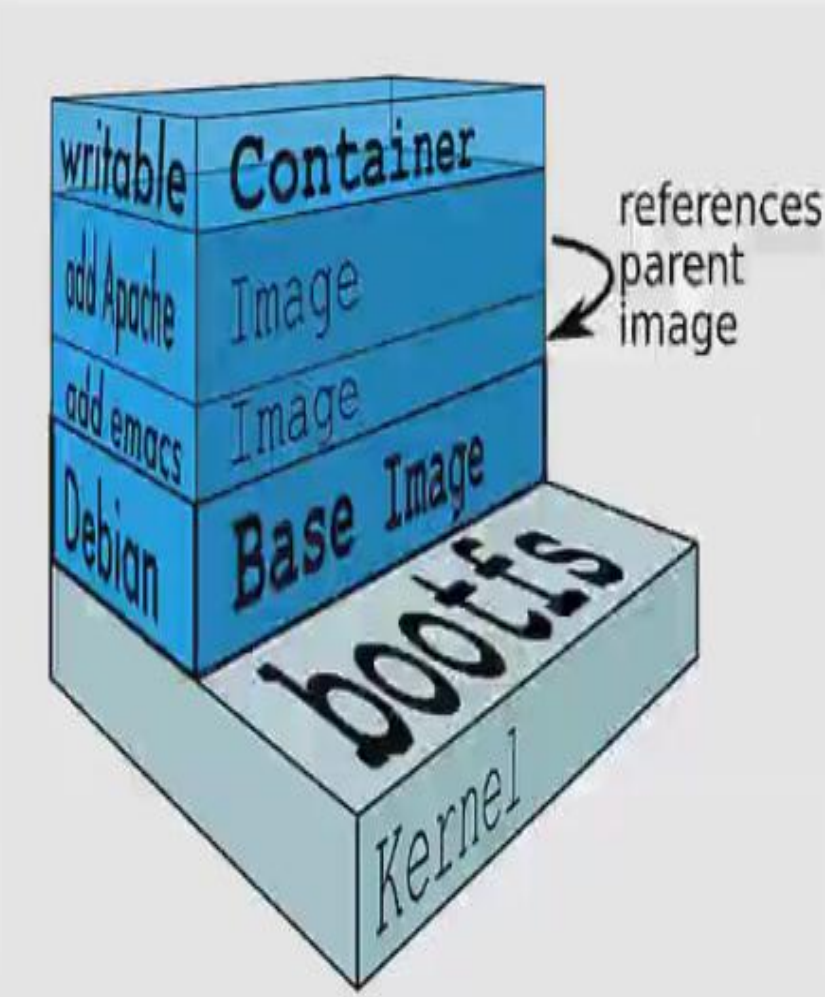
Docker Fundamentals

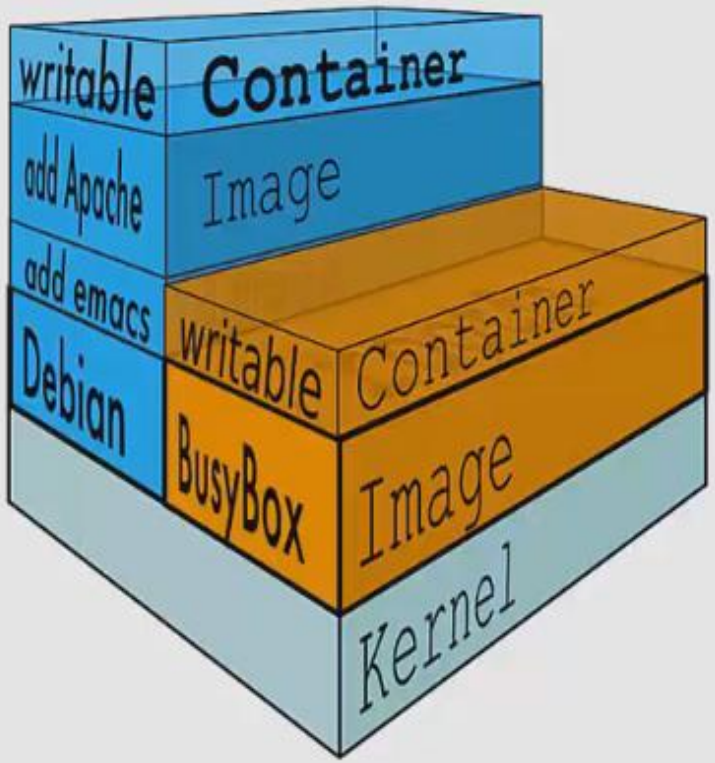
Image Layers

- Images are comprised of multiple layers
- A layer is also just another image
- Every image contains a base layer
- Docker uses a copy on write system
- Layers are read only



The Container Writable Layer

- Docker creates a top writable layer for containers
- Parent images are read only
- All changes are made at the writeable layer

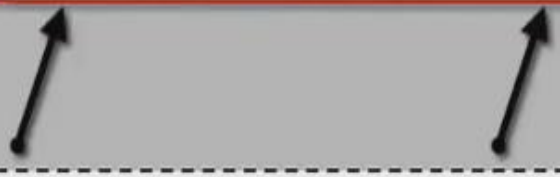


Docker Commit

- **docker commit** command saves changes in a container as a new image
- Syntax
  `docker commit [options] [container ID] [repository:tag]`
- Repository name should be based on username/application
- Can reference the container with container name instead of ID

Save the container with ID of 984d25f537c5 as a new image in the repository johnnytu/myapplication. Tag the image as 1.0

```
docker commit 984d25f537c5 username/application 1.0
```



User name Application name

Build New Image

EXERCISE

1. Create a container from an Ubuntu image and run a bash terminal
   `docker run -i -t ubuntu:14.04 /bin/bash`
2. Inside the container, install curl
   `apt-get install curl`
3. Exit the container terminal
4. Run `docker ps -a` and take note of your container ID
5. Save the container as a new image. For the repository name use <your name>/curl. Tag the image as 1.0
   `docker commit <container ID> <yourname>/curl:1.0`
6. Run `docker images` and verify that you can see your new image

Use New Image

EXERCISE

1. Create a container using the new image you created in the previous exercise. Run /bin/bash as the process to get terminal access

```
docker run -i -t <yourname>/curl:1.0 /bin/bash
```

2. Verify that curl is installed

```
which curl
```

Intro to Dockerfile

*A **Dockerfile** is a configuration file that contains instructions for building a Docker image*

- Provides a more effective way to build images compared to using `docker commit`
- Easily fits into your continuous integration and deployment process

Dockerfile Instructions

- Instructions specify what to do when building the image
- **FROM** instruction specifies what the base image should be
- **RUN** instruction specifies a command to execute

```
#Example of a comment
FROM ubuntu:14.04
RUN apt-get install vim
RUN apt-get install curl
```

Run Instruction

- Each RUN instruction will execute the command on the top writable layer and perform a commit of the image
- Can aggregate multiple RUN instructions by using "&&"

```
RUN apt-get update && apt-get install -y \
      curl \
      vim \
      openjdk-7-jdk
```

Docker Build

- Syntax
  `docker build [options] [path]`  build context
- Common option to tag the build
  `docker build -t [repository:tag] [path]`

Build an image using the current folder as the context path. Put the image in the johnnytu/myimage repository and tag it as 1.0

```
docker build -t johnnytu/myimage:1.0 .
```

As above but use the myproject folder as the context path

```
docker build -t johnnytu/myimage:1.0 myproject
```

Build from Dockerfile

EXERCISE

1. In your home directory, create a folder called test
2. In the test folder, create a file called “Dockerfile”
3. In the file, specify to use Ubuntu 14.04 as the base image
   ```
   FROM ubuntu:14.04
   ```
4. Write an instruction to install curl and vim after an apt-get update
   ```
   RUN apt-get update && apt-get install -y curl \
   vim
   ```
5. Build an image from the Dockerfile. Give it the repository `<yourname>/testimage` and tag it as 1.0
   ```
   docker build -t johnnytu/testimage:1.0 .
   ```
6. Create a container using your newly built image and verify that curl and vim are installed

CMD Instruction

- CMD defines a default command to execute when a container is created
- CMD performs no action during the image build
- Shell format and EXEC format
- Can only be specified once in a Dockerfile
- **Can be overridden at run time**

Shell format

```
CMD ping 127.0.0.1 -c 30
```

Exec format

```
CMD ["ping", "127.0.0.1", "-c", "30"]
```

Try CMD

EXERCISE

1. Go into the test folder and open your Dockerfile from the previous exercise
2. Add the following line to the end
   ```
   CMD ["ping", "127.0.0.1", "-c", "30"]
   ```
3. Build the image
   ```
   docker build -t <yourname>/testimage:1.1 .
   ```
4. Execute a container from the image and observe the output
   ```
   docker run <yourname>/testimage:1.1
   ```
5. Execute another container from the image and specify the echo command
   ```
   docker run <yourname>/testimage:1.1 echo "hello world"
   ```
6. Observe how the container argument overrides the CMD instruction

ENTRYPOINT Instruction

- Defines the command that will run when a container is executed
- Run time arguments and CMD instruction are passed as parameters to the ENTRYPOINT instruction
- Shell and EXEC form
- EXEC form preferred as shell form cannot accept arguments at run time
- Container essentially runs as an executable

```
ENTRYPOINT ["ping"]
```

Start and Stop Containers

- Find your containers first with **`docker ps`** and note the ID or name
- **`docker start`** and **`docker stop`**

List all containers

```
docker ps -a
```

Start a container using the container ID

```
docker start <container ID>
```

Stop a container using the container ID

```
docker stop <container ID>
```

Getting terminal access

- Use **`docker exec`** command to start another process within a container
- Execute /bin/bash to get a bash shell
- `docker exec -i -t [container ID] /bin/bash`
- Exiting from the terminal will not terminate the container

```
johnnytu@new-docker:~$ docker exec -it serene_shockley bash
root@4cfbac7ba80a:/usr/local/tomcat# cd
root@4cfbac7ba80a:~# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  3 04:57 ?        00:00:03 /usr/bin/java
root        34     0  0 04:59 ?        00:00:00 bash
root        40    34  0 04:59 ?        00:00:00 ps -ef
root@4cfbac7ba80a:~#
```

Deleting Containers

- Can only delete containers that have been stopped
- Use **`docker rm`** command
- Specify the container ID or name

Deleting local Images

- Use `docker rmi` command
- `docker rmi [image ID]`
  or
  `docker rmi [repo:tag]`
- If an image is tagged multiple times, remove each tag

Docker Hub Repositories

- Users can create their own repositories on Docker Hub
- Public and Private
- Push local images to a repository

Pushing Images to Docker Hub

- Use **`docker push`** command
- Syntax
  `docker push [repo:tag]`
- Local repo must have same name and tag as the Docker Hub repo

Push to Docker Hub

EXERCISE

1. Login to your Docker Hub account
2. Create a new public repository called “testexample”
3. Tag your local image to give it the same repo name as the repository you created on Docker Hub

```
docker tag <yourname>/testimage:1.1
<yourname>/testexample:1.1
```

4. Push the new image to Docker Hub

```
docker push <yourname>/testexample:1.1
```

5. Go to your Docker Hub repository and check for the tag

Volumes

*A **Volume** is a designated directory in a container, which is designed to persist data, independent of the container's life cycle*

- Volume changes are excluded when updating an image
- Persist when a container is deleted
- Can be mapped to a host folder
- Can be shared between containers

Mount a Volume

- Volumes are mounted when creating or executing a container
- Can be mapped to a host directory
- Volume paths specified must be absolute

Execute a new container and mount the folder /myvolume into its file system

```
docker run -d -P -v /myvolume nginx:1.7
```

Execute a new container and map the /data/src folder from the host into the /test/src folder in the container

```
docker run -i -t -v /data/src:/test/src nginx:1.7
```

Volumes in Dockerfile

- VOLUME instruction creates a mount point
- Can specify arguments JSON array or string
- Cannot map volumes to host directories
- Volumes are initialized when the container is executed

String example

```
VOLUME /myvol
```

String example with multiple volumes

```
VOLUME /www/website1.com /www/website2.com
```

JSON example

```
VOLUME ["myvol", "myvol2"]
```

Uses of volumes

- De-couple the data that is stored from the container which created the data
- Good for sharing data between containers
  - Can setup a data containers which has a volume you mount in other containers
- Mounting folders from the host is good for testing purposes but generally not recommended for production use

Mapping ports

- **Recall:** containers have their own network and IP address
- Map exposed container ports to ports on the host machine
- Ports can be manually mapped or auto mapped
- Uses the `-p` and `-P` parameters in **`docker run`**

Maps port 80 on the container to 8080 on the host

```
docker run -d -p 8080:80 nginx:1.7
```

Automapping ports

- Use the `-P` option in **`docker run`**
- Automatically maps exposed ports in the container to a port number in the host
- Host port numbers used go from 49153 to 65535
- Only works for ports defined in the EXPOSE instruction

Auto map ports exposed by the NGINX container to a port value on the host

```
docker run -d -P nginx:1.7
```

EXPOSE instruction

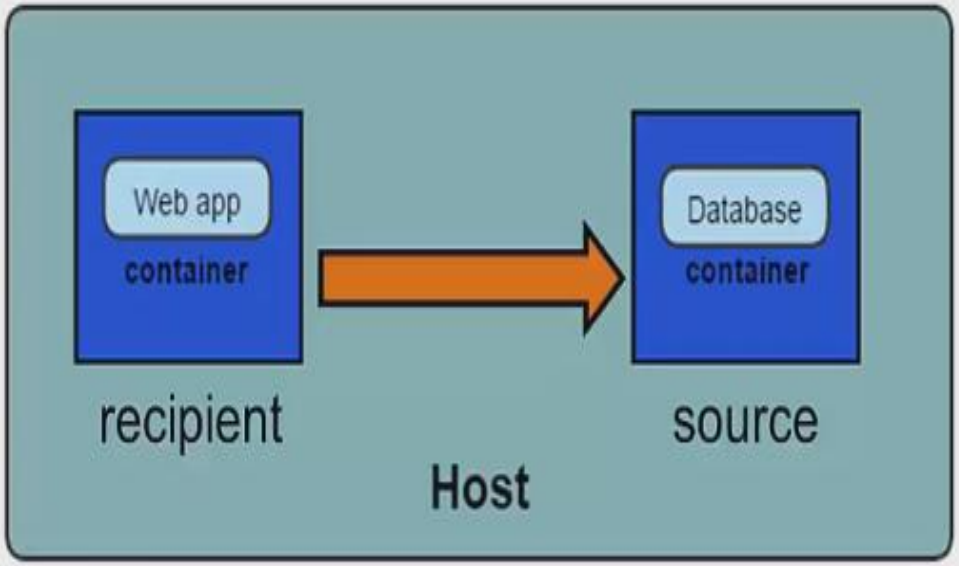
- Configures which ports a container will listen on at runtime
- Ports still need to be mapped when container is executed

```
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y nginx

EXPOSE 80 443

CMD ["nginx", "-g", "daemon off;"]
```

Linking Containers

Linking *is a communication method between containers which allows them to securely transfer data from one to another*

- Source and recipient containers
- Recipient containers have access to data on source containers
- Links are established based on container names



Creating a Link

1. Create the source container first
2. Create the recipient container and use the `--link` option

- **Best practice –** give your containers meaningful names

Create the source container using the postgres

```
docker run -d --name database postgres
```

Create the recipient container and link it

```
docker run -d -P --name website --link database:db nginx
```