Docker Volume

Docker volumes are a way to persist and manage data across container lifecycles. Unlike bind mounts, which map a host directory to a container, Docker volumes are fully managed by Docker, making them a preferred option for storing persistent data. A **Docker volume** is a special directory that Docker manages. It also ensure that data remains accessible even if containers are deleted.

Overview of Docker volume features

Data sharing across containers:

Docker volumes make it possible for different containers to exchange and use the same data. This feature is quite helpful when microservices or multiple containers need to connect and work together by sharing common data.

Volume Snapshotting:

Volume snapshotting is supported by a few storage drivers and plugins for Docker volumes. You can create a snapshot at a specific point in time that can be later used for backup, cloning, or versioning purposes.

Mount Options:

Docker volumes provide flexible mount options so that the volume can be mounted inside containers in a way that works for you. Read-only mode, choosing the volume driver, configuring consistency parameters, and more are all included in the mount options.

Volume Security:

Security features for Docker volumes include access control and permission management. You can specify who is allowed to read, write, or execute on a volume, ensuring that sensitive information is kept secure and that only authorized containers or users can access it.

Why Use Docker Volumes?

- **Persistent Data Storage**: Ensures data is not lost when a container stops.
- **Container Independence**: Containers can be deleted without affecting the volume.
- **Performance Optimization**: Volumes are managed by Docker, optimized for containerized environments.
- **Data Sharing**: Multiple containers can share the same volume.

Creating a Volume

To create a named volume, use the following command:

```
docker volume create my_volume
```

To list all volumes:

```
docker volume ls
```

To inspect a volume:

```
docker volume inspect my_volume
```

Using a Volume in a Container

To mount a volume inside a container, use the -v flag:

docker run -d --name my_container -v my_volume:/data busybox

- This mounts my_volume inside the container at /data.
- Any data written to /data inside the container persists even after the container is removed.

To verify that data persists, you can start another container with the same volume:

docker run --rm -v my_volume:/data busybox ls /data

Removing a Volume

To remove a volume, first ensure it is not being used by any container:

docker volume rm my_volume

To remove all unused volumes:

docker volume prune

Anonymous Volumes

Docker can create unnamed (anonymous) volumes when running a container:

docker run -d --name my_container -v /data busybox

- Since no volume name is specified, Docker creates an anonymous volume.
- To find it, run docker volume ls and check docker inspect my_container.

Anonymous volumes are removed automatically when the container is deleted, unless manually specified otherwise.

Bind Mounts vs. Volumes

Feature	**Bind Mounts**	**Docker Volumes**
Storage Location	Any directory on host machine	Managed by Docker in /var/lib/docker/volumes/
Performance	Slightly slower (depends on filesystem)	Optimized for Docker
Flexibility	Can access any host file	Only accessible through Docker
Security	Less secure	More secure

Use **bind mounts** if you need direct access to host files and **volumes** for persistent Docker-managed storage.

Sharing Volumes between Containers

Multiple containers can share a volume:

docker run -d --name container1 -v shared_volume:/data busybox
docker run -d --name container2 -v shared_volume:/data busybox

- Both containers can read/write to /data.
- Useful for microservices that need shared data.

Backing Up & Restoring Volumes

To back up a volume:

docker run --rm -v my_volume:/data -v $(pwd):/backup busybox tar cvf /backup/backup.tar /data

To restore from a backup:
docker run --rm -v my_volume:/data -v $(pwd):/backup busybox tar xvf /backup/backup.tar -C /

Adding Data via a Running Container

You can write files into a mounted volume from inside a running container.
Steps:

1. **Create a volume** (if not already created):
   ```
   docker volume create my_volume
   ```

2. **Run a container with the volume attached**:
   ```
   docker run -it --rm -v my_volume:/data busybox sh
   ```

3. **Inside the container, create files in the volume**:
   ```
   echo "Hello, Docker!" > /data/hello.txt
   ```

4. **Exit the container**:
   ```
   Exit
   ```

5. **Verify the data by launching another container**:
   ```
   docker run --rm -v my_volume:/data busybox cat /data/hello.txt
   ```
   Expected output:
   ```
   Hello, Docker!
   ```

Containerization

A Dockerfile is a text-based document that's used to create a container image. It provides instructions to the image builder on the commands to run, files to copy, startup command, and more.

Common instructions

Some of the most common instructions in a Dockerfile include:

- FROM <image> - this specifies the base image that the build will extend.
- WORKDIR <path> - this instruction specifies the "working directory" or the path in the image where files will be copied and commands will be executed.
- COPY <host-path> <image-path> - this instruction tells the builder to copy files from the host and put them into the container image.
- RUN <command> - this instruction tells the builder to run the specified command.
- ENV <name> <value> - this instruction sets an environment variable that a running container will use.
- EXPOSE <port-number> - this instruction sets configuration on the image that indicates a port the image would like to expose.
- USER <user-or-uid> - this instruction sets the default user for all subsequent instructions.
- CMD ["<command>", "<arg1>"] - this instruction sets the default command a container using this image will run.

Creating the Dockerfile

Now that you have the project, you’re ready to create the Dockerfile.

1. Download and install Docker Desktop.

2. Create a file named Dockerfile in the same folder as the file package.json.

3. In the Dockerfile, define your base image by adding the following line:
   ```
   FROM node:20-alpine
   ```

4. Now, define the working directory by using the WORKDIR instruction. This will specify where future commands will run and the directory files will be copied inside the container image.
   ```
   WORKDIR /app
   ```

5. Copy all of the files from your project on your machine into the container image by using the COPY instruction:
   ```
   COPY . .
   ```

6. Install the app's dependencies by using the yarn CLI and package manager. To do so, run a command using the RUN instruction:
   ```
   RUN yarn install –production
   ```

7. Finally, specify the default command to run by using the CMD instruction:
   ```
   CMD ["node", "./src/index.js"]
   ```

8. And with that, you should have the following Dockerfile:
   ```
   FROM node:20-alpine
   WORKDIR /app
   COPY . .
   RUN yarn install --production
   CMD ["node", "./src/index.js"]
   ```

Building images

Most often, images are built using a Dockerfile. The most basic docker build command might look like the following:
```
docker build .
```

The final . in the command provides the path or URL to the build context. At this location, the builder will find the Dockerfile and other referenced files. When you run a build, the builder pulls the base image, if needed, and then runs the instructions specified in the Dockerfile.

Publishing images

Once you have an image built and tagged, you're ready to push it to a registry. To do so, use the docker push command:
```
docker push my-username/my-image
```

Container environment variables

Environment variables should not be **baked into the Docker image** (i.e., they should not be defined inside the Dockerfile directly) but should be provided dynamically at **container runtime** instead.

Why shouldn't the .env File Be inside the Image?

1. **Security Risk**: If .env is included in the Docker image, it could expose sensitive data (e.g., database credentials) if the image is shared.
2. **Flexibility**: Keeping environment variables outside the image allows easy configuration changes without rebuilding the image.
3. **Portability**: Different environments (development, staging, production) may require different settings, which are better managed externally.

How to Introduce Container Environment Variables?

- Using docker run --env (Best for Quick Testing)
  docker run -e MONGO_URI=mongodb://mongo:27017/mydatabase -e NODE_ENV=production -p 3000:3000 my-app
  This dynamically injects the environment variables into the running container.

- Using an .env File (Recommended)
  Instead of storing the .env inside the image, you can provide it dynamically.
  1. Create a .env file (DO NOT include it in the image)
     MONGO_URI=mongodb://mongo:27017/mydatabase
     NODE_ENV=production
     PORT=3000
  2. Pass it at runtime using --env-file:
     docker run --env-file .env -p 3000:3000 my-app
     This ensures that the .env file remains outside the image but is still available inside the running container.