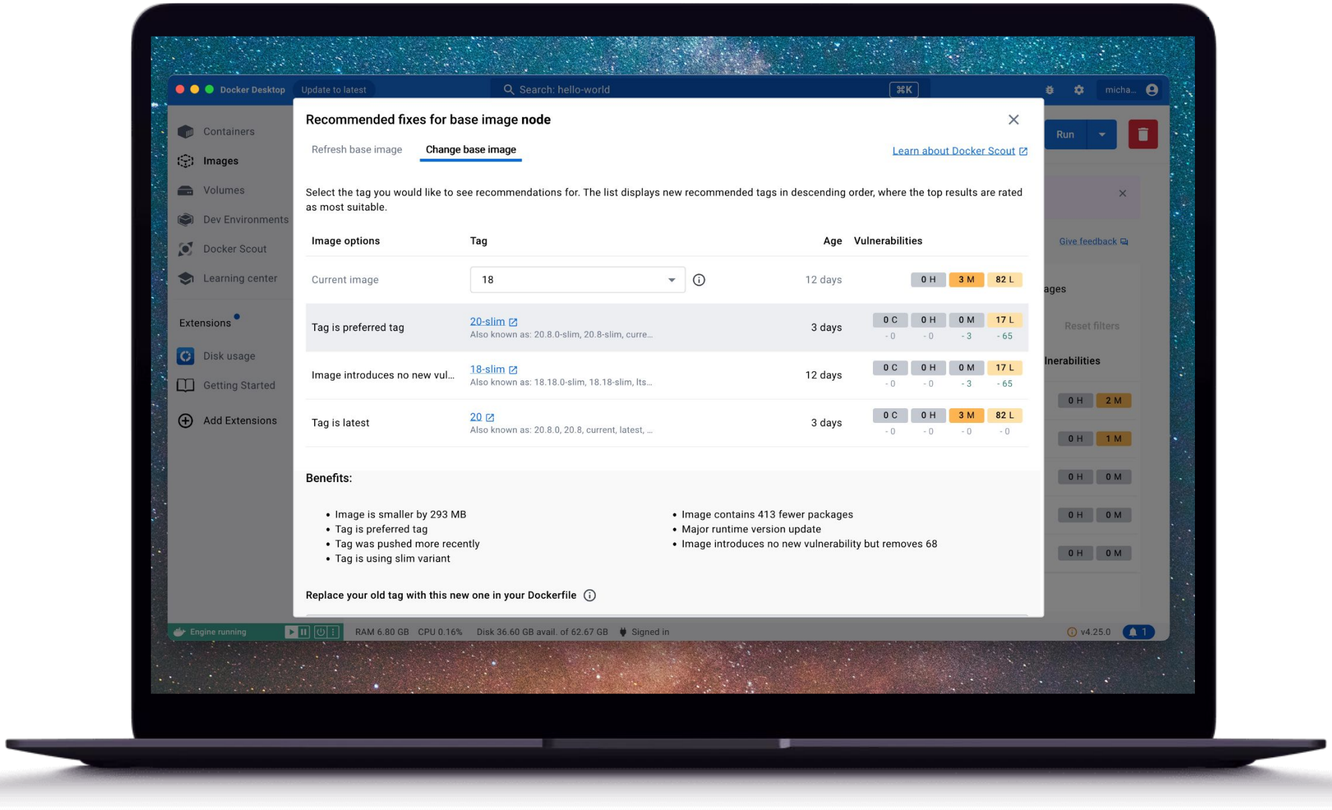
Docker Desktop

- The local install for everything Docker!
- Provides the ability to…
  - Run containerized workloads
  - Build images
  - Launch Kubernetes with one click
- Provides both CLI and GUI interfaces
- Works across Mac, Windows, and Linux



Docker Scout

- Provide developers with the tools needed to build secure software supply chains
  - Vulnerability information
  - Remediation recommendations
- Provide organizations with understandings on what's going on with their images



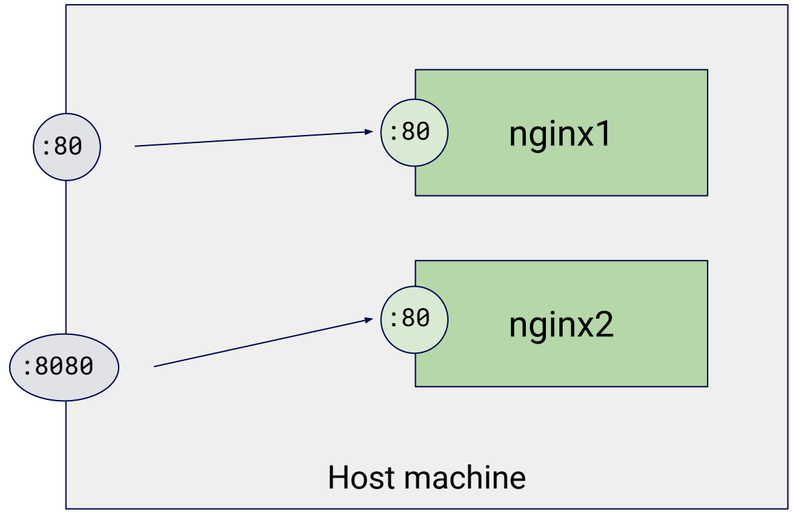
Starting a container using the CLI

```
# Start a mongo container, named mongo, in the background
docker run --name mongo -d mongo

# Start an interactive ubuntu container and auto-remove it when it exits
docker run --rm -it ubuntu

# Start nginx and expose its web server port to the host
# The port mapping is [host port]:[container port]
docker run -d -p 80:80 nginx
```

Explaining port mappings

- Every container gets its own IP address
- When a container uses a port, it's only listening on its IP address
- Exposing a port = create config to connect the host network interfaces to the container

```
docker run -p 80:80 --name nginx1 nginx
docker run -p 8080:80 --name nginx2 nginx
```



“Advanced” container starts

```
# Start a postgres container and specify the root user's password
docker run -d -e POSTGRES_PASSWORD=test postgres

# Adding labels to containers
docker run -d -p 80:80 --label app=sample-app --label component=website nginx
```

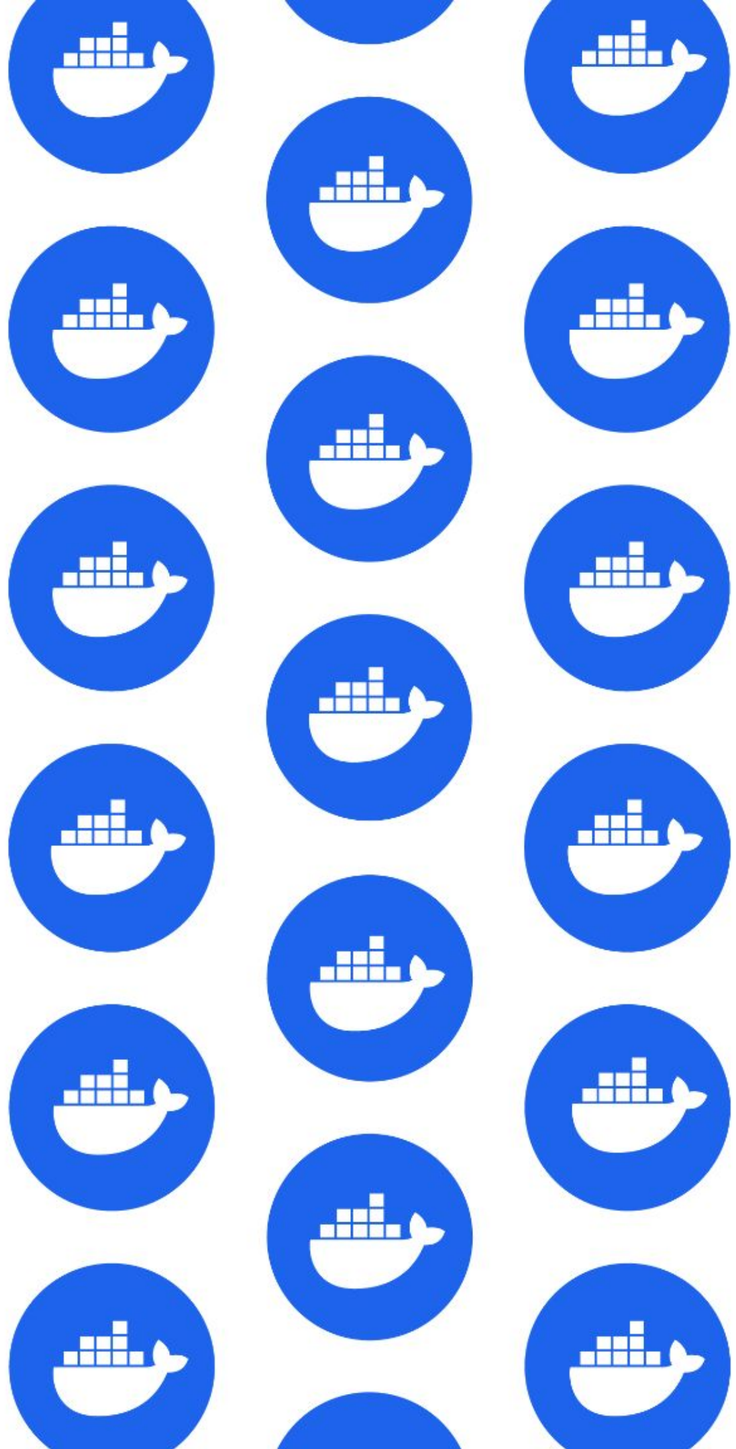
Working with containers

```
# List running containers
docker ps

# List all running containers, filtering by a specific label
docker ps -a

# List all running containers, including stopped containers
docker ps --filter=label=app=sample-app

# Remove a container whose ID begins with abc (-f stops it first)
docker rm -f abc
```

Troubleshooting/debugging containers

```
# Run a single process inside an existing container
docker exec <container-identifier> <command>
docker exec abc ls /usr/share/nginx/html

# Start an interactive shell inside a container
docker exec -ti abc bash
docker exec -ti abc sh

# View logs for a container
docker logs <container-id>
```

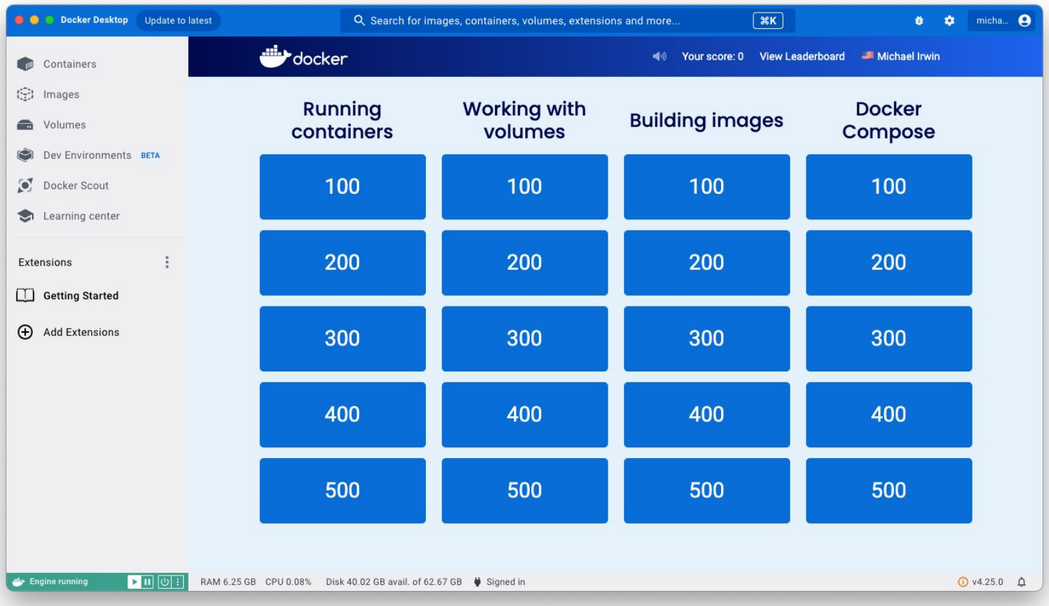
HANDS-ON TIME!!!



Hands-on instructions

1. Go to the workshop repo at **github.com/mikesir87/dockercon23-workshop-materials**
2. Follow the README instructions to install the extension workshop
3. Open the extension and start with the **Running containers** category

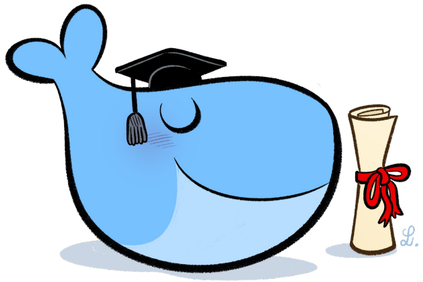
We'll resume at 8:40.



02 Using containers for development

This section's learning objectives

- By the end of this section, you'll be able to answer these questions:
  - How can I use containers in local development?
  - How can I persist data created by containers?
  - How can I share my host files with a running container?

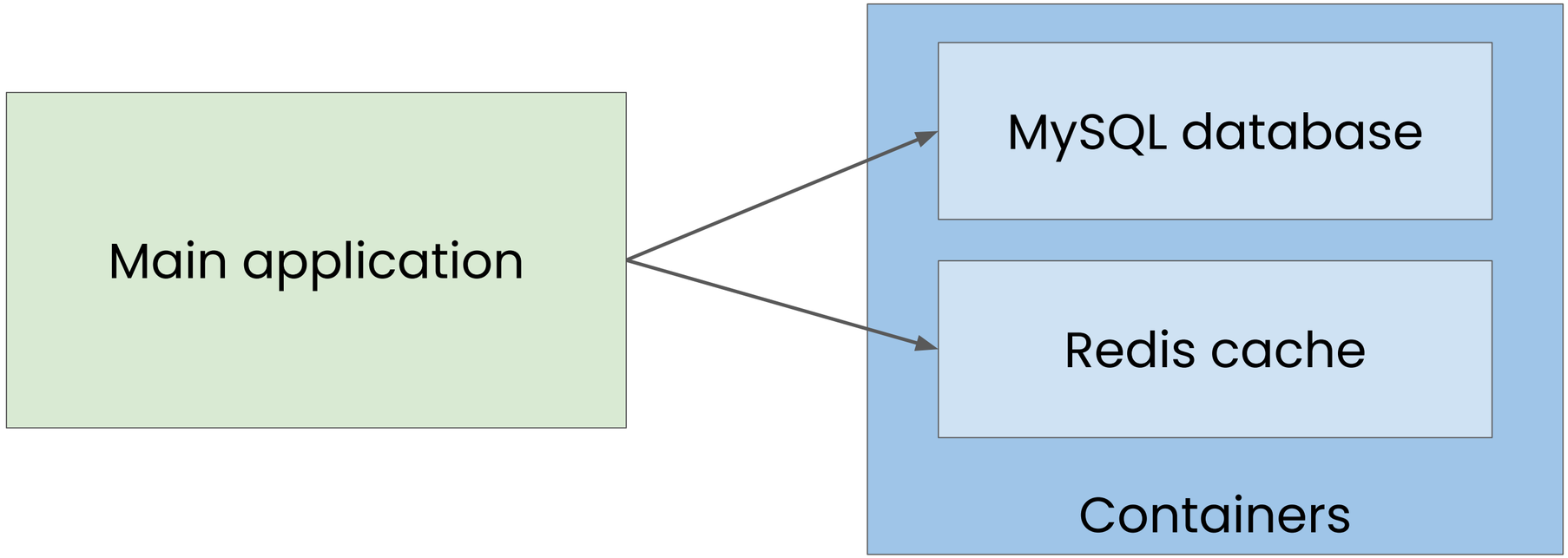


Using containers in development

- There are several ways to use containers
  - Run dependent services
  - Test your applications
  - Provide the dev environment
  - And more!

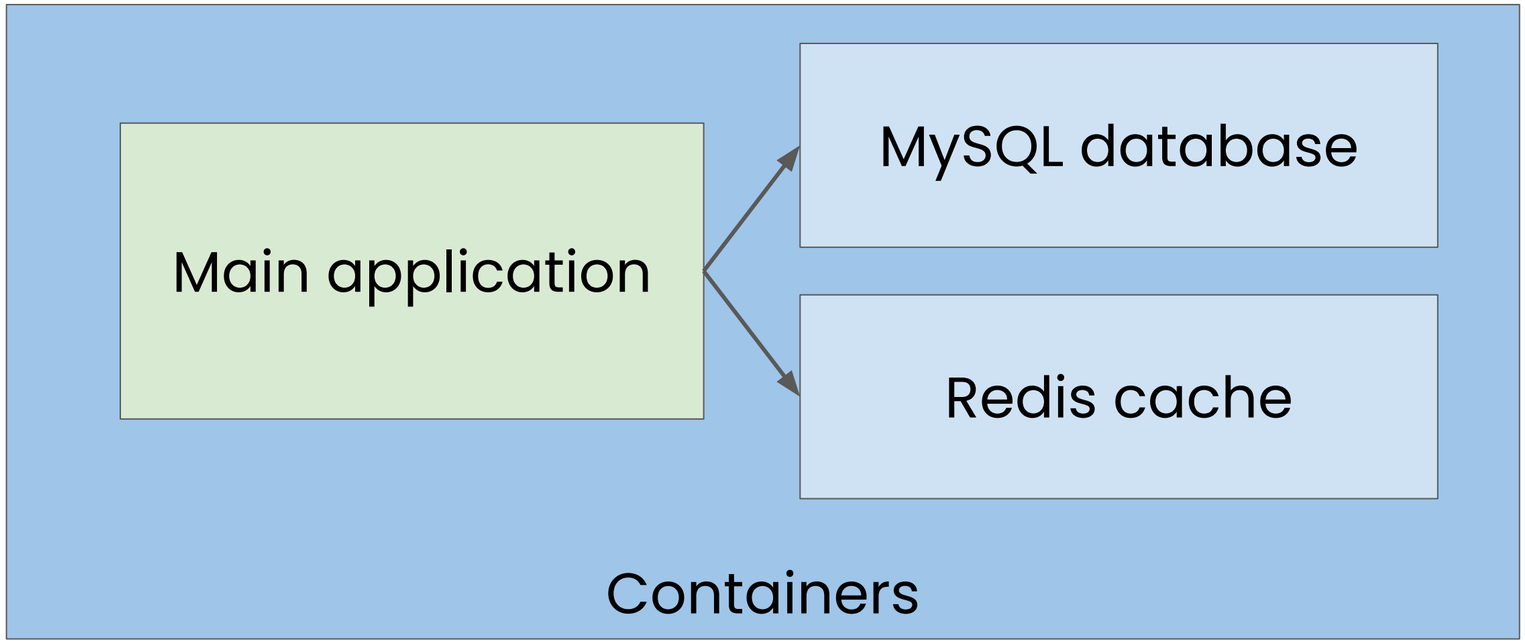
Run dependent services

- Keep doing development the way you've always done it
- Identify the dependent services and run those as containers



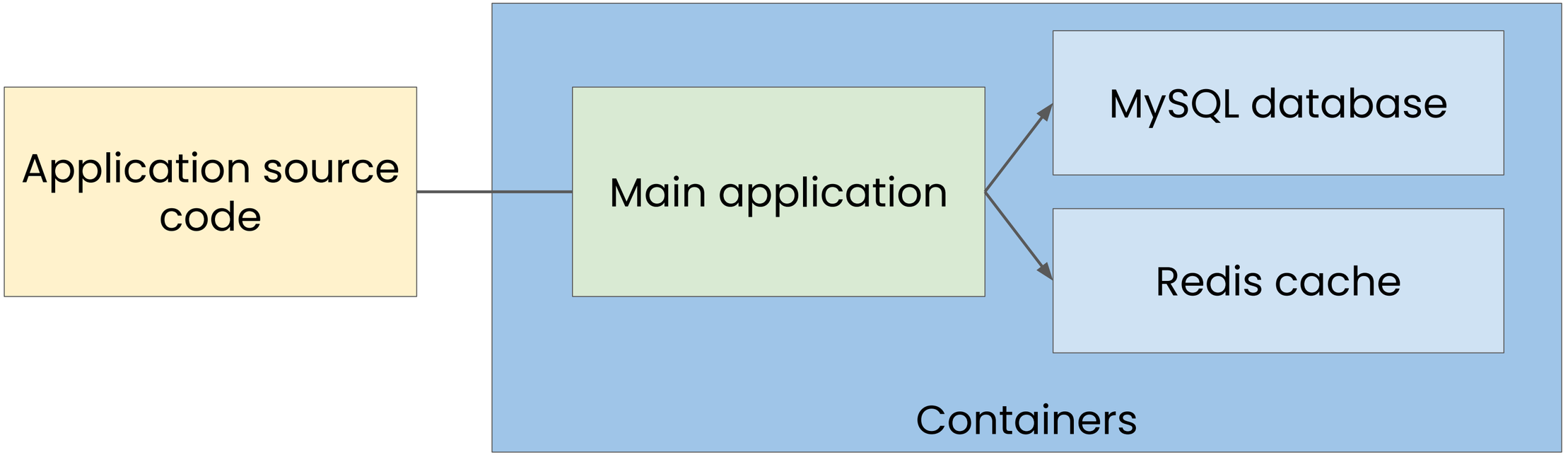
Test your app with containers

- Build the container images locally to validate everything works
- Feedback loop = build a new image and restart the container with each code change



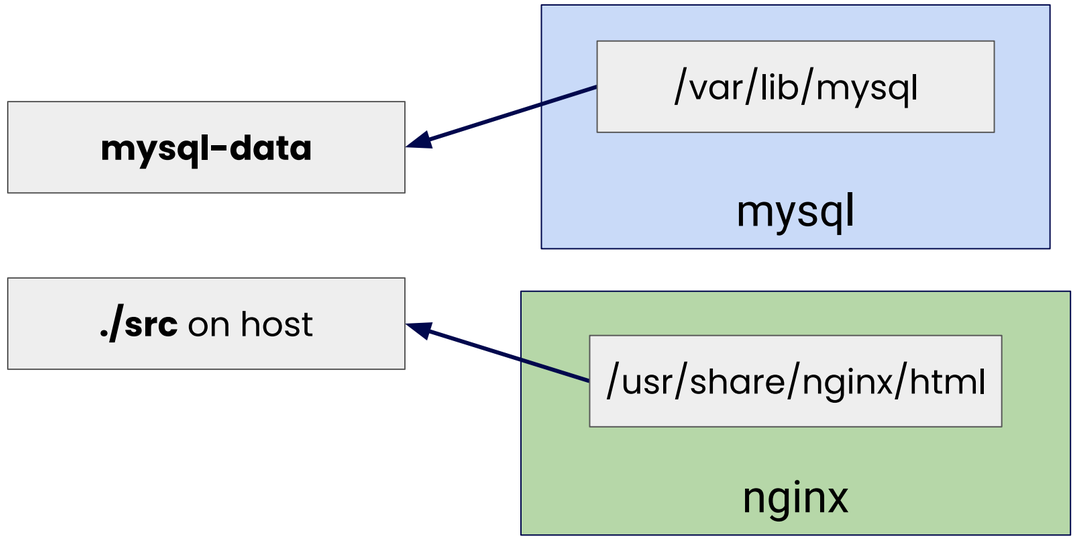
Develop *in* containers

- Use the container for the runtime environment and either share or sync file changes into the container
- Feedback loop = immediate (or close to it)



Volumes and bind mounts

- The container's filesystem comes from the image
- Bind mounts let us share files from the host
- Volumes let us persist data in the container

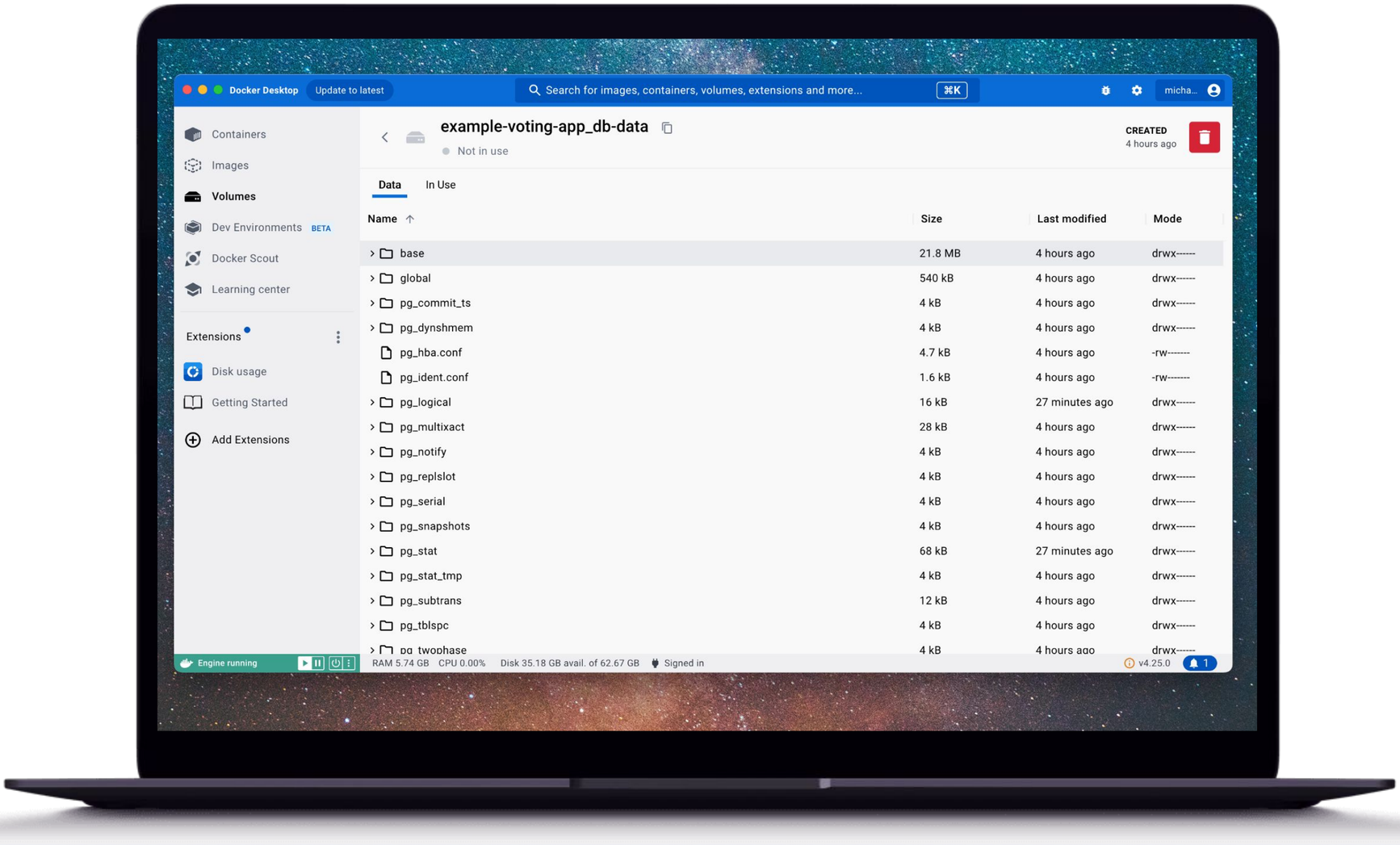



```
docker run -v mysql-data:/var/lib/mysql ... mysql

docker run -v ./src:/usr/share/nginx/html ... nginx
```

Working with volumes

```
# Create a named volume
docker volume create mysql-files

# List all volumes
docker volume ls

# Connect a volume to a container
docker run -v mysql-files:/var/lib/mysql ... mysql

# Remove a volume (only works if it's not in use)
docker volume rm mysql-files
```

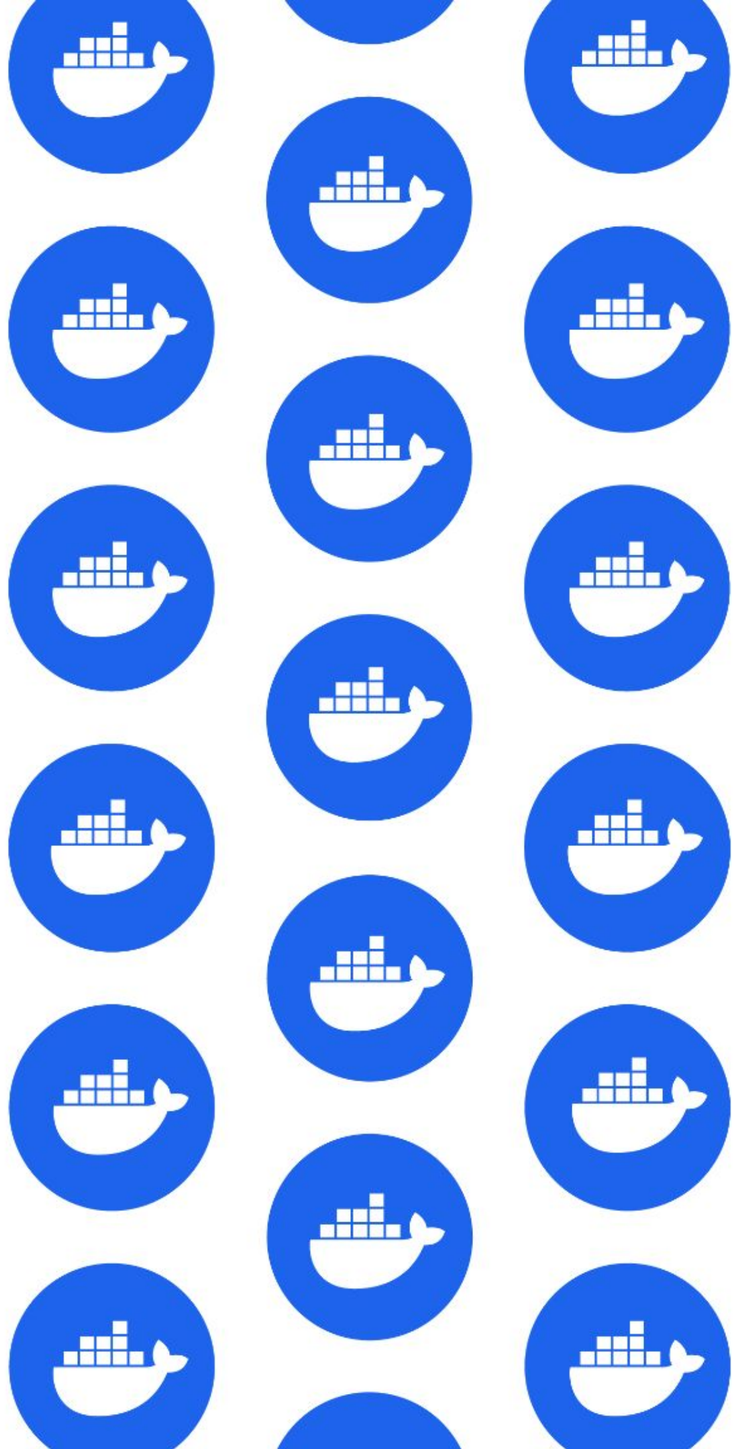
Visualizing what's in a volume



HANDS-ON TIME!!!

Work on the “Working with volumes” category

We’ll work until 9:15 and then take a break until 9:30

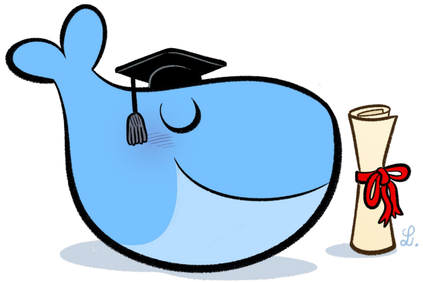


03

Building your own images

This section's learning objectives

- By the end of this section, you'll be able to answer these questions:
  - What actually *is* an image? What are image layers?
  - How can I see what's in an image or how it was built?
  - What's a `Dockerfile` and how do I use it to build an image?
  - How do I share my custom-built images?

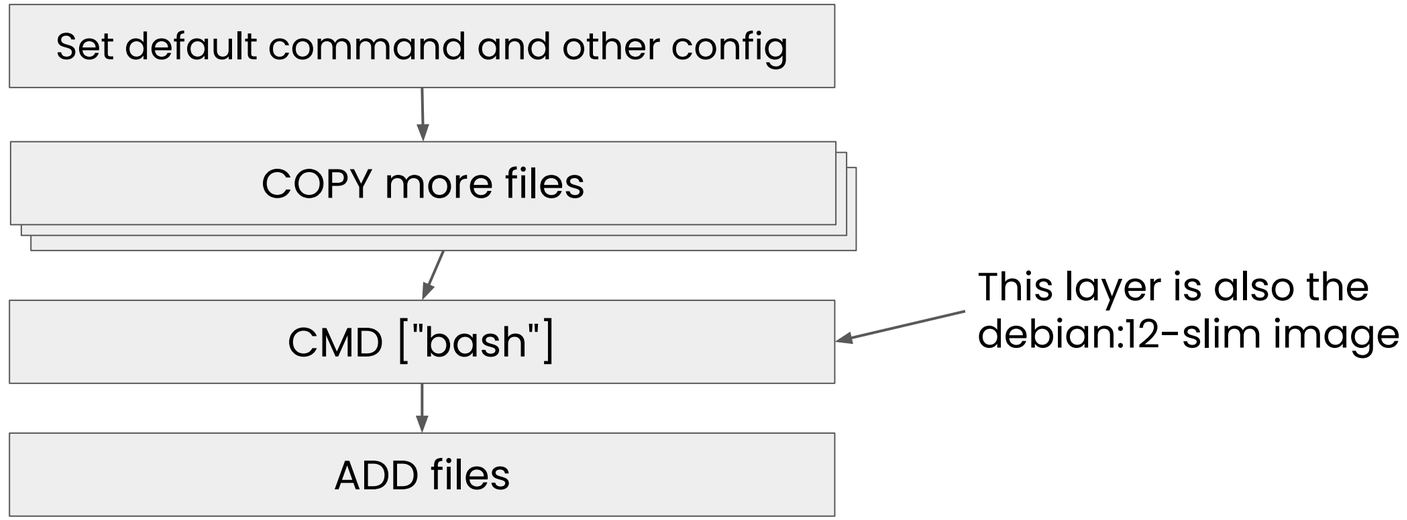


When I pull image, what's actually being downloaded?

Image layering

- Images are composed of layers
- Each layer contains a set of filesystem changes
- See the layers by using the `docker image history` command

```
docker image history nginx
```



Creating an image manually

- You can create your own image manually
  - Start a container with `docker run`
  - Make changes with `docker cp` or `docker exec`
  - Commit the changes using `docker commit`

Or script it!

- Build images using a **`Dockerfile`**
- Text-based file with instructions on how to build an image
  - **`FROM`** - the base image to start from
  - **`WORKDIR`** - set the working directory in the image
  - **`COPY`** - copy files from host into the image
  - **`RUN`** - run a command
  - **`EXPOSE`** - specify a port the container wants to use
  - **`CMD`** - set the default command

```
FROM node:lts
WORKDIR /usr/local/app
COPY package.json yarn.lock ./
RUN yarn install
COPY ./src ./src
EXPOSE 3000
CMD ["node", "src/index.js"]
```

Image naming

- Image names have several components, separated by slashes

```
my-registry.com/namespace/image-repository:tag
```

The registry this image came from/will go to (defaults to Docker Hub)

The namespace. Typically a username, org, or team name. Can be multiple segments.

The final image repository

The image tag. Can be different versions, builds, variants, etc.

Image naming examples

- alpine
  - Docker Hub; `library` namespace (Official Images); `alpine` image; `latest` tag
- mikesir87/cats:1.0
  - Docker Hub; `mikesir87` namespace; `cats` image; `1.0` tag
- my-registry.com/org/webapp:3accda9
  - Registry at `my-registry.com`; `org` namespace; `webapp` image; `3accda9` tag

Building and pushing images

```
# Build an image using the Dockerfile in this directory and name the
# new image mikesir87/my-website. The mikesir87 portion is my Docker Hub
# username, so swap it out with your username
docker build -t mikesir87/my-website .

# Push the image to Docker Hub!
docker push mikesir87/my-website
```

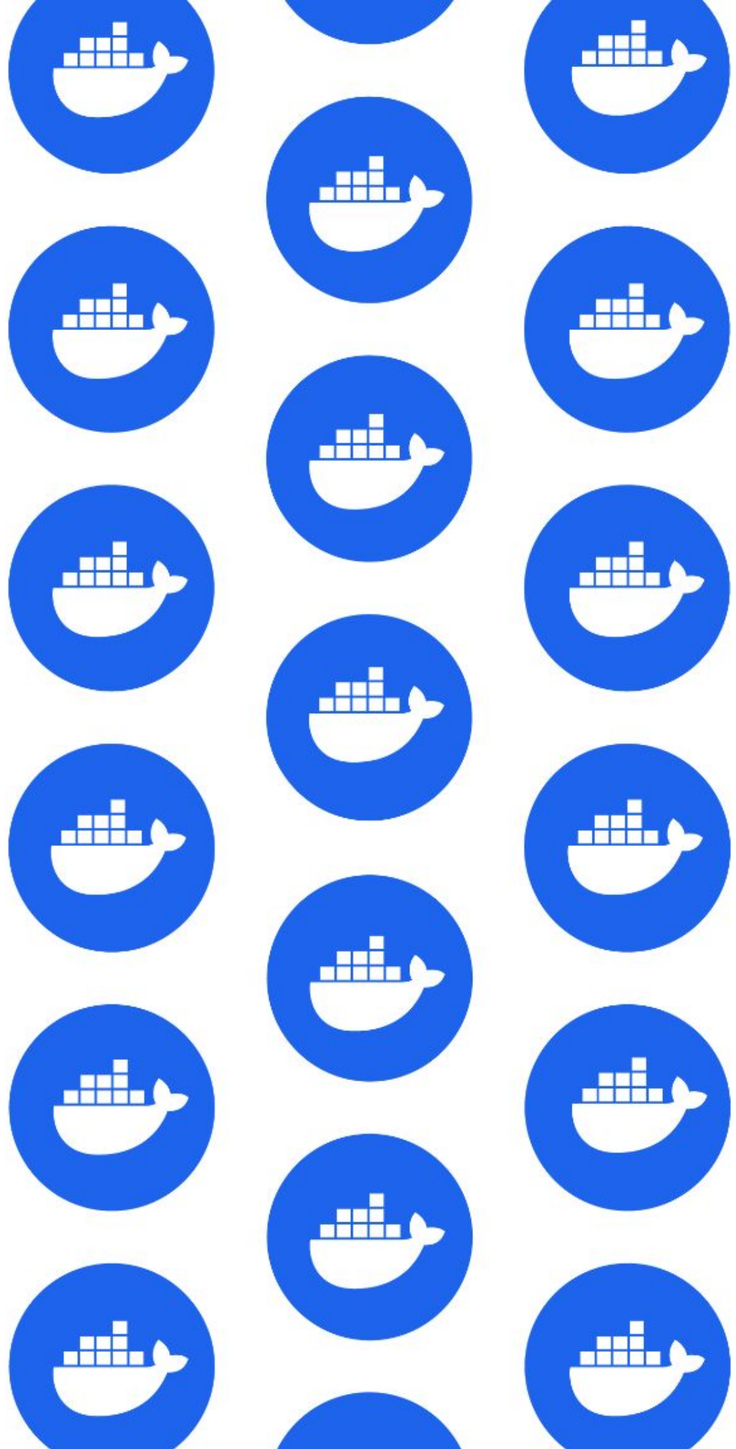
A few build best practices

- Each container should do one thing and do it well
  - Avoid creating “general purpose” images
- Don’t ship dev tooling into production
  - Multi-stage image builds help tremendously here!
- Use or create trusted base images
  - Can leverage our Docker Official Images or build your own
- Pin your images (don’t use the default :latest tag)
  - `FROM node` -> `FROM node:20.5-alpine3.17`
  - Can pin to specific SHA sums too! `FROM node@sha256:0b889cbf7…`

HANDS-ON TIME!!!

Work on the “Building images” category

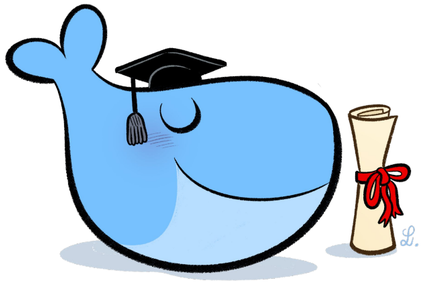
We’ll resume at 10:10



04 Multi-service applications

This section's learning objectives

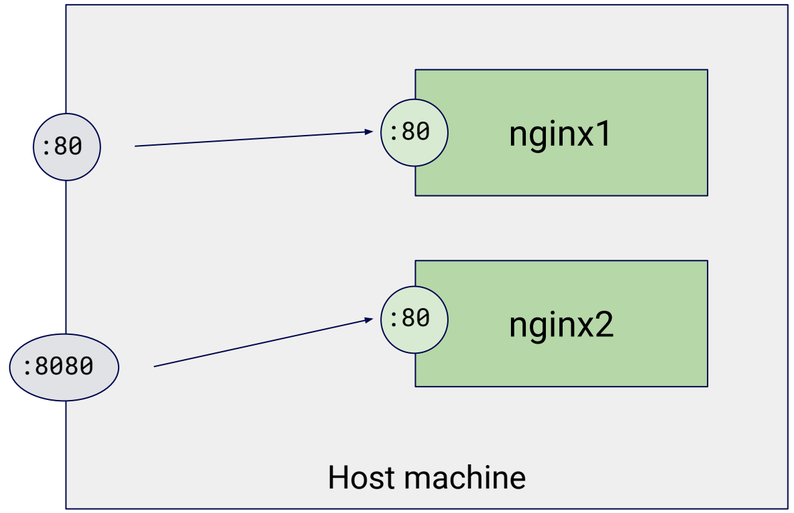
- By the end of this section, you'll be able to answer these questions:
  - How can I allow containers to communicate with each other?
  - How can one container find another container?
  - How do I create and use networks and network aliases?



If each container should do one thing, how do we connect them together?

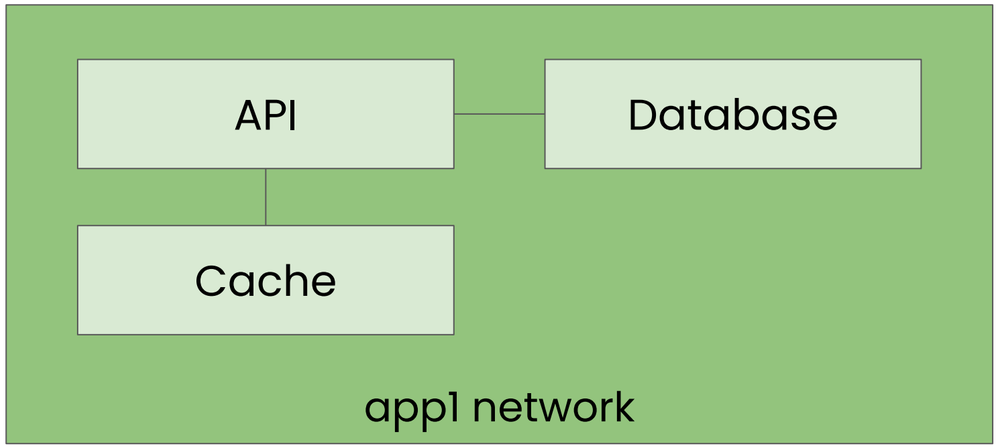
Network communication

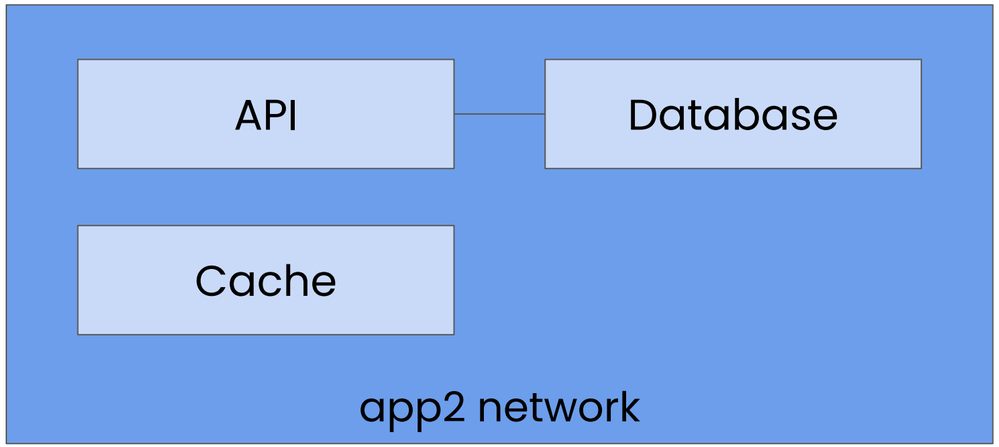
- The network namespace gives...
  - Each container its own IP address
  - Its own loopback interface (localhost)
- Each containerized process listens on ports for its network interfaces
  - Remember port mappings?



Docker networks

- Any two containers on the same network can talk to each other
- Containers can be on multiple networks
- Create and manage networks with the `docker network` command
- Attach containers either at launch or later



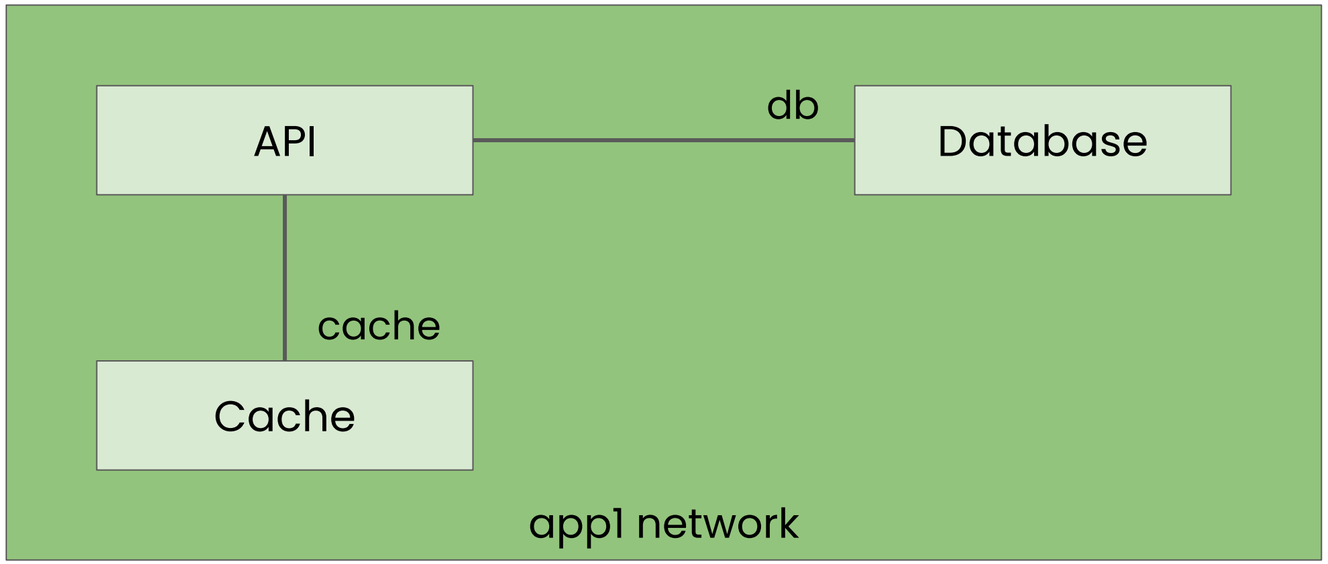


Basic network commands

```
# Create a network named app
docker network create app

# Start a nginx container on the app network
docker run -d --network app nginx

# Connect an existing container, with id abcd1234, to the app network
docker network connect app abcd1234
```

Service discovery

- Containers can be given a “network alias”
- Effectively, this becomes a DNS name that will resolve only within the same networks



Basic network commands

```
# Create a network named app
docker network create app

# Start a mysql container on the app network, given it an alias of db
docker run -d --network app --network-alias db ... mysql

# Connect an existing container, but give it a network alias
docker network connect --alias=cache app abcd1234
```

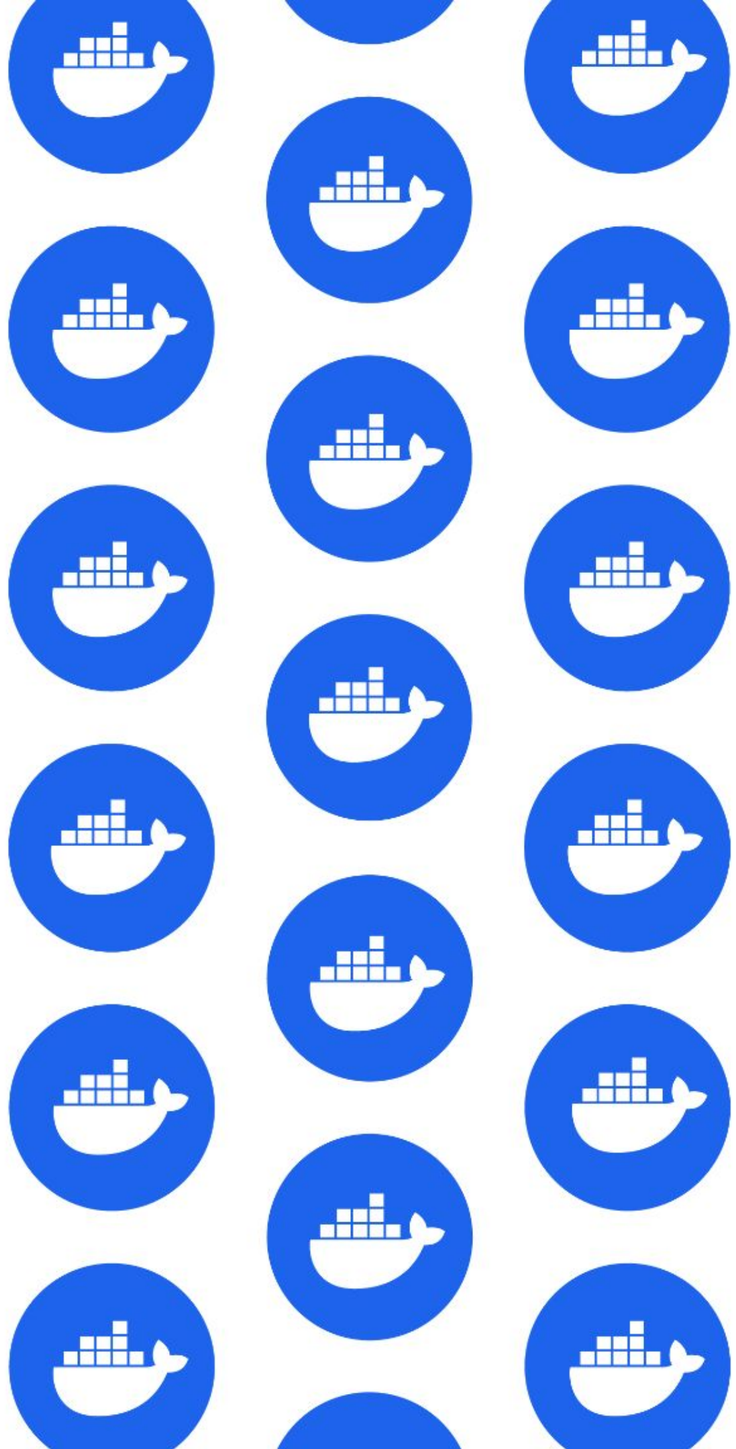
A few multi-service best practices

- Again… each container should do one thing and do it well
- Use service discovery to locate other dependencies
  - In Docker, that’s through the use of networks and network aliases
  - In Kubernetes, that’s through the use of `Service` objects
  - Many cloud provider container systems have DNS-based discovery mechanisms

HANDS-ON TIME!!!

Work on the “Multi-service applications” category

Work until 11 and take a break until 11:15

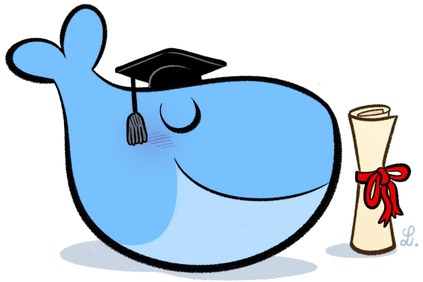


05

Intro to Docker Compose

This section's learning objectives

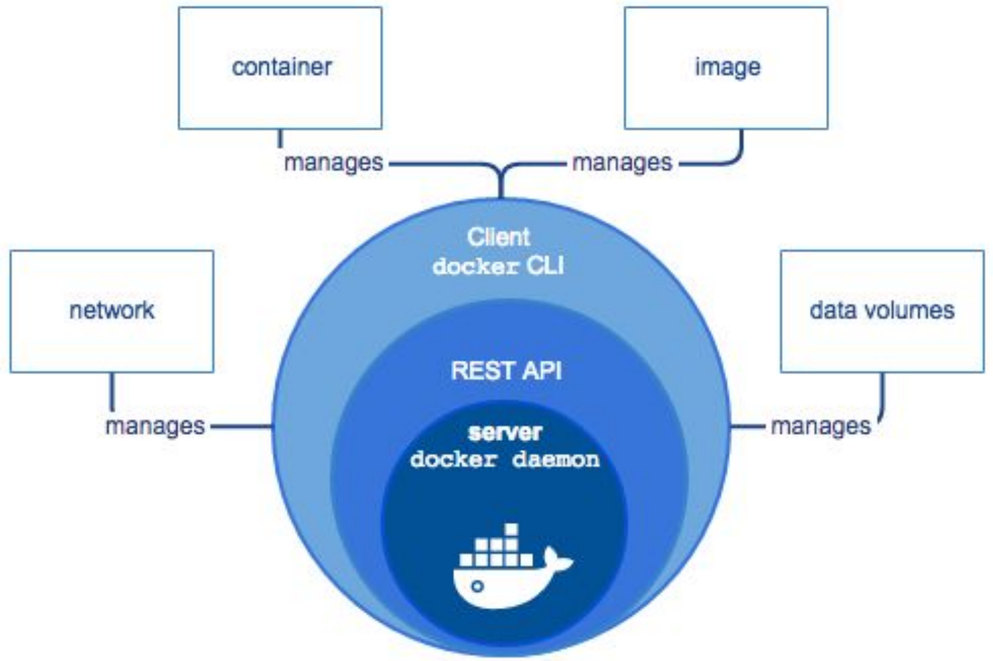
- By the end of this section, you'll be able to answer these questions:
  - How do the Docker CLI and Engine interact?
  - What is Docker Compose?
  - How do I use Compose to launch applications?
  - How do I create my own Compose file?
  - What new things are going on with Compose?



With networks, volumes, and the many flags of containers, it's got to be easier to define everything, right?

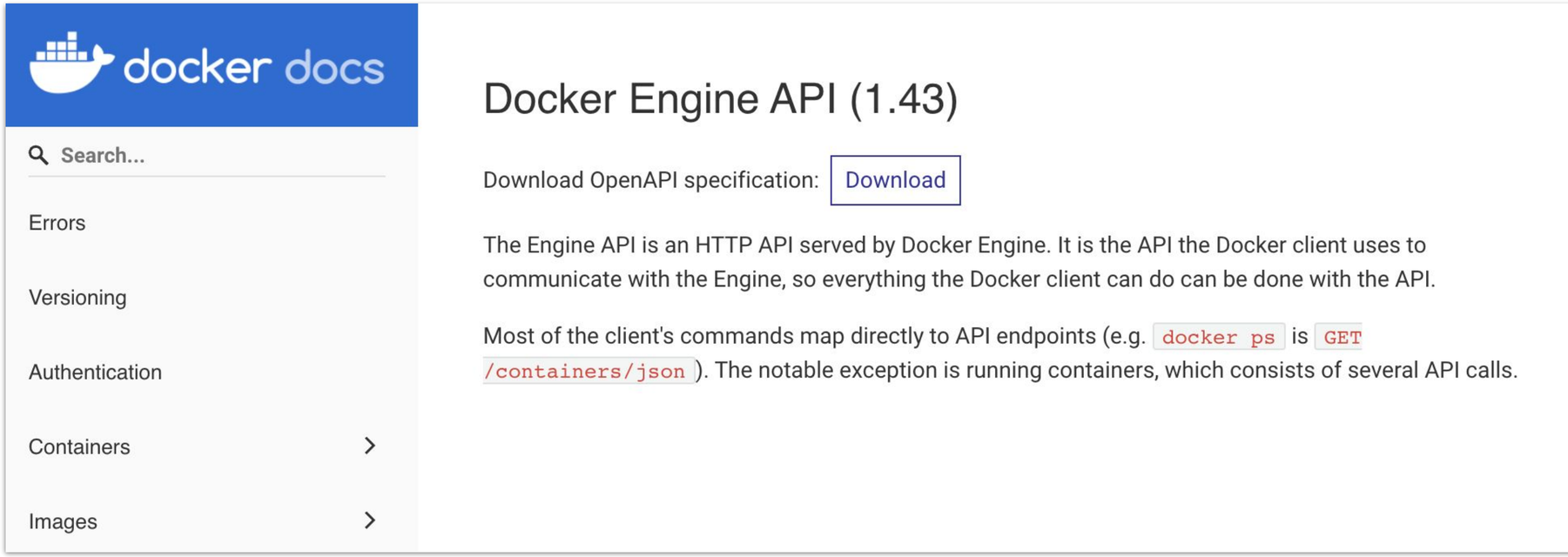
The Docker Engine

- The Docker Daemon/Engine
  - Manages everything
  - Exposes a REST API
- The Docker CLI
  - Interacts with the API
  - Provides a CLI-based user interface
  - Can be configured to point to remote engines

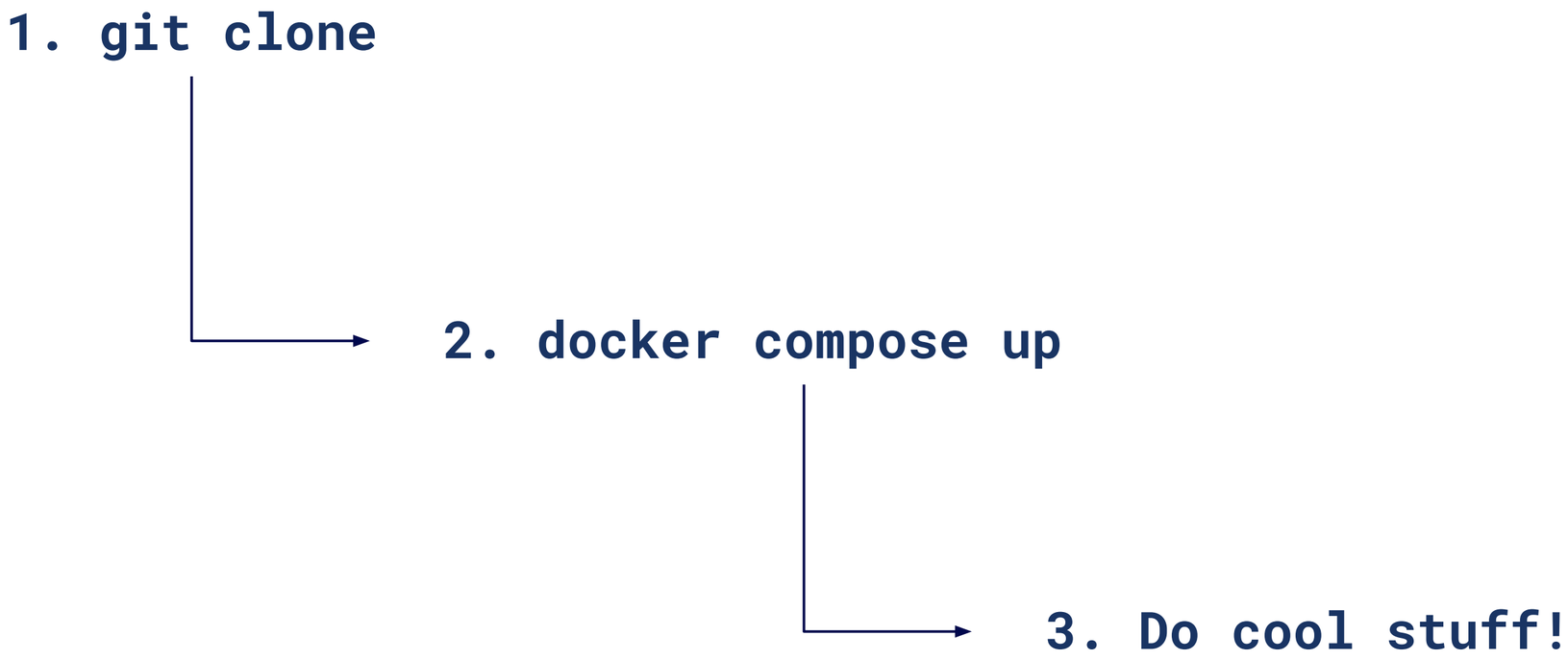


The Engine API

- The API is documented online
- The API provides the ability to do anything on Docker daemon
  - Provides ability to build other tooling/automations on top of the engine



The Goal



Basic Compose commands

```
# Use Compose to reconcile and launch what's defined
docker compose up

# Tear everything down (leaves volumes by default)
docker compose down

# Share logs from all of the application services
docker compose logs
```

Converting a `docker run` command

```
docker run -d -p 3306:3306 \
    -e MYSQL_ROOT_PASSWORD=test \
    -e MYSQL_DATABASE=dev \
    -v mysql-data:/var/lib/mysql \
    mysql:8.0
```

Converting a `docker run` command

```
docker run -d -p 3306:3306 \
    -e MYSQL_ROOT_PASSWORD=test \
    -e MYSQL_DATABASE=dev \
    -v mysql-data:/var/lib/mysql \
    mysql:8.0
```

```
services:
  db:
    image: mysql:8.0
```

The services represent each of the building blocks of my application.

I define a service named **db** and specify the container image it will use.

Converting a `docker run` command

```
docker run -d -p 3306:3306 \
    -e MYSQL_ROOT_PASSWORD=test \
    -e MYSQL_DATABASE=dev \
    -v mysql-data:/var/lib/mysql \
    mysql:8.0
```

Next, we bring over the port mapping.

The Compose specification has both a short-form syntax and a long-form syntax that's more verbose.

```
services:
  db:
    image: mysql:8.0
    ports:
    - 3306:3306
```

OR

```
services:
  db:
    image: mysql:8.0
    ports:
    - target: 3306
      published: 3306
```

Converting a `docker run` command

```
docker run -d -p 3306:3306 \
    -e MYSQL_ROOT_PASSWORD=test \
    -e MYSQL_DATABASE=dev \
    -v mysql-data:/var/lib/mysql \
    mysql:8.0
```

Next, we bring over the environment variables.

The Compose specification allows you to define them as either a key/value mapping or as an array of **KEY=VALUE** strings.

```
services:
  db:
    image: mysql:8.0
    ports:
    - 3306:3306
    environment:
      MYSQL_ROOT_PASSWORD: test
      MYSQL_DATABASE: dev
```

OR

```
services:
  db:
    image: mysql:8.0
    ports:
    - 3306:3306
    environment:
      - MYSQL_ROOT_PASSWORD=test
      - MYSQL_DATABASE=dev
```

Converting a `docker run` command

```
docker run -d -p 3306:3306 \
    -e MYSQL_ROOT_PASSWORD=test \
    -e MYSQL_DATABASE=dev \
    -v mysql-data:/var/lib/mysql \
    mysql:8.0
```

Next, we define the volume, ensuring our data is persisted across environment restarts

```
services:
  db:
    image: mysql:8.0
    ports:
    - 3306:3306
    volumes:
      - mysql-data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: test
      MYSQL_DATABASE: dev
volumes:
  mysql-data:
```

Let’s launch it!

Adding another service

- With Compose, it's easy to add other services to help with debugging and troubleshooting
- This new service is adds a database visualizer

```
services:
  db:
    ...
  phpmyadmin:
    image: phpmyadmin:5.2
    ports:
      - 8080:80
...
```

Using depends_on

- Ensures the configured service doesn't start until conditions are met
- The default condition is simple - the referenced service is up and running
- Other conditions provide for health checks passing or successful exits

```
services:
  db:
    ...
  phpmyadmin:
    ...
    depends_on:
      - db
...
```

```
services:
  db:
    ...
  phpmyadmin:
    ...
    depends_on:
      db:
        condition: service_healthy
...
```

Compose networking

- Each Compose stack gets its own network
- Each service has a DNS entry added that matches its name
  - The phpmyadmin service can connect to the database using the name **db**

```
services:
  db:
    ...
  phpmyadmin:
    image: phpmyadmin:5.2
    ports:
      - 8080:80
    depends_on:
      - db
    environment:
      PMA_HOST: db
      PMA_USER: root
      PMA_PASSWORD: test
...
```

Let’s launch it!

Compose Watch

- Watch for file changes and perform actions
  - Sync files from host to container
  - Auto rebuild images and restart containers
- Available with the new `docker compose watch` command

```
services:
  ...
  client:
    ...
    develop:
      watch:
        - path: ./client/src
          action: sync
          target: /usr/src/app/src
        - path: ./client/yarn.lock
          action: rebuild
...
```

Compose Watch

- Include Compose files from other locations for composable apps
  - Reference local files
  - Pull from remote git repos
- Each Compose file can reference their own relative paths

```
include:
  - another-compose.yaml
  - git@github.com:namespace/repo.git
services:
  ...
volumes:
  ...
```

06 Recap

What we covered

Container/image basics

What they are, why they matter, how to think about them. How they're different than VMs.

How to run containers and how to build your own images.

Multi-service applications

How to build container networks and use DNS to provide service discovery between containers.

Intro to Docker Compose

The tool that builds on the Docker Engine to help your team be super productive..


Welcome! Glad to have you on the team!
We already assigned you an easy first task. Challenge - see if you can knock it out by lunch!


Welcome! Glad to have you on the team!
Well done! We knew you had it in you and glad to hear the onboarding was easy! Again... welcome aboard!

Going forward

- Attend the keynotes and other sessions
- Join the Docker Community Slack (if you haven’t yet)
- Interact with the Docker roadmap (github.com/docker/roadmap)

Thank you!


dockercon 23