Optimizing Docker images

INTERMEDIATE DOCKER

Mike Metzger
Data Engineering Consultant

Docker image explanation

- Docker images are the base of a given container
- Holds all content initially available to a container instance

Docker image concerns

- Tempting to add all potentially needed components to an image
- Size becomes large / unwieldy
- Difficult to handle security / updates due to dependency issues
- Harder to combine containers without wasting space / bandwidth

Docker image recommendations

- Split containers to the smallest level needed
- Easier to combine multiple containers later vs. building a single large image
- Like
  - building with reusable components
  - vs. building from scratch each time
- Updates to specific software only affect containers using that image instead of all containers needing the update
- Can optimize for size, making use and distribution much easier

Docker image breakdown example

- Consider a data engineering project using the following software:
  - Postgresql database
  - Python ETL software
  - Web server software
- Possible to use a single image, but we would need to update the image each time we had an update to the ETL or web server setup.
- What would happen if we needed to add another web server?

```
FROM ubuntu
RUN apt update
RUN apt install -y postgresql
RUN apt install -y nginx
RUN apt install -y python3.9
...
```

Example with minimized containers

- Better options with Docker
- Split each into its own container
  - Postgresql database container
  - Python ETL components
  - Web server
- Can build an optimized configuration for our use, and can add / remove components as needed

```
bash> docker run -d postgresql:latest
bash> docker run -d nginx:latest
...
```

Determining image size

- Using `docker images`
- Shows individual image details, including size
- More in-depth options covered later

```
bash> docker images
```

```
REPOSITORY        TAG                  SIZE
postgres          latest               448MB
postgres          15                   442MB
apache/airflow    2.7.1-python3.9      1.4GB
alpine            latest               7.73MB
```

Let's practice!

INTERMEDIATE DOCKER

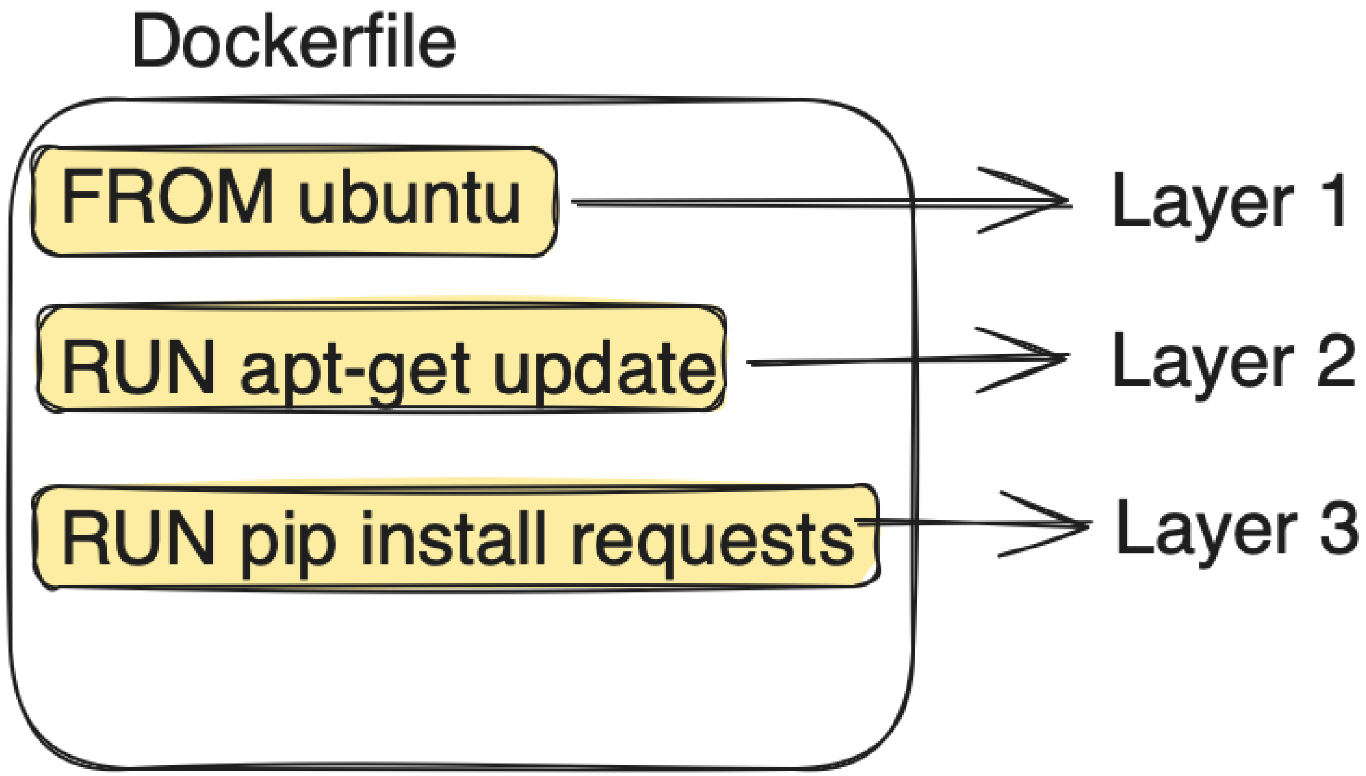
Understanding layers

INTERMEDIATE DOCKER

Mike Metzger
Data Engineering Consultant

Docker layers

- Docker images are made up of layers
- A layer generally references a change or command within a Dockerfile
- Layers can be cached / reused
- The order of commands within a Dockerfile can affect whether layers are reused



Why do we care about layers?

- Reusability
  - Faster build time
  - Smaller builds

docker image inspect

- How to determine the layers within an image?
- `docker image inspect <img id | name>`
- Provides much information about the content of a Docker image
- The `RootFS:Layers` section provides details about layers in a given Docker image

```
repl@host:~$ docker image inspect alpine
[
    {        "Id": "sha256:05455a08881ea9cf0e752bc48e61bbd71a34c029bb13df01e40e3e70e
        "RepoTags": [
            "alpine:latest"
        ],
        "Created": "2024-01-27T00:30:48.743965523Z",
```

docker image inspect example

```
bash> docker image inspect postgres:latest
```

```
"RootFS": {
   "Type": "layers",
   "Layers": [
"sha256:6f2d01c02c30cc1ffac781aff795cba8eeb29cc27756fe37bf525169856369c6",
"sha256:c6ad2d5a3cad837ae66b5560e9c577bfad062556b1f00791d8d733ce44a577ce",
"sha256:2153552a84ccbf7e4a28a50e766b72345072e59f8af0ff068baf98b413132e0c",
"sha256:6c00217b1e4b15c25eb3f6e28b1af8c295f469568014621e31a4c5eb5a8aca6f",
"sha256:167177d78e2a33aa822faebe9f01683c648ae78179059db05cd25737f215c305",
...
```

jq command-line tool

- Sometimes difficult to analyze the results from `docker image inspect`
- `jq` commandline tool is used to read JSON data, like what's returned from `docker image inspect`
- Can use `jq` to query data

jq recipes with Docker

- Method to see just a specific section, for example the `RootFS` data:
  - `docker image inspect <id> | jq '.[0] | .RootFS'`

```
{
"Type": "layers",
"Layers": [ "sha256:0f5c115c5eea96...",
            "sha256:20792593831cdc..."
          ]
}
```

jq recipes with Docker (part 2)

- Method to count number of layers using `jq`:
  - `docker image inspect <id> | jq '.[0] | {LayerCount: .RootFS.Layers | length}'`

```
{
  "LayerCount": 2
}
```

Let's practice!

INTERMEDIATE DOCKER

Multi-stage builds

INTERMEDIATE DOCKER

Mike Metzger
Data Engineering Consultant

Single-stage builds

- Typical Docker images are created using a single `FROM` command
- Each addition to the source image adds space and makes its management
- Consider an application that must be compiled prior to use
  - You can add all the necessary components to the image, compile it, and then configure the final image for use
  - This often leaves superfluous content in the image even if it is not used

```
FROM ubuntu
RUN apt update
RUN apt install gcc -y
...
RUN make
CMD ["data_app"]
```

Multi-stage builds

- Multi-stage builds use multiple containers
- Typically has one or more build stages
- Final components are copied into a final container image
- The build stages are then removed automatically
  - Saving space and minimizing the size of the container image
- Uses some additional syntax in the Dockerfile
  - `AS <alias>`
  - `COPY --from=<alias>`

Multi-stage build example

```
# Create initial build stage
FROM ubuntu AS stage1
# Install compiler and compile code
RUN apt install gcc -y
...
RUN make

# Start new stage to create final image
FROM alpine-base
# Copy from first stage to final
COPY --from=stage1 /data_app /data_app
# Run application on container start
CMD ["data_app"]
```

Let's practice!

INTERMEDIATE DOCKER

Multi-platform builds

INTERMEDIATE DOCKER

Mike Metzger
Data Engineering Consultant

Multi-platform?

- What does multi-platform mean?
  - Different OS types
    - `linux`
    - `windows`
    - `macos`
  - Different CPU types
    - `x64_64` or `amd64`
    - `arm64`
    - `arm7`
- Usually referred to as `os/cpu` , such as `linux/amd64`

Creating multi-platform builds

- Is built on multi-stage build behavior
- The initial / build stage tends to use cross-compilers and relies on the architecture of the host system
- Final stage uses the architecture / OS for the intended target.

Multi-platform Dockerfile options

- Build stage uses the `--platform=$BUILDPLATFORM` flag
  - `$BUILDPLATFORM` represents the platform of the host running the build
- Sometimes uses the `ARG` directive
  - Passes local environment variables into the Docker build system
  - In this case, `TARGETOS` and `TARGETARCH`
  - `ARG TARGETOS TARGETARCH`
  - The environment variables at the host level can be defined previously or using the `env` command.

Multi-platform example

```
# Initial stage, using local platform
FROM --platform=$BUILDPLATFORM golang:1.21 AS build
# Copy source into place
WORKDIR /src
COPY . .
# Pull the environment variables from the host
ARG TARGETOS TARGETARCH
# Compile code using the ARG variables
RUN env GOOS=$TARGETOS GOARCH=$TARGETARCH go build -o /final/app .

# Create container and load the cross-compiled code
FROM alpine
COPY --from=build /final/app /bin
```

Building a multi-platform build

- To create a multi-platform build, instead of using `docker build`, we must use `docker buildx` with assorted options
- `docker buildx` provides more commands and capabilities over `docker build`, including the option to specify a platform

```
docker buildx build --platform linux/amd64,linux/arm64 -t multi-platform-app .
```

- Prior to running the build, we must also have a new *builder* container present. This is done with the `docker buildx create --bootstrap --use` command.

Let's practice!

INTERMEDIATE DOCKER