Docker Takeaways

Author: William Roberts

Organization: Space Science and Engineering Center @ University of Wisconsin Madison

Purpose of this talk

Expose listeners to interesting and useful Docker concepts. My goal is to spark ideas and bring attention to as many tools and concepts as I can, without wasting time on specifics. We will NOT dive into in-depth on examples, and I do not expect you to understand every concept.

Examples are on this github repository: https://github.com/wroberts4/docker_take_aways

Intended audience

This talk is aimed at people who already have a very basic understanding of Docker. However, you don’t need to be familiar with it to take away useful information!

Introduction to Docker

Docker is an open platform for creating application "images", and running said application images in "containers". Docker makes it trivial to run software between systems and environments; if you have Docker, you can run the software! Personally, Docker is my favorite and most used development tool because allows me to save lots of time and effort in both development and deployment. Similar to how virtual environments (anaconda/mamba) simplifies package management and deployment, Docker simplifies system and application management and deployment.

Create an image

Dockerfile:

```
FROM centos:7
RUN yum -y install python3
COPY hello.txt /hello.txt
```

hello.txt:

```
hello world!
```

Command line:

```
$ docker build -t my_image .
$ docker run --rm -it centos:7 pwd
/
$ docker run --rm -it my_image cat hello.txt
hello world!
$ docker run --rm -it my_image python3 –version
Python 3.6.8
```

Create an image cont.

- Once an image is created, it is static and will not change unless remade. This is great for production environments
- Each Docker command increases the size of the final image
- Images can be ran between systems, with a few caveats listed below
- Containers run on their own filesystem, and do not persist between runs
- While images are the same between systems, there are a few things that may affect images and running containers:
  - CPU architecture: ARM vs amd64
  - System level resources, such as file descriptors: --ulimit nofile=10000:10000

Create an image cont.

- Since containers are running instances of images, you can create multiple containers of the same image all running at once. This allows you to scale certain processes very easily and efficiently
- Combined use cases for Docker:
  - Background processes: Run a docker container in the background using the “-d” flag
  - Screen sessions: Connect to a running docker container using “docker exec”
  - Virtual environments: Keep environments separate by using different images

Create an image cont.

- “docker build -f my_dockerfile” will tell Docker which Dockerfile to use. It defaults to the file named “Dockerfile” in the current working directory
- “docker build path/to/build/context” will tell Docker where to look for context specified in the build stage: COPY commands

Pre-built software

- CentOS
- Jupyter
- Java
- Conda/Mamba
- Nginx
- Docker in Docker
- Minio
- So many more: https://hub.docker.com

Dockerfile commands

- FROM: Creates a new stage with the given image.
- RUN: Execute the provided commands.
- COPY: Copy data from disk into image.
- ENV: Set an environment variable.
- ARG: Set a variable only usable when building image.
- USER: Set the default user.
- CMD: Set the default input.
- ENTRYPOINT: Set a wrapper script to run.
- WORKDIR: Set the default working directory.
- LABEL: Set metadata on the image.

Docker only Dockerfile commands

- The following requires “--format docker” on non-Docker systems
- ONBUILD: Triggers other commands when building from this image.
- SHELL: Set the default shell to use.

Multi-stage builds

Dockerfile:

```
FROM condaforge/mambaforge:4.12.0-0 as environment
RUN conda-pack -n ${ENV_NAME} -o /${ENV_NAME}.tar
RUN mkdir /${ENV_NAME}
RUN cd /${ENV_NAME} && tar xf /${ENV_NAME}.tar
RUN mamba install -y -p /${ENV_NAME} python xarray

FROM debian:bullseye-slim
COPY --from=environment /$ENV_NAME /$ENV_NAME
RUN echo "source /${ENV_NAME}/bin/activate" > ~/.bashrc
SHELL ["bash", "--login", "-c"]
RUN echo 'exec "$@"' > /entrypoint.sh
ENTRYPOINT ["bash", "--login", "/entrypoint.sh"]
ONBUILD COPY current_time.txt /current_time.txt
```

Multi-stage builds cont.

- The above example is a sudo-code example. Using this method, my image went from 1200 MB to 400 MB!
- Intermediate stages are discarded in the final image. This is great for reducing size and protecting sensitive information such as usernames and passwords
- Caching intermediate stages is a little clunky, but improvements are around the corner. Historically, you would build each stage individually and cache from each stage

Docker logs

- Docker automatically saves container stdout into logs
- You can control the size and number of log files by using the following flags upon starting a container: “--log-opt max-size=20m --log-opt max-file=3”. This is essential for non-stop running containers that produce output such as a web server to not take up infinite storage.
- The command, “docker logs” is used to access said logs. Example: “docker logs --since 1s --until 2s -f my_centos”

Useful command line flags

- --log-opt
- --memory
- --cpus
- --network
- -d
- --restart
- --privileged
- --name
- --cache-from
- --no-cache
- --user

Docker volumes

- Docker volumes are managed by docker, not the host filesystem. Thus, Docker volumes are not dependent on individual filesystems
- Docker volumes do not run into permission issues when mounting. They will run as the user inside the container. This is different from bind mounts (from local disk) which will only mount as root for some systems (linux).
- “docker volume create” is the command to create volumes manually. It has extra advanced options that are rarely used

Docker volumes cont.

- Docker volumes use a name, bind mounts use a path. If the volume does not exist, it is created:
  - “docker run -v my_volume:/data” uses a Docker volume
  - “docker run -v /data:/data” uses a bind mount
- You can pass additional options when mounting a directory or volume such as read only:
  - “docker run -v /data:/data:ro”

Docker networks

- Docker containers have different ways to access other systems/machines. By default, Docker containers are on their own network that can open up ports to the host machine.
- You can create a network, and have multiple containers communicate on said network: “docker network create”

Docker networks cont.

- “--network host” will set the network to the same as the host. This can be security threat, but is useful for development
- Another use case is to set the network to none, and this isolates the container for all outside systems; including the internet! This is great for testing software that pushes data to a remote location, while ensuring it doesn’t actually push: “docker run --rm -it --network none centos:7 ping google.com”

Docker deploy to registry

- Docker makes it simple to make images accessible. Once an image is built, you can push it to dockerhub, gitlab, or other hosting sites using “docker push”. The default image registry is dockerhub: “docker.io/library”
- Users can either use your image with “docker pull” or directly in their run commands.
- Again, these images are static and work across different systems.
- Larger images take longer to pull. Docker images are not logistically designed to be many gigabytes large. Typically, you should mount large data using volumes

Buildkit

- Buildkit is a new and improved way of building images. Docker has started to implement it using “docker buildx” and is the default in Docker Desktop (Mac/Windows). It includes plethora of features:
    - Parallel builds
    - Multi-stage caching
    - Caching to a directory, registry, or image-metadata
    - Multi-platform image builds (ARM, amd64, etc)
    - Cache directories between builds (pip’s download cache)
    - Pass sensitive data more generally and safely

Buildkit cont.

- For some setups, docker buildx may not be the default. Try setting “DOCKER_BUILDKIT=1”
- For non Docker Desktop users (linux), there’s options to download the buildkit command, or run the buildkit daemon in the background
- Skim the documentation and keep an eye out for cool features: https://docs.docker.com/engine/reference/commandline/buildx_build

Non-root images

Dockerfile:

```
FROM centos:7
RUN useradd --create-home -u 1000 user1
WORKDIR /home/user1
USER user1
```

Non-root images cont.

- The suggested and secure way to run Docker containers is as a non-root user.
- Currently when you map a local directory on linux systems, it is always mapped to the root user inside the container. This causes permission issues. The workaround is to use Docker volumes, object storage, or enable the docker daemon flag “--userns-remap” and “--userns=host”. Non linux Docker is more secure when it comes to UID mapping, but it is still best practice to run as non-root

Docker deploy to cloud

hello.yml:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-world
        image: centos:7
        stdin: true
        tty: true
```

Docker deploy to cloud cont.

- Docker images can be deployed in the cloud (kubernetes).
- There are many more applications (and with them complications) that come with cloud deployment which are out of scope of this talk
- Common cloud providers: AWS, Google Cloud, Microsoft Azure. These are all paid providers, but each has decent free-tiers to get started. Moving to the cloud can be more dynamic and/or affordable than building your own infrastructure from the ground up