Docker for Robotics

How to use and create Docker containers

Presenter:
Corrie Van Sice
Research Scientist Associate
Texas Robotics, Nuclear and Applied Robotics Group

Join Zoom Meeting
https://utexas.zoom.us/j/96864105983
Meeting ID: 968 6410 5983

Workshop Resources

Go to the Github repository:

https://github.com/ut-texas-robotics/docker_workshop

- This Presentation
- Exercise Files and Docs
- Code

Format

1. **Presentation**
   - Introduce Core Concepts
2. **Demo**
   - Robot ROS Demo
3. **Workshop Exercises**
   - Install Docker Engine and Explore the CLI
   - Create a Custom Docker Image
   - Configure with Docker Compose
4. **Presentation & Discussion**
   - Development Strategies

Learning Goals

- What is Docker Engine and the CLI
- Use **images** and **containers**
- Write **Dockerfiles** to create images
- Use **docker compose** to build and launch projects
- Understand volumes, devices and network
- Interface with robot **hardware** and **ROS**
- Development strategies using git and scripting

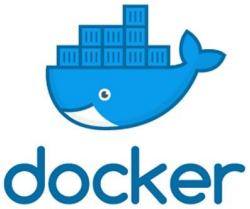
Why use Docker for Robotics?

Many dependencies are limiting without it.

- Multiple versions of ROS
- Multiple versions of Ubuntu
- Learning and Vision, multiple versions of CUDA
- Cutting Edge Hardware needs the latest OS

Archive your code so it works far in the future.

Maintain a working system state that resists unintended changes.

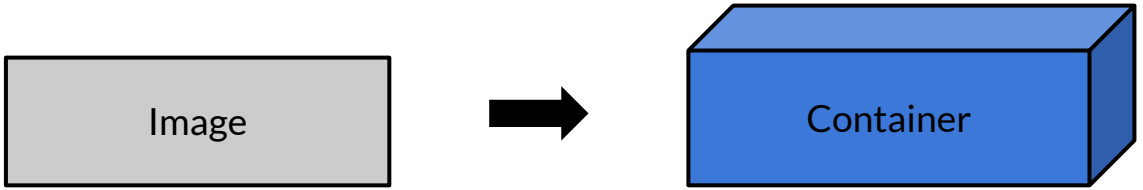


What is Docker?

Docker is a tool that allows you to package applications and their dependencies into lightweight containers. These containers run consistently across various environments.

Important Components:
Images, Containers, Daemon, CLI, Registry

Image vs Container



Read only, immutable

Describes a container

Composed of layers

Contains meta-data

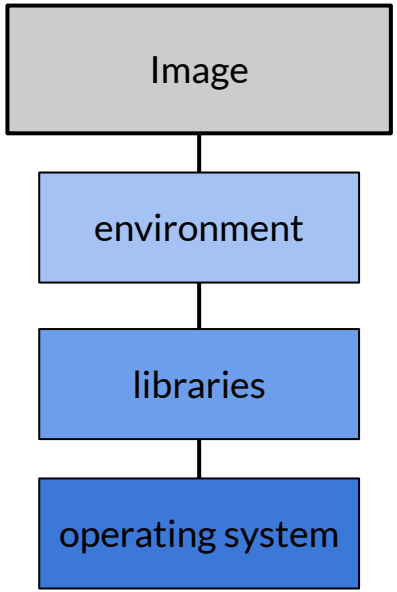
Read/Write, Changes state

Isolated environment

Can connect to volumes, networks, devices, etc.

Start, stop, remove

Docker Image



A Docker image is a **read-only** snapshot of a filesystem, comprising libraries, environment variables, and configurations needed to run an application's code. *The application code may or may not be included in the image.* Images can be downloaded from a registry, created from Dockerfiles or created from container states.

Where do images come from?

Docker Registries

A place where existing images are stored online.

Can be private or public.

Images are already built.

NRG Private Server

Dockerfiles

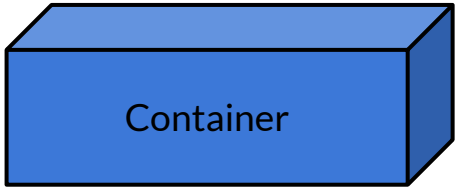
Files used to build new images.

Often included in git repos.

Image must be built.

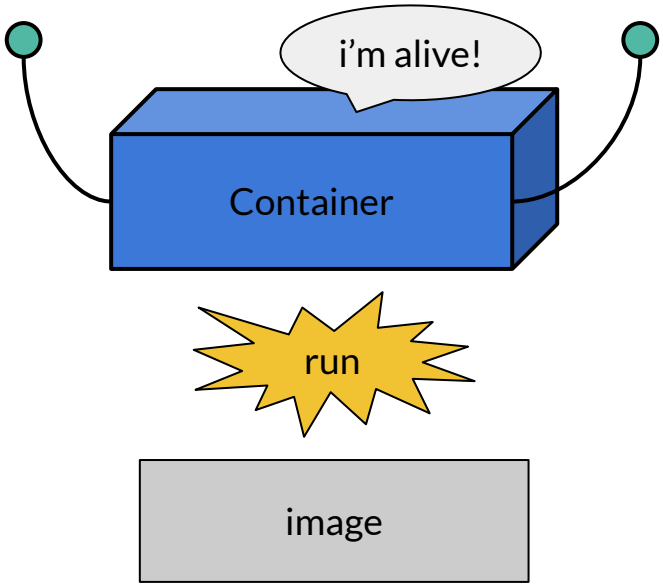
```
FROM nvidia/cuda:11.8.0-cudnn8-devel-ubuntu18.04

SHELL ["/bin/bash", "-c"]
RUN apt-get update &&\
        apt-get -y install nano tmux curl
WORKING_DIR /root
RUN git clone –recursive
        https://github.com/ut-texas-robotics/some_repo.git &&\
        cd some_repo &&\
        make
CMD [“/bin/bash”]
```

Docker Container



A container is a **runnable and read/writable** instance of a docker image. It is defined by an image and the configuration options you provide when you create it. A container is a highly isolated environment–you can run multiple containers simultaneously; connect containers to volumes, networks, and other subsystems on the host, as well as each other. **When a container is removed, any changes to its state that aren't stored in persistent storage disappear.**

Docker Container



A container is a **runnable and read/writable** instance of a docker image.

It is **configured on creation**, for example to connect to devices and volumes.

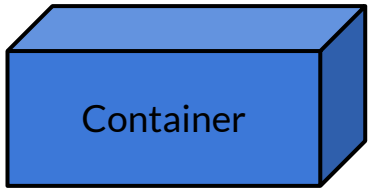
A **highly isolated** and configurable environment: you can run multiple containers simultaneously; connect containers to volumes, networks, and other subsystems, as well as each other.

When a container is removed, any changes to its state are removed.

Running Containers

start a container

```
docker run <image> [command]

commands:
      bash command
      script

examples:
docker run my_image
docker run my_image ./task_list.sh
```

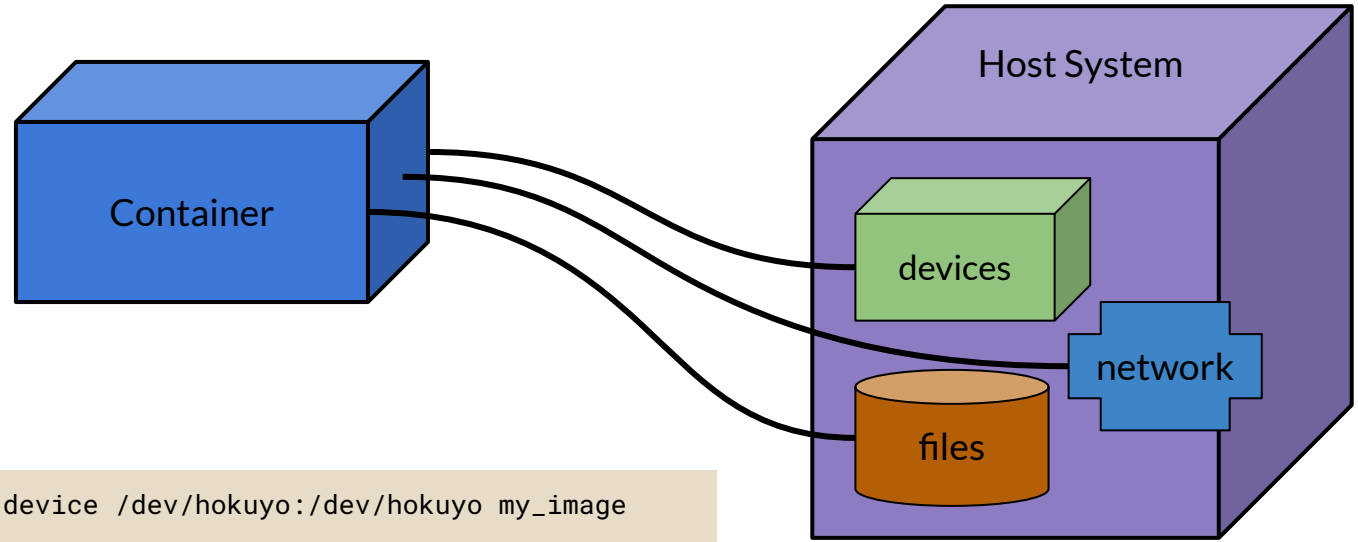


Containers are virtual environments with their own filesystem.

They often contain only the resources necessary to perform a certain function or run a certain application.

Container Configuration

Containers are isolated from the base system by default. They must be explicitly configured to have access to the host network, devices, and files.



example:

```
docker run --device /dev/hokuyo:/dev/hokuyo my_image
```

To Review...

Describes the steps to create a new image.

Read only, immutable

Composed of layers

Contains meta-data

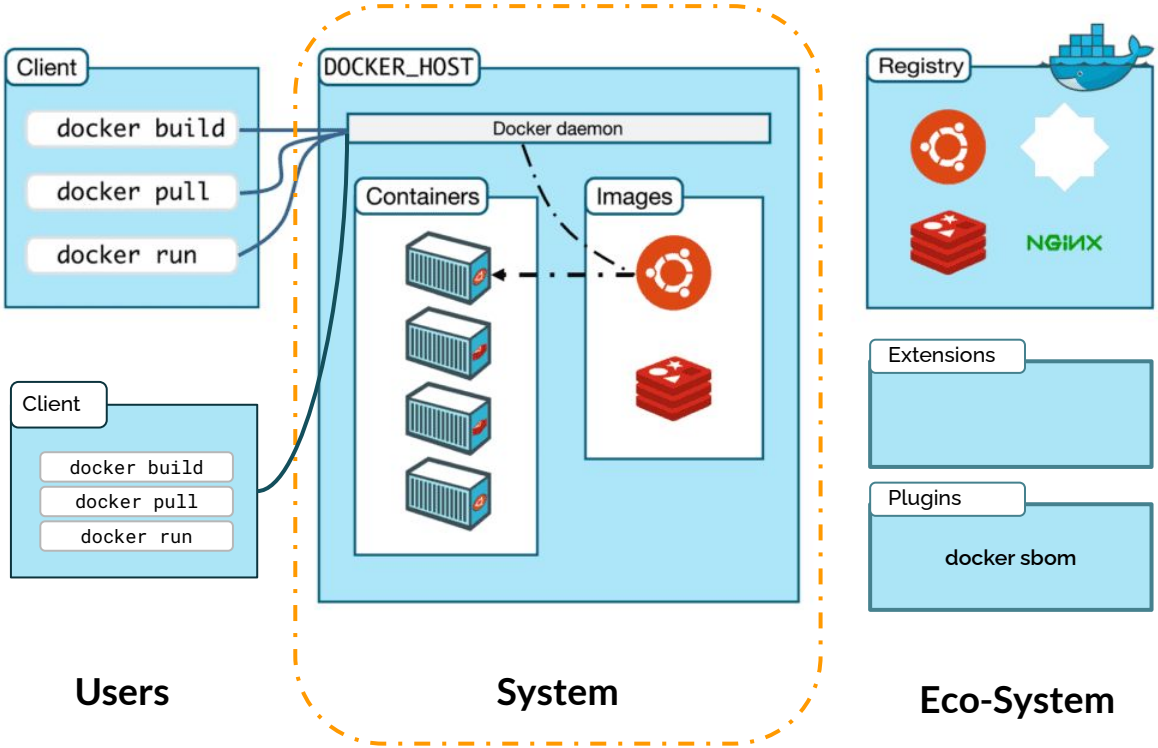
Read/Write, Changes state

Isolated environment

Configure to connect volumes, networks, devices, etc.

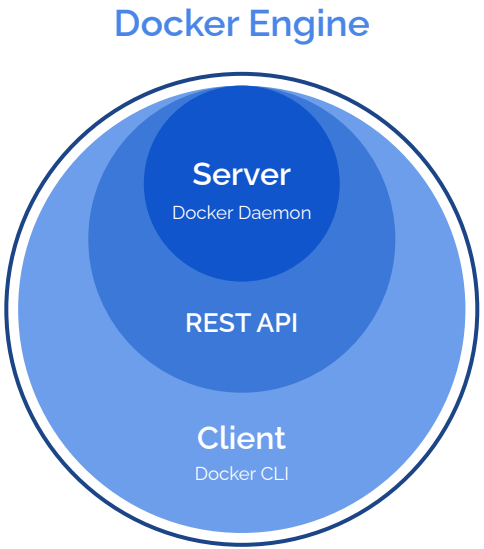
Start, stop, remove

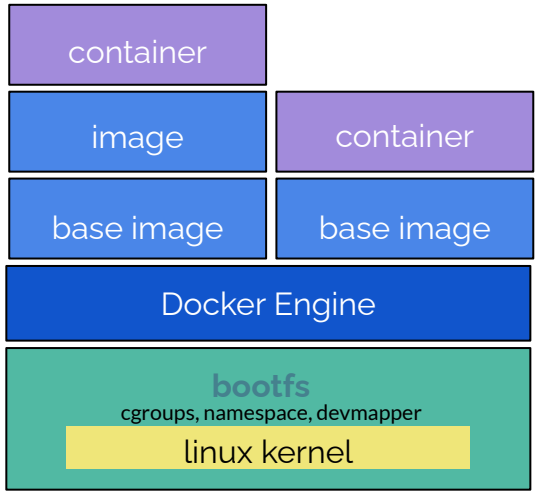
Docker Architecture



Docker Engine

Client/Server model: the server is exposed via a REST API, which clients use to make requests of the server. It interacts with a Linux Kernel host system via cgroups, namespaces, and device mapper. When you run a container, Docker creates a set of namespaces for that container.





Robot Demo

Running a BWIbot from Docker

Start a bot from a container.

https://github.com/utexas-bwi/bwi-docker

Hands-On

Install Docker Engine and Explore the CLI

Install Docker Engine

- Install Docker Engine on Ubuntu
- Install NVIDIA Container Toolkit
- Post-Installation Steps
- Software Bill of Materials plugin

Docker Commands

`docker pull`	pull an image from a registry
`docker sbom`	list the “software bill of materials” of an image
`docker image ls`	list the images on your system
`docker ps` `docker container ls`	list the running containers (use `-a` to list all containers)
`docker run`	create a container from an image
`docker exec`	execute a command in a running container
`docker stop`	stop a container
`docker container rm`	delete a container

Dockerfiles

Create new images

Docker images are essentially snapshots of a file system.

Dockerfiles are scripts used to build custom Docker images.

When building images, it's important to grasp the concept of layers and the build structure created by commands.

Images for Development

```
> Docker
    Dockerfile
    docker-compose.yml
    entrypoint.sh
    bash_utils.sh
    robot.env
```

Q: What goes in my Dockerfile?

A:

When writing an image for development environments, it can be helpful to consider the function of an image as building that environment, and not packaging an entire application or configuring all parameters. You can use other tools for configuration, like docker compose and entrypoint scripts.

Many docker users write images for containing entire applications and setting configs, and this is not wrong. It just depends on how the container is intended to be used.

Dockerfile Instructions

Images are built from Dockerfile instructions, such as:

FROM

RUN

COPY

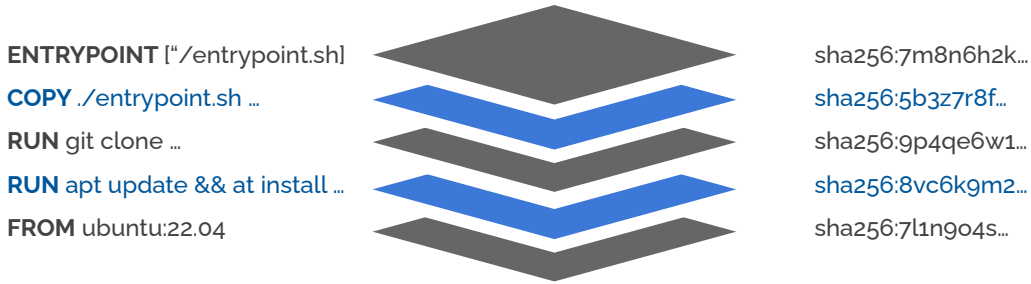
Each instruction is built as a layer in the image.

Dockerfile

```
FROM ubuntu:22.04

SHELL ["/bin/bash", "-c"]
RUN apt-get update &&\
       apt-get -y install nano tmux curl
WORKING_DIR /root
RUN git clone –recursive
       https://github.com/ut-texas-robotics/some_repo.git
       &&\
       cd some_repo &&\
       make
COPY ./my_file.sh /root/my_file.sh
CMD [“/bin/bash”]
```

Layers

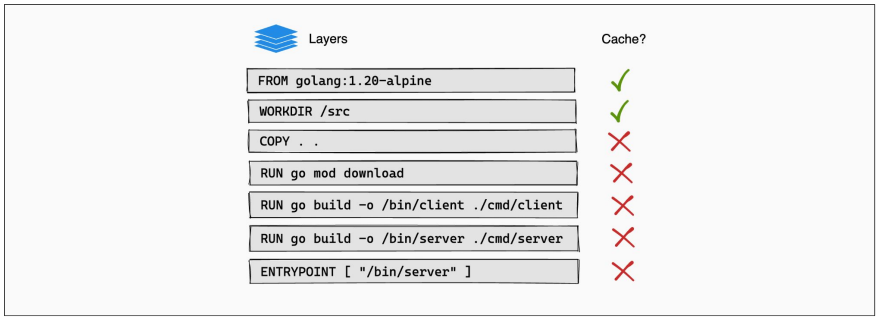
A Docker image is composed of multiple layers. Each layer represents a set of changes to the file system. This layering system allows for efficiency because when an image is modified, only the affected layers need to be updated, not the entire image. This makes image creation and distribution faster and more efficient.



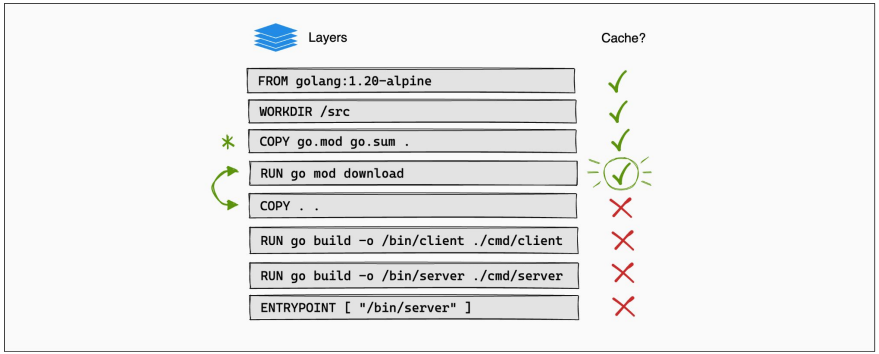
Cached layers

When you run a build, the builder attempts to reuse layers from earlier builds. If a layer of an image is unchanged, then the builder picks it up from the build cache. If a layer has changed since the last build, that layer, and all layers that follow are rebuilt.

changed files rebuild from where the change originated



organizing commands can save build time



Dockerfile Commands

FROM
- pull a base image from a registry, docker hub, or your own collection
- examples: ros, conda, ubuntu:22.04, etc.

RUN
- each of these commands is a layer
- RUN is not the same as doing an individual bash command
- RUN <command> or RUN ["executable", "param1", "param2"]

ENV
- these are environment variables you need in the build process, not necessarily in your config

SHELL
- set the default shell to use - if left undefined, its /bin/sh

COPY
- copy a file into the container, such as an entrypoint script

CMD
- CMD is not the same as RUN
- this is the command that is executed when you start a container.
- this does not contribute to building your image, it only provides a directive for what should be executed when you start a container from the image
- CMD ["executable","param1","param2"], or CMD <command>, or CMD ["param1", "param2"]

ENTRYPOINT
- this is a script that runs when you start a container, rather than a simple command
- it is copied into the image when you build it
- the benefit is that you can dynamically make changes to the container using the script
- the challenge is that you may need to pass parameters to it

Docker Compose

Configuring containers with a docker-compose.yaml

Setup the image build and the container configs.

Run multiple containers at once, such as a database and an app.

Configure volumes, devices, runtime, and more.

Build and Run with docker compose

Docker compose can play a part in both building images and running containers.

It simplifies container configuration using .yaml files to set parameters.

As a result, it shortens command line interactions:

- `docker compose build`
- `docker compose up`
- `docker compose down`

Services

Services are specific to docker compose. They define what image to use/create, the name of the container, and its configurations.

Multiple services can exist in a docker-compose file and dependencies can be set between them. For example, one service may setup a database container, while another creates a web app container that relies on the database.

Volumes

Volumes function in two ways generally:

- make system files available to the container to help it function
- allow persistent storage of files edited in a container

Volumes bind a resource on the host to a location in the container.

Host location	Container location	Purpose
`./project_dir`	`/root/project_dir`	persist project files
`/tmp/.X11-unix`	`/tmp/.X11-unix`	connect to host X11 Server

Devices

Makes devices available to the container, such as usb and bluetooth devices. Device names can be defined on a host with udev rules, or just mapped to a name.

Host location	Container location	Device
`/dev/hokuyo`	`/dev/hokuyo`	2D Lidar
`/dev/input/js0`	`/dev/input/js0`	PS4 Joystick
`/dev/USB0tty`	`/dev/robot_base`	A mobile base

Entrypoint

An entrypoint is a script that runs when the container starts. It offers a dynamic way to setup a container.

A common task in an entrypoint is creating a user in the container that matches the host user and setting file permissions.

Entrypoints can be set in Dockerfiles and also in docker-compose files.

Dev Methods

Using a container as a development environment.

Edit code on the host.

Run git commands on the host.

Build/run in the container.