Docker

Docker is a platform for developing, shipping, and running applications in containers. Containers are lightweight, portable, and self-sufficient units that can run applications and their dependencies in isolated environments. Docker provides a consistent and reproducible environment across different machines, making it easier to manage and deploy applications.

Here are the key components of Docker:

1. **Docker Engine:** The core of Docker is the Docker Engine, which is responsible for building, running, and managing containers. It consists of a server (daemon) and a REST API that allows you to interact with the daemon.

2. **Docker Images:** Images are lightweight, standalone, and executable packages that include everything needed to run a piece of software, including the code, runtime, libraries, and system tools. Images are used to create containers.

3. **Docker Containers:** Containers are instances of Docker images that run in isolated environments. They encapsulate an application and its dependencies, ensuring consistency across different environments. Containers are portable and can run on any system that supports Docker.

Now, let's look at some common Docker commands and their usage:

1. Docker Build

- **Command:**

  ```
  docker build -t image_name:tag .
  ```

- **Usage:** Builds a Docker image from a Dockerfile in the current directory.

2. Docker Run

- **Command:**

  ```
  docker run [options] image_name:tag
  ```

- **Usage:** Creates and runs a container based on a specified image.

3. Docker Pull

- **Command:**

  ```
  docker pull image_name:tag
  ```

- **Usage:** Downloads a Docker image from a registry.

4. Docker Push

- **Command:**

  ```
  docker push image_name:tag
  ```

- **Usage:** Uploads a Docker image to a registry.

5. Docker PS

- **Command:**

  ```
  docker ps [options]
  ```

- **Usage:** Lists the currently running containers.

6. Docker Images

- **Command:**

  ```
  docker images [options]
  ```

- **Usage:** Lists all available Docker images on the local machine.

7. Docker Stop

- **Command:**

  ```
  docker stop container_id
  ```

- **Usage:** Stops a running container.

8. Docker RMI (Remove Image)

- **Command:**

  ```
  docker rmi image_name:tag
  ```

- **Usage:** Removes a Docker image from the local machine.

9. Docker RM (Remove Container)

- **Command:**

  ```
  docker rm container_id
  ```

- **Usage:** Removes a stopped container.

10. Docker Exec

- **Command:**

  ```
  docker exec [options] container_id command
  ```

- **Usage:** Executes a command inside a running container.

Dockerfiles [Single stage & Multistage]

A Dockerfile is a script that contains a set of instructions for building a Docker image. Docker images are lightweight, standalone, and executable packages that include everything needed to run a piece of software, including the code, runtime, libraries, and system tools. Dockerfiles are used to automate the process of creating Docker images.

Here's a basic example of a Dockerfile:

```
# Use an official base image
FROM ubuntu:20.04

# Set the working directory
WORKDIR /app

# Copy application code into the container
COPY . .

# Install dependencies
RUN apt-get update && \
    apt-get install -y python3 && \
    apt-get clean

# Expose a port
EXPOSE 8080

# Define the command to run the application
CMD ["python3", "app.py"]
```

Now, let's break down each part of this Dockerfile:

1. `FROM`: Specifies the base image. In this case, it's Ubuntu 20.04.
2. `WORKDIR`: Sets the working directory inside the container. Subsequent instructions will be executed from this directory.
3. `COPY`: Copies files from the host machine to the container. Here, it copies the content of the current directory (the "." represents the current directory) into the "/app" directory inside the container.
4. `RUN`: Executes commands during the image build process. In this example, it updates the package list, installs Python3, and cleans up the package cache.
5. `EXPOSE`: Informs Docker that the container will listen on the specified network ports at runtime. It doesn't actually publish the ports, but it is a helpful documentation for other developers.
6. `CMD`: Provides the default command to run when the container starts. It specifies the command to run our application (in this case, a Python script).

Now, let's move on to a multi-stage Dockerfile example. Multi-stage builds allow you to use multiple `FROM` statements in a single Dockerfile, resulting in a smaller and more efficient final image.

```
# Stage 1: Build stage
FROM node:14 as build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Production stage
FROM nginx:1.21
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

In this example:

1. The first stage (`build`) uses the Node.js image to build a JavaScript/Node.js application. It copies the necessary files, installs dependencies, and runs the build process.
2. The second stage uses the smaller Nginx image and copies the output from the `build` stage into the appropriate directory. This results in a final image containing only the compiled application and the Nginx server.

This approach helps reduce the size of the final image by excluding unnecessary build dependencies from the production stage. The `--from=build` flag in the `COPY` command allows copying files from a specific build stage.

Docker Networking

Docker provides a networking model that allows containers to communicate with each other and with the outside world. Docker networking involves creating networks, connecting containers to networks, and managing the flow of traffic between containers. Here are some Docker networking concepts along with examples and commands:

1. Docker Bridge Network:

- **Create a Bridge Network:**

  ```
  docker network create my_bridge_network
  ```

- **List Docker Networks:**

  ```
  docker network ls
  ```

- **Run a Container in a Specific Network:**

  ```
  docker run --name container1 --network my_bridge_network -d nginx
  ```

- **Connect a Running Container to a Network:**

  ```
  docker network connect my_bridge_network container2
  ```

2. Docker Host Network:

- **Run a Container with Host Network:**

  ```
  docker run --name container_host_net --network host -d nginx
  ```

3. Docker Overlay Network:

Overlay networks allow communication between containers across multiple Docker hosts.

- **Create an Overlay Network:**

  ```
  docker network create --driver overlay my_overlay_network
  ```

- **List Overlay Networks:**

  ```
  docker network ls
  ```

- **Run a Service in the Overlay Network:**

  ```
  docker service create --name service1 --network my_overlay_network nginx
  ```

4. Docker Macvlan Network:

Macvlan networks allow containers to have their own MAC address on the network.

- **Create a Macvlan Network:**

  ```
  docker network create -d macvlan --subnet=192.168.1.0/24 --
  gateway=192.168.1.1 -o parent=eth0 my_macvlan_network
  ```

- **Run a Container in Macvlan Network:**

  ```
  docker run --name container_macvlan --network my_macvlan_network -d nginx
  ```

5. Docker Inspect:

Inspecting a network provides detailed information about the network configuration.

- **Inspect a Network:**

  ```
  docker network inspect my_bridge_network
  ```

6. Docker Port Mapping:

Mapping ports allows accessing services running inside containers.

- **Run Container with Port Mapping:**

  ```
  docker run -p 8080:80 --name web_server -d nginx
  ```

7. Docker DNS Resolution:

Docker provides DNS resolution for container names.

- **Run Containers with DNS Resolution:**

- ```
  docker run --name container_dns1 --network my_bridge_network -d nginx
  docker run --name container_dns2 --network my_bridge_network -d nginx
  ```

  Containers can communicate with each other using container names as hostnames.

These are some fundamental Docker networking commands and concepts. Depending on your use case, you might choose different network types or combinations to suit your application architecture and deployment requirements.
  ```

Docker-Compose

Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to define your application's services, networks, and volumes in a YAML file, and then you can use `docker-compose` commands to start and manage your entire application stack. Here are some Docker Compose concepts, examples, and commands:

1. Docker Compose YAML File:

Create a file named `docker-compose.yml` to define your services, networks, and volumes. Below is a simple example:

```
version: '3'

services:
  web:
    image: nginx
    ports:
      - "8080:80"

  db:
    image: postgres
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword
```

2. Docker Compose Commands:

- **Build and Start Services:**

  ```
  docker-compose up -d
  ```

  This command reads the `docker-compose.yml` file, creates the defined services, and starts the containers in the background (`-d` flag).

- **Stop Services:**

  ```
  docker-compose down
  ```

  This command stops and removes the containers defined in the `docker-compose.yml` file.

- **View Logs:**

  ```
  docker-compose logs
  ```

  View the logs of all services.

- **Scale Services:**

  ```
  docker-compose up -d --scale web=3
  ```

  This scales the number of instances of the `web` service to 3.

- **List Services:**

  ```
  docker-compose ps
  ```

  List the running services and their status.

3. Environment Variables:

You can set environment variables for your services in the `docker-compose.yml` file:

```
services:
  web:
    image: nginx
    environment:
      - MY_VARIABLE=value
```

4. Volumes:

Define volumes for persistent storage:

```
services:
  web:
    image: nginx
    volumes:
      - ./html:/usr/share/nginx/html
```

This maps the local `./html` directory to the `/usr/share/nginx/html` directory in the container.

5. Networks:

Specify custom networks:

```
networks:
  my_network:
    driver: bridge

services:
  web:
    image: nginx
    networks:
      - my_network
```

6. Dependencies and Restart Policies:

```
services:
  web:
    image: nginx
    depends_on:
      - db
    restart: always

  db:
    image: postgres
```

The `depends_on` key ensures that the `web` service starts only after the `db` service is up. The `restart: always` policy ensures that the container restarts automatically.

7. Override Compose Files:

You can use multiple compose files for different environments:

```
docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d
```

8. Docker Compose Build:

You can also build images using `docker-compose`:

```
docker-compose build
```

This command builds the images defined in the `docker-compose.yml` file.
These are just a few examples of what you can do with Docker Compose. The tool is flexible and can be tailored to fit various application architectures and deployment scenarios.

Docker Volumes

Docker volumes are a way to persist and share data between Docker containers. They provide a mechanism for managing data that needs to live beyond the lifetime of a container. Here's an overview of Docker volumes along with examples and commands:

1. Creating Volumes:

- **Creating a Named Volume:**

  ```
  docker volume create my_volume
  ```

- **Creating a Volume During Container Launch:**

  ```
  docker run -v my_volume:/path/in/container -d nginx
  ```

2. Listing Volumes:

```
docker volume ls
```

3. Inspecting a Volume:

```
docker volume inspect my_volume
```

4. Mounting Volumes in Containers:

- **Mounting a Volume to a Container:**

  ```
  docker run -v my_volume:/path/in/container -d nginx
  ```

- **Mounting a Host Directory to a Container:**

  ```
  docker run -v /host/path:/path/in/container -d nginx
  ```

5. Bind Mounts:

Bind mounts allow you to mount a directory from the host into a container.

- **Using Bind Mounts:**

  ```
  docker run -v /host/data:/path/in/container -d nginx
  ```

6. Using Volumes with Docker Compose:

Docker Compose simplifies the definition and management of multi-container applications.

- **Docker Compose Example:**

```
version: '3'
services:
  web:
    image: nginx
    volumes:
      - my_volume:/path/in/container
volumes:
  my_volume:
```

Run with `docker-compose up`.

7. Removing Volumes:

- **Removing a Volume:**

  ```
  docker volume rm my_volume
  ```

8. Data-Only Containers (Legacy):

In older Docker versions, data-only containers were used to create shared volumes.

- **Creating a Data-Only Container:**

  ```
  docker create -v /path/in/container --name data_container busybox
  ```

  Other containers can then use `--volumes-from` to access this data.

9. Named Volumes with Docker Compose:

```
version: '3'
services:
  web:
    image: nginx
    volumes:
      - my_volume:/path/in/container
volumes:
  my_volume:
```

10. Using Volumes for Database Containers:

```
docker run -v /path/to/db/data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw
-d mysql:latest
```

These commands and examples cover the basic usage of Docker volumes. Volumes play a crucial role in Docker for persistent data storage and sharing between containers, making it easier to manage data in containerized applications.

50 Docker Errors & Solutions

Docker errors can arise from various issues, and troubleshooting them can sometimes be challenging. Here are 50 common Docker errors along with potential solutions:

1. **Error: Cannot connect to the Docker daemon**
    - Solution: Ensure Docker is running (`docker info`).
2. **Error: Got permission denied while trying to connect to the Docker daemon socket**
    - Solution: Run Docker commands with `sudo` or add your user to the `docker` group.
3. **Error: Unable to locate package docker-ce**
    - Solution: Follow the correct installation instructions for your Linux distribution.
4. **Error: Image is not on the Docker host**
    - Solution: Pull the image with `docker pull`.
5. **Error: No space left on device**
    - Solution: Clean up unused images/containers or resize the Docker disk space.
6. **Error: Container command 'XXX' not found**
    - Solution: Ensure the command exists in the container or check the container image.
7. **Error: Cannot start container: port has already been allocated**
    - Solution: Choose a different host port or stop the conflicting container.
8. **Error: Connection refused**
    - Solution: Ensure the service inside the container is running or check firewall settings.
9. **Error: Unable to access jarfile /path/to/file.jar**
    - Solution: Check the file path or container's working directory.
10. **Error: Exited with code 137**
    - Solution: Insufficient resources (increase memory or CPU).

11. **Error: No such file or directory**
    - Solution: Check file paths and existence inside the container.
12. **Error: port is already allocated**
    - Solution: Choose a different port or stop the conflicting service.
13. **Error: OCI runtime create failed: container_linux.go:348: starting container process caused "exec: "XXX": executable file not found in $PATH"**
    - Solution: Ensure the command exists or check the path within the container.
14. **Error: Cannot assign requested address**
    - Solution: Check network configurations and available IP addresses.
15. **Error: Cannot create container for service XXX: network XXX not found**
    - Solution: Ensure the network specified in the docker-compose.yml exists.
16. **Error: COPY failed: no such file or directory**
    - Solution: Verify file paths and existence in the Dockerfile.
17. **Error: unauthorized: authentication required**
    - Solution: Log in to the Docker registry using `docker login`.
18. **Error: Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock**
    - Solution: Add your user to the `docker` group or use `sudo`.
19. **Error: image operating system "XXX" cannot be used on this platform**
    - Solution: Use images compatible with your platform (e.g., multi-platform images).
20. **Error: Container command 'XXX' not found or does not exist**
    - Solution: Check the command spelling or verify the image contents.
21. **Error: Network unreachable**
    - Solution: Check network configurations or restart Docker.
22. **Error: No module named 'requests'**

- Solution: Install required Python packages in your Docker image.

23. **Error: Bind for 0.0.0.0:XXX failed: port is already allocated**
    - Solution: Choose a different port or stop the service using that port.
24. **Error: Cannot connect to the Docker daemon at tcp://XXX:2376. Is the docker daemon running?**
    - Solution: Ensure Docker is running or check the Docker daemon configuration.
25. **Error: Cannot connect to the Docker daemon. Is 'docker -d' running on this host?**
    - Solution: Start the Docker daemon with `dockerd`.
26. **Error: The requested image's platform (linux/amd64) does not match the detected host platform (linux/arm64)**
    - Solution: Use images compatible with your host platform.
27. **Error: no matching manifest for XXX in the manifest list entries**
    - Solution: Pull an image compatible with your architecture.
28. **Error: The container name "/XXX" is already in use**
    - Solution: Choose a different container name or remove the existing container.
29. **Error: ERROR: for service Cannot start service XXX: driver failed programming external connectivity**
    - Solution: Release the port from another service or choose a different one.
30. **Error: error while creating mount source path '/host/path': mkdir /host/path: file exists**
    - Solution: The specified path should not be an existing directory.
31. **Error: Unable to find image 'XXX' locally**
    - Solution: Pull the image with `docker pull` before running the container.
32. **Error: rpc error: code = 2 desc = "oci runtime error: exec format error"**
    - Solution: Use images compatible with your platform.
33. **Error: No such container: XXX**
    - Solution: Ensure the container exists or check the container name.

34. **Error: invalid argument: unknown flag: --network**
    - Solution: Update Docker to a version that supports the specified flag.
35. **Error: Dockerfile parse error line XXX: unknown instruction: YYY**
    - Solution: Check Dockerfile syntax and instruction spelling.
36. **Error: oci runtime error: container_linux.go:247: starting container process caused "process_linux.go:359: container init caused "rootfs_linux.go:54: mounting \"XXX\" to rootfs \"/var/lib/docker/overlay2/XXX/merged\" at \"/path/in/container\" caused \"not a directory\""**
    - Solution: The source path in a volume mount should be a directory.
37. **Error: network XXX not found**
    - Solution: The specified network should exist or create it using `docker network create`.
38. **Error: Unable to load the image. The file is too big.**
    - Solution: Optimize the image size or use a more lightweight base image.
39. **Error: Unable to find a node that satisfies the following conditions**
    - Solution: Ensure your Docker Swarm has available nodes.
40. **Error: Error response from daemon: invalid mode: /path/in/container**
    - Solution: Check the syntax of the volume mount mode.
41. **Error: failed to build: COPY failed: no such file or directory**
    - Solution: Ensure the file or directory exists in the build context.
42. **Error: failed to register layer: Error processing tar file(exit status 1): write /path/in/container: no space left on device**
    - Solution: Free up disk space or resize the Docker disk.
43. **Error: The server requested authentication method unknown to the client [tls]**
    - Solution: Upgrade Docker to a version that supports the TLS authentication method.
44. **Error: Another program is using port XXX**

- Solution: Stop the program using the conflicting port.

45. **Error: /usr/bin/docker: Error response from daemon: Conflict. The container name "/XXX" is already in use by container "YYY".**
    - Solution: Remove or rename the existing container with the conflicting name.
46. **Error: open /var/run/docker.sock: no such file or directory**
    - Solution:

Ensure Docker is running or check the Docker daemon socket.

47. **Error: Couldn't find an alternative telinit implementation to spawn**
    - Solution: Use `init` as the Docker entry point in your Dockerfile.
48. **Error: error parsing HTTP 403 response body: invalid character '<' looking for beginning of value: "...**
    - Solution: Check Docker Hub credentials or registry authentication.
49. **Error: unknown: Authentication is required**
    - Solution: Log in to the registry with `docker login`.
50. **Error: invalid reference format: repository name must be lowercase**
    - Solution: Ensure repository names are lowercase.