DOCKER

Docker Overview:

Docker is a platform for developing, shipping, and running applications in containers. Containers provide a lightweight and consistent environment, ensuring that an application and its dependencies run consistently across different environments. Docker facilitates the packaging of an application and its dependencies into a single container that can run on any Docker-enabled system.

Key Components of Docker:

1. **Docker Engine:**
    - The core technology that enables building, shipping, and running containers.
    - Comprises a daemon process (`dockerd`), a REST API for interacting with the daemon, and a command-line interface (`docker`).
2. **Docker Image:**
    - A lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, and system tools.
3. **Container:**
    - An instance of a Docker image that can be executed as a process.
    - Containers run in isolated user spaces on a host operating system.
4. **Docker Hub:**
    - A cloud-based registry service for sharing Docker images.
    - Developers can pull and push Docker images to and from Docker Hub.

Problems Docker Solves:

1. **Consistent Environment:**
    - Ensures that an application runs the same way across development, testing, and production environments.
2. **Dependency Management:**

   - Eliminates "it works on my machine" issues by encapsulating an application and its dependencies in a container.

3. **Isolation:**
   - Containers provide process and file system isolation, allowing multiple applications to run on the same host without interfering with each other.
4. **Resource Efficiency:**
   - Containers share the host OS kernel, making them more lightweight and efficient compared to virtual machines.
5. **Portability:**
   - Containers can run on any system with Docker installed, irrespective of the underlying infrastructure or cloud provider.
6. **Scalability:**
   - Enables easy scaling by replicating containers horizontally across multiple hosts.

Examples of Docker Usage:

1. **Web Applications:**
   - Docker is commonly used to package web applications and their dependencies, making it easy to deploy and scale across different environments.
2. **Microservices Architecture:**
   - Docker is well-suited for deploying and managing microservices, where different components of an application run in separate containers.
3. **Continuous Integration/Continuous Deployment (CI/CD):**
   - Docker is often integrated into CI/CD pipelines to create a consistent environment for testing and deploying applications.
4. **Database Containers:**

    - Docker allows packaging and running databases in containers, making it easy to manage different database instances without conflicts.

5. **Development Environments:**
    - Developers can use Docker to create development environments that mirror production, reducing the "it works on my machine" problem.
6. **Legacy Application Migration:**
    - Legacy applications can be containerized, allowing them to run in modern environments without modification.
7. **Serverless Architectures:**
    - Docker containers can be used as the underlying technology for serverless platforms, providing a consistent and portable runtime for serverless functions.

Dockerfile

A Dockerfile is a script that contains instructions for building a Docker image. It specifies the base image, sets up the environment, installs dependencies, and defines how the application should run. Let's go through the key components of a Dockerfile using a simple example.

Example Dockerfile:

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Explanation:

1. **FROM:**
    - Specifies the base image for the Docker image. In this case, it uses the official Python 3.8 image as a starting point.
2. **WORKDIR:**
    - Sets the working directory inside the container to `/app`. Subsequent instructions will be executed in this directory.
3. **COPY:**
    - Copies the current directory (the directory where the Dockerfile is located) into the container at `/app`. This includes your application code.

4. **RUN:**
    - Executes commands in the container during the build process. In this example, it installs the Python dependencies specified in `requirements.txt`.
5. **EXPOSE:**
    - Informs Docker that the container will listen on the specified network ports at runtime. It does not actually publish the ports; it's more of a documentation feature.
6. **ENV:**
    - Sets an environment variable, in this case, `NAME` with the value "World". This variable can be accessed by the application inside the container.
7. **CMD:**
    - Provides the default command to run when the container starts. In this example, it runs `python app.py`. You can override this command when running the container.

Usage:

1. Save the above Dockerfile in a file named `Dockerfile` in your project directory.
2. Create a file named `requirements.txt` if your application has Python dependencies. Add the necessary dependencies.
3. Create your Python application file (e.g., `app.py`) in the same directory.

4. Build the Docker image:

    ```
    docker build -t my-python-app .
    ```

5. Run the Docker container:

    ```
    docker run -p 4000:80 my-python-app
    ```

This example assumes a simple Python web application. Adjust the Dockerfile based on your specific application, its dependencies, and how it needs to be executed within the container.

Docker Image

A Docker image is a lightweight, standalone, and executable software package that includes everything needed to run a piece of software, including the code, runtime, libraries, dependencies, and system tools. Docker images are built from a set of instructions called a Dockerfile, which specifies the steps needed to create the image.

Here are some key concepts related to Docker images:

1. **Containerization:** Docker uses containerization technology to package applications and their dependencies into a container. A container is a runnable instance of a Docker image.
2. **Dockerfile:** A Dockerfile is a text document that contains instructions for building a Docker image. It specifies the base image, sets up the environment, installs dependencies, and defines how the application should run.
3. **Layers:** Docker images are built in layers, and each instruction in the Dockerfile adds a new layer to the image. Layers are cached, and if a Dockerfile hasn't changed, Docker can reuse existing layers, making the build process more efficient.
4. **Registry:** Docker images can be stored and shared through a registry. Docker Hub is a popular public registry, but you can also set up private registries for more control over image distribution.
5. **Base Image:** The starting point for a Docker image is often a base image that provides a minimal operating system environment. Common base images include Alpine Linux, Ubuntu, and others.
6. **Tag:** Docker images can have tags to differentiate between different versions or configurations of the same application. For example, an image might have tags like "latest," "v1.0," or "development."

1. Pulling Docker Images:

a. Pulling an Image from Docker Hub:

```
docker pull imageName:tag
# Example: docker pull ubuntu:latest
```

2. Listing Docker Images:

a. Listing Downloaded Images:

```
docker images
```

3. Removing Docker Images:

a. Removing an Image:

```
docker rmi imageName:tag
# Example: docker rmi ubuntu:latest
```

b. Removing All Unused Images:

```
docker image prune
```

4. Tagging Docker Images:

a. Tagging an Image:

```
docker tag sourceImage:tag targetImage:tag
# Example: docker tag ubuntu:latest my-ubuntu:latest
```

5. Building Docker Images:

a. Building an Image from a Dockerfile:

```
docker build -t imageName:tag path/to/Dockerfile
# Example: docker build -t my-node-app:latest .
```

6. Inspecting Docker Images:

a. Inspecting Image Metadata:

```
docker image inspect imageName:tag
# Example: docker image inspect ubuntu:latest
```

b. Displaying Image Layers:

```
docker image history imageName:tag
# Example: docker image history ubuntu:latest
```

7. Saving and Loading Docker Images:

a. Saving an Image to a TAR file:

```
docker save -o output.tar imageName:tag
# Example: docker save -o ubuntu_latest.tar ubuntu:latest
```

b. Loading an Image from a TAR file:

```
docker load -i input.tar
# Example: docker load -i ubuntu_latest.tar
```

8. Exporting and Importing Docker Images:

a. Exporting an Image to a TAR file:

```
docker export -o output.tar containerName
# Example: docker export -o ubuntu_container.tar ubuntu_container
```

b. Importing an Image from a TAR file:

```
docker import input.tar imageName:tag
# Example: docker import ubuntu_container.tar my-ubuntu-container:latest
```

These commands cover various aspects of working with Docker images, including pulling, listing, removing, tagging, building, inspecting, and managing images. Adjust these examples based on your specific use cases and requirements.

Docker Container

Docker containers are instances of Docker images, and they encapsulate applications and their dependencies, allowing them to run consistently across different environments. Here are some basic Docker container commands with examples:

1. **Pulling an Image:** To download a Docker image from a registry (e.g., Docker Hub), you use the `docker pull` command. For example, to pull the official Nginx image:
   ```
   docker pull nginx
   ```
2. **Running a Container:** To run a container from an image, you use the `docker run` command. For example, to start a basic Nginx web server:
   ```
   docker run -d -p 8080:80 nginx
   ```
   - `-d`: Run the container in the background (detached mode).
   - `-p 8080:80`: Map port 8080 on the host to port 80 on the container.
3. **Listing Running Containers:** To see the list of running containers, you use the `docker ps` command:
   ```
   docker ps
   ```
4. **Stopping a Container:** To stop a running container, you use the `docker stop` command and specify the container ID or name:
   ```
   docker stop <container_id or container_name>
   ```
5. **Starting a Stopped Container:** To start a stopped container:
   ```
   docker start <container_id or container_name>
   ```
6. **Removing a Container:** To remove a container, you use the `docker rm` command and specify the container ID or name:
   ```
   docker rm <container_id or container_name>
   ```
7. **Viewing Container Logs:** To view the logs of a running container:
   ```
   docker logs <container_id or container_name>
   ```
8. **Executing Commands in a Running Container:** To execute a command inside a running container:
   ```
   docker exec -it <container_id or container_name> /bin/bash
   ```
   This opens an interactive shell (`bash`) inside the container.

These are basic commands, and Docker provides many more options and features for managing containers.

Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. It allows you to define a multi-container application in a single file, then spin up the entire application stack with a single command. Here's an overview of Docker Compose with examples:

1. Compose File:

Docker Compose configurations are written in YAML format and typically named `docker-compose.yml`. This file defines services, networks, and volumes for your application.
Example `docker-compose.yml` for a simple web application with a web server and a database:

```yaml
version: '3'
services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
  db:
    image: postgres:latest
    environment:
      POSTGRES_DB: mydatabase
      POSTGRES_USER: myuser
      POSTGRES_PASSWORD: mypassword
```

2. Services:

Each service in the `docker-compose.yml` file corresponds to a container. In the example above, there are two services: `web` and `db`.

3. Building and Running Containers:

To build and run the containers defined in the `docker-compose.yml` file:

```bash
docker-compose up
```

- The `-d` flag can be used to run the containers in the background.

4. Stopping Containers:

To stop the running containers:

```bash
docker-compose down
```

- The `down` command stops and removes containers, networks, and volumes defined in the `docker-compose.yml` file.

5. Scaling Services:

You can scale services easily. For example, to run three instances of the `web` service:
```bash
docker-compose up --scale web=3 -d
```

6. Environment Variables:

You can define environment variables for services in the `docker-compose.yml` file. In the example, the `db` service uses environment variables for configuring the PostgreSQL database.

7. Volumes:

Docker Compose allows you to define volumes to persist data between container restarts. For example, adding a volume for the `web` service:
```yaml
services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html
```
In this case, the `./html` directory on the host is mounted to the `/usr/share/nginx/html` directory in the `web` container.

8. Networks:

Docker Compose automatically creates a default network for your services to communicate. You can define custom networks and specify which services should be connected to them.

```
networks:
  custom_network:
services:
  web:
    networks:
      - custom_network
  db:
    networks:
      - custom_network
```

In this example, both the `web` and `db` services are connected to the `custom_network`.

9. Override Compose File:

You can use an override file (e.g., `docker-compose.override.yml`) to add or override settings in the main `docker-compose.yml` file. This is useful for local development or testing.

10. Docker Compose Commands:

- `docker-compose ps`: List containers.
- `docker-compose exec <service_name> <command>`: Execute a command in a running container.
- `docker-compose logs`: View container logs.

Docker Compose simplifies the process of managing multi-container applications, making it easier to define, configure, and run complex application stacks.

Docker Network

Docker provides a networking feature that allows containers to communicate with each other and with the outside world. Docker networking enables seamless connectivity between containers on the same host or across multiple hosts in a cluster. Here's an overview of Docker networking concepts along with examples:

Docker Networking Concepts:

1. **Bridge Network:**
   - The default network type for containers.
   - Containers on the same bridge network can communicate with each other.
   - Each container gets its own IP address within the bridge network.
2. **Host Network:**
   - Containers share the host's network stack.
   - Containers can directly bind to host ports.
3. **Overlay Network:**
   - Used for connecting containers across multiple Docker hosts (in a swarm cluster).
   - Provides network abstraction, allowing containers to communicate as if they are on the same host.
   - Commonly used in Docker Swarm mode.
4. **Macvlan Network:**
   - Allows containers to have their own MAC address, making them appear as physical devices on the network.
   - Useful when containers need to be part of an existing network.
5. **User-Defined Bridge Network:**
   - Custom bridge networks can be created by users.
   - Containers on the same custom bridge network can communicate with each other.

Docker Network Commands:

1. **Create a Custom Bridge Network:**

   ```
   docker network create mynetwork
   ```

2. **Run a Container on a Specific Network:**

   ```
   docker run --network mynetwork --name mycontainer -d nginx
   ```

3. **Inspect Network Details:**

   ```
   docker network inspect mynetwork
   ```

4. **Connect a Container to a Network:**

   ```
   docker network connect mynetwork mycontainer
   ```

5. **Disconnect a Container from a Network:**

   ```
   docker network disconnect mynetwork mycontainer
   ```

6. **List Networks:**

   ```
   docker network ls
   ```

Example: Custom Bridge Network

1. **Create a Custom Bridge Network:**

   ```
   docker network create mynetwork
   ```

2. **Run Containers on the Custom Network:**

3.
   ```
   docker run --network mynetwork --name container1 -d nginx
   docker run --network mynetwork --name container2 -d nginx
   ```

4. **Inspect Network Details:**

   ```
   docker network inspect mynetwork
   ```

   Look for the containers under the "Containers" section.

5. **Communication Between Containers:** Containers `container1` and `container2` can communicate with each other using their container names as hostnames.
6.

Docker networking is a powerful feature that facilitates communication between containers, and it becomes especially crucial in scenarios where you have multiple containers or services working together.

Docker Volumes

Docker volumes are a way to persistently store and manage data used by Docker containers. Volumes provide a mechanism for sharing data between containers, as well as persisting data beyond the lifecycle of a single container. Here's an overview of Docker volumes along with examples:

Docker Volume Concepts:

1. **Named Volumes:**
   - Named volumes are a way to assign a specific name to a volume.
   - They are easily referenced by name when attaching them to containers.
   - Docker manages the volume's location on the host.
2. **Bind Mounts:**
   - Bind mounts allow you to directly mount a directory from the host into a container.
   - Data is shared between the host and the container in real-time.
   - Docker does not manage the content of bind mounts.

Docker Volume Commands:

1. **Create a Named Volume:**

   ```
   docker volume create mydata
   ```

2. **List Volumes:**

   ```
   docker volume ls
   ```

3. **Inspect Volume Details:**

   ```
   docker volume inspect mydata
   ```

4. **Run a Container with a Named Volume:**

   ```
   docker run -d --name mycontainer -v mydata:/path/in/container nginx
   ```

   This command runs an Nginx container and creates/uses the named volume "mydata."

5. **Run a Container with a Bind Mount:**

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

   This command runs an Nginx container and mounts the host directory "/host/path" into the container.

6. **Inspect Container Mounts:**

   ```
   docker inspect mycontainer
   ```

   Look for the "Mounts" section to see the volume or bind mount details.

Example: Named Volume

1. **Create a Named Volume:**

   ```
   docker volume create mydata
   ```

2. **Run a Container with the Named Volume:**

   ```
   docker run -d --name mycontainer -v mydata:/usr/share/nginx/html nginx
   ```

   This command runs an Nginx container with the named volume "mydata" mounted to the container's "/usr/share/nginx/html" directory.

3. **Write Data to the Volume:**

```
4. docker exec -it mycontainer sh
5. echo "Hello, Docker Volumes!" > /usr/share/nginx/html/index.html
   exit
```

6. **Verify Data Persistence:** Stop and remove the container, then run a new container using the same named volume:

```
7. docker rm -f mycontainer
   docker run -d --name newcontainer -v mydata:/usr/share/nginx/html nginx
```

   The new container should have the same data written to the volume.

Example: Bind Mount

1. **Run a Container with a Bind Mount:**

   ```
   docker run -d --name mycontainer -v /host/path:/usr/share/nginx/html nginx
   ```

   This command runs an Nginx container with a bind mount, linking the host directory "/host/path" to the container's "/usr/share/nginx/html" directory.

2. **Write Data to the Bind Mount:** You can add or modify files directly on the host in the "/host/path" directory.

3. **Verify Data Sharing:** Changes made on the host in "/host/path" are reflected inside the container at "/usr/share/nginx/html."

Docker volumes are essential for managing data in containerized applications, allowing for data persistence, sharing, and easy management across container instances.