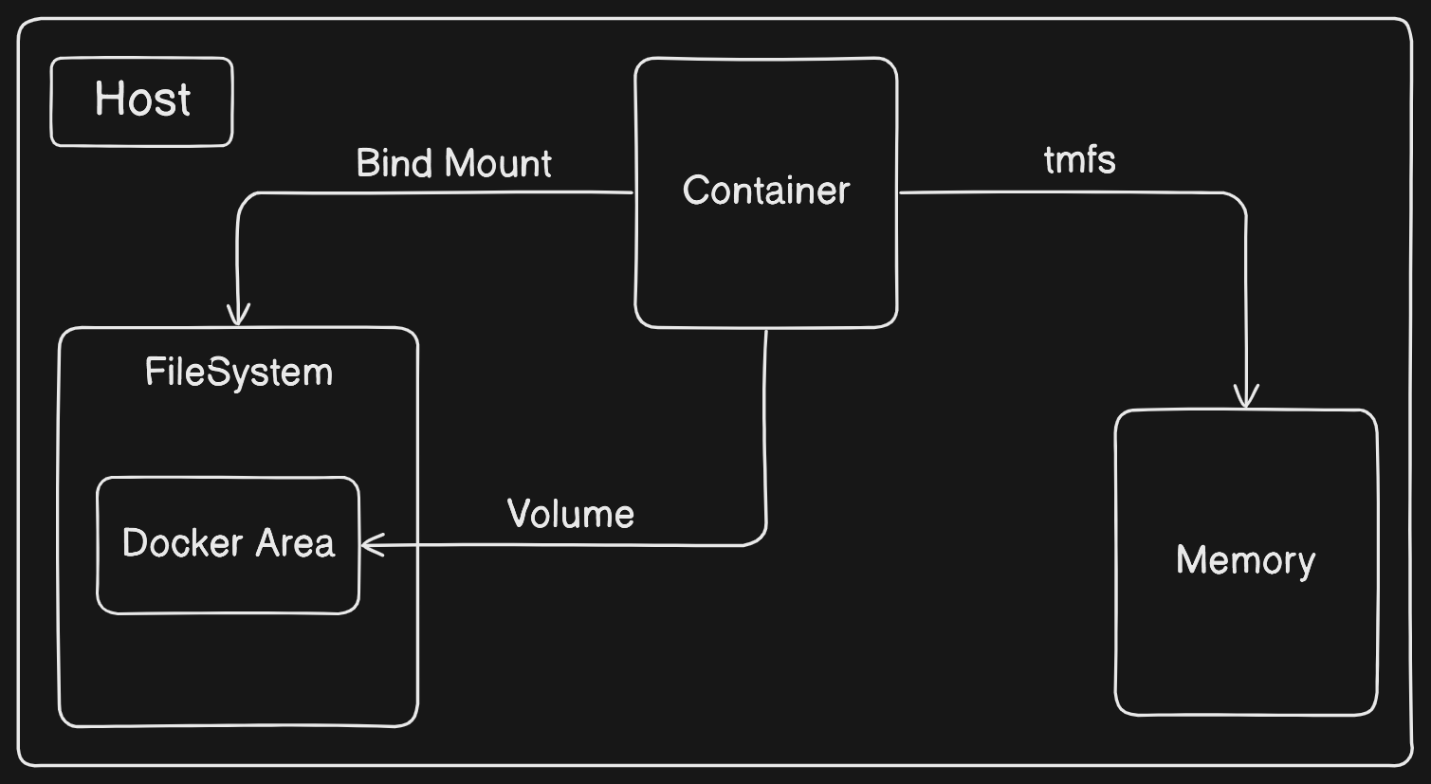
**In-Memory Storage**

Many service software and web applications require handling sensitive configuration files such as private key files, database passwords, and API key files. These files should never be included in the image or written to disk for security reasons. Instead, utilizing in-memory storage is crucial to ensure sensitive data remains secure. This readme provides guidance on implementing in-memory storage using Docker containers with tmpfs mounts.

**Using tmpfs Mounts**

To implement in-memory storage, we can use tmpfs mounts with Docker containers. The tmpfs mount type allow us to create *a memory-based filesystem* within the container's file tree.

**Command Syntax**

**We can use the following command to mount a tmpfs device into a container's file tree**:

docker run --rm \

--mount type=tmpfs,dst=/tmp \

--entrypoint mount \

 alpine:latest -v

**Command Explanation**

**--mount type=tmpfs,dst=/tmp:** This part of the command specifies that a tmpfs device will be mounted at the /tmp directory within the container's file system.

**--entrypoint mount:** This specifies the entry point for the container as the mount command.

**alpine:latest:** This indicates the Docker image to be used for the container.

**Configuration Details**

When the above command is executed, it creates an empty tmpfs device and attaches it to the container's file tree at /tmp. Files created under this file tree will be stored in memory instead of on disk. The mount point is configured with sensible defaults for generic workloads.

**Mount-point Configuration**

**Upon execution, the command displays a list of all mount points for the container. Here's a breakdown of the configuration provided:**

tmpfs on /tmp type tmpfs (rw,nosuid,nodev,noexec,relatime): This line describes the mount-point configuration.

**tmpfs on /tmp:** Indicates that a tmpfs device is mounted to the tree at /tmp.

**type tmpfs:** Specifies that the device has a tmpfs filesystem.

**rw:** Indicates that the tree is read/write capable.

**nosuid:** Specifies that suid bits will be ignored on all files in this tree.

**nodev:** Indicates that no files in this tree will be interpreted as special devices.

**noexec:** Specifies that no files in this tree will be executable.

**relatime:** Indicates that file access times will be updated if they are older than the current modify or change time.

**Additional Options**

**We can further customize the tmpfs mount by adding the following options:**

* **tmpfs-size:** Specifies the size limit of the tmpfs device.
* **tmpfs-mode:** Specifies the file mode for the tmpfs device.

**Example Command with Additional Options**

docker run --rm \

--mount type=tmpfs,dst=/app/tmp,tmpfs-size=16k,tmpfs-mode=1770 \

--entrypoint mount \

alpine:latest -v

This command limits the tmpfs device mounted at /tmp to **16 KB** and configures it to be *not readable* by other in-container users.

**Example Scenario**

Consider a scenario where you're developing a microservice-based application that requires handling *sensitive* configuration files and *temporary data* processing. Instead of storing these files on disk, you opt for in-memory storage to enhance security and performance.

**Implementation:**

**Step 01. Docker Configuration:**

Create a Dockerfile for the microservice.

Configure the Dockerfile to use a tmpfs mount for storing sensitive data in memory.

**[Dockerfile: docker file ase folder a]**

**Step 02. Sensitive Data Processing:**

Now task is to develop a microservice to generate and process sensitive data within the memory-based filesystem.

We have to ensure that any temporary files or sensitive configuration files are created and accessed within the mounted tmpfs directory.

**Create a simple micorservice named example\_microservice.py**

**[example\_microservice.py: file ta ase folder a]**

**In this completion:**

We have created a simple microservice named example\_microservice.py that processes sensitive data within the mounted tmpfs directory.

The microservice ensures the temporary directory exists and writes sensitive data to a file within the mounted tmpfs directory.

The microservice then reads the sensitive data from the file and performs some processing on it.

The processed data is then printed to the console.

The microservice uses the os module to interact with the file system and ensure that all file operations are performed within the mounted tmpfs directory.

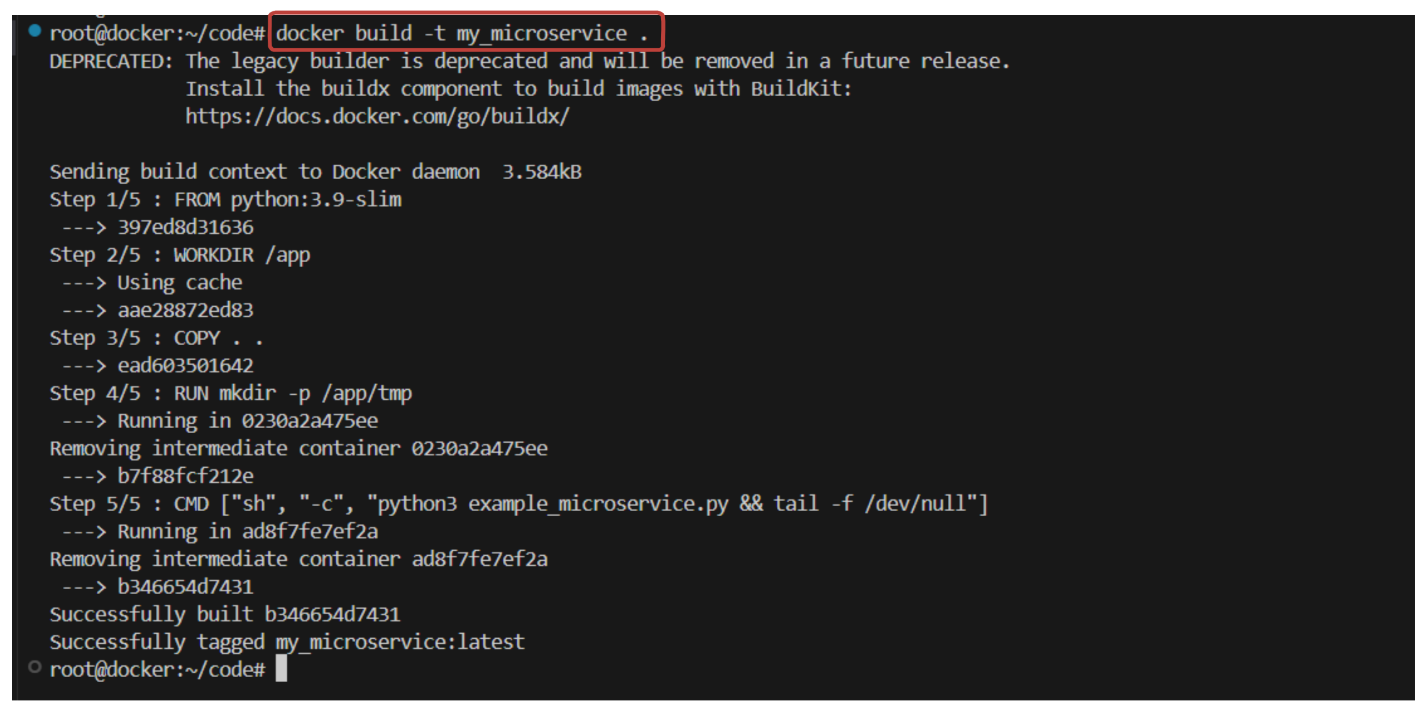
**Step 03: Build and Run Docker Container:**

**Build the Docker image for the microservice.**

**Run the Docker container, ensuring that the tmpfs mount is properly configured.**

**# Build Docker image**

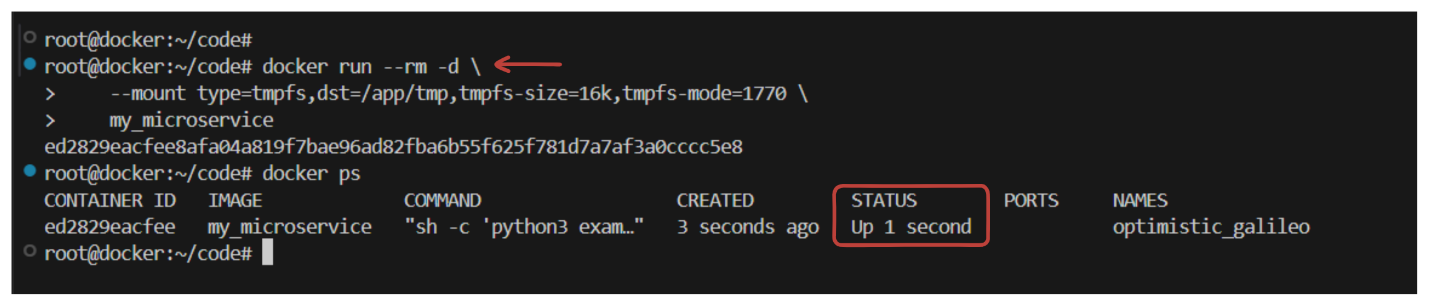
**docker build -t my\_microservice .**

****

docker run --rm -d \

--mount type=tmpfs,dst=/app/tmp,tmpfs-size=16k,tmpfs-mode=1770 \

my\_microservice



**Step 04: Ensure that the tmpfs mount is properly configured**

To ensure that the tmpfs mount is properly configured when running the Docker container involves verifying that the container has access to the tmpfs-mounted directory and that sensitive data is being stored within this directory. Here's how we can ensure this:

**Verify Mount Configuration**: When running the Docker container, ensure that the tmpfs mount is configured correctly. we can do this by inspecting the container's mounts:

docker inspect <container\_id\_or\_name>



This command will display information about the container, including its mounts. Ensure that the tmpfs mount is listed and mounted at the expected directory (e.g., /app/tmp).

**Check File Storage Location**: Confirm that sensitive data is being stored within the tmpfs-mounted directory. We can do this by examining the files created or accessed by the container.

docker exec -it <container\_id\_or\_name> /bin/bash

cd tmp && ls

cat sensitive\_data.txt

