Container Network - In-Memory Storage with Docker

In-memory storage

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
```
alt text

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 Scenerio
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.

```
# Use an official Python runtime as a base image
FROM python:3.9-slim

# Set the working directory in the container
WORKDIR /app

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

# Ensure the directory exists
RUN mkdir -p /app/tmp

# Run the application and keep the container running
CMD ["sh", "-c", "python3 example_microservice.py && tail -f /dev/null"]
```

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

```
import os
```

```
# Path to temporary directory within the mounted tmpfs
tmp_dir = "/app/tmp"

def process_sensitive_data(data):
    # Ensure the temporary directory exists.
    if not os.path.exists(tmp_dir):
        os.makedirs(tmp_dir)

    # Write sensitive data to a temporary file within the mounted tmpfs
    with open(os.path.join(tmp_dir, "sensitive_data.txt"), "w") as f:
        f.write(data)

    # Perform processing on the sensitive data
    with open(os.path.join(tmp_dir, "sensitive_data.txt"), "r") as f:
        processed_data = f.read()
        print(f"Processed Data: {processed_data}")

# Example usage
data_to_process = "This is sensitive data"
process_sensitive_data(data_to_process)
```

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 .
```

alt text

```
docker run --rm -d \
    --mount type=tmpfs,dst=/app/tmp,tmpfs-size=16k,tmpfs-mode=1770 \
    my_microservice
```

alt text

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
alt text

So we have successfully completed the task!

explain these for a simple child in a very easiest manner in details with basics also correct any error

Understanding In-Memory Storage with Docker (Made Simple for Beginners)

Imagine you are working on a secret project and you have a diary where you write your secrets. You don't want to save this diary in your room because someone might find it. Instead, you keep it in your mind and erase it once you're done. This is exactly how **in-memory storage** works!

Now, let's go step by step to understand it clearly.

What is In-Memory Storage?

Think of your computer's memory (RAM) like a whiteboard. You can write things on it, but as soon as you turn off the power, everything is erased. **In-memory storage** uses this idea to store important and secret information **temporarily** in the computer's memory instead of saving it permanently on the hard drive.

This is useful because:

- **It keeps data safe** (Hackers can't find it later because it's erased when the system stops).
- **It is fast** (Reading and writing data in memory is quicker than on a hard drive).

How Do We Use In-Memory Storage in Docker?

Docker is like a special lunchbox that holds everything needed to run a small program (called a **container**). But instead of storing secret things inside the lunchbox (which someone could steal), we use **tmpfs mounts**, which act like a magic notebook that disappears when the container is closed.

Step 1: Using `tmpfs` Mounts in Docker

A **tmpfs mount** is like a temporary sticky note where you write information, and it vanishes when you close the program.

To tell Docker to use this, we use this command:

```
bash

docker run --rm \
    --mount type=tmpfs,dst=/tmp \
    --entrypoint mount \
    alpine:latest -v
```

Breaking Down the Command

- `**docker run --rm**`: Start a new container and delete it when done.
- `**--mount type=tmpfs,dst=/tmp**`: This creates a special memory-only folder inside the container at `**/tmp**`.
- `**alpine:latest**`: This is the lightweight Linux system used in Docker.
- `**--entrypoint mount**`: This makes the program check where things are stored.

What Happens?

- A new storage area (`**/tmp**`) is created in memory inside the Docker container.
- Anything placed inside `**/tmp**` disappears when the container stops.

Step 2: Customizing In-Memory Storage

We can set rules for the temporary storage:

- **How big it should be** (`**tmpfs-size**`).
- **Who can access it** (`**tmpfs-mode**`).

For example, this command creates a **16 KB memory storage** that only certain users can read:

```
bash

docker run --rm \
    --mount type=tmpfs,dst=/app/tmp,tmpfs-size=16k,tmpfs-mode=1770 \
    --entrypoint mount \
    alpine:latest -v
```

Step 3: A Simple Example – Secret Microservice

Imagine we are building a small service (**microservice**) that writes secret messages into memory instead of saving them on the computer.

Creating the Dockerfile

A **Dockerfile** is like a recipe that tells Docker how to build the container.

```
dockerfile

# Use Python as the base system
FROM python:3.9-slim

# Set the work folder inside the container
WORKDIR /app

# Copy the program into the container
COPY . .

# Make sure the tmpfs folder exists
RUN mkdir -p /app/tmp

# Start the service and keep the container running
CMD ["sh", "-c", "python3 example_microservice.py && tail -f /dev/null"]
```

Writing the Secret Microservice Code

Create a Python file (`example_microservice.py`) that will store and read secret data.

```
python

import os

# Folder for temporary secret storage
tmp_dir = "/app/tmp"

def process_sensitive_data(data):
    # Make sure the folder exists
    if not os.path.exists(tmp_dir):
        os.makedirs(tmp_dir)

    # Write the secret data in memory (not on disk)
    with open(os.path.join(tmp_dir, "sensitive_data.txt"), "w") as f:
        f.write(data)

    # Read the secret data back and print it
    with open(os.path.join(tmp_dir, "sensitive_data.txt"), "r") as f:
        processed_data = f.read()
        print(f"Processed Data: {processed_data}")

# Example usage
data_to_process = "This is secret data"
process_sensitive_data(data_to_process)
```

What This Program Does?

1. **Creates a secret storage folder (`/app/tmp`)** in memory.
2. **Writes secret data** into a temporary file inside this folder.
3. **Reads and prints** the secret data.
4. **Since it's stored in memory, it disappears when the container stops!**

Step 4: Running the Microservice

Now, we need to **build and run** our secret microservice.

Building the Docker Image

```
bash

docker build -t my_microservice .
```

This command tells Docker to follow the **Dockerfile** and create a container named `my_microservice`.

Running the Docker Container with In-Memory Storage

```
bash

docker run --rm -d \
    --mount type=tmpfs,dst=/app/tmp,tmpfs-size=16k,tmpfs-mode=1770 \
    my_microservice
```

- `--rm -d`: Runs the container in the background and removes it after stopping.
- `--mount type=tmpfs,dst=/app/tmp,...`: Uses **in-memory storage** instead of saving to disk.

Step 5: Checking if Everything Works

To check that our service is storing data in memory:

1. **See if the tmpfs storage is mounted** inside the container:

   ```
   bash

   docker inspect <container_id_or_name>
   ```

   This will show information about the container, including the temporary memory storage.

2. **Check if the secret file exists inside memory storage**:

   ```
   bash

   docker exec -it <container_id_or_name> /bin/bash
   cd /app/tmp && ls
   cat sensitive_data.txt
   ```

   - `docker exec -it <container_id> /bin/bash`: Opens a shell inside the running container.
   - `cd /app/tmp && ls`: Lists files in the temporary memory folder.
   - `cat sensitive_data.txt`: Displays the stored secret.

Since this storage is in memory, once we **stop the container**, the secret disappears!

Final Thoughts

✅ **We successfully built a secure way to store sensitive data without writing to disk.**
✅ **Our data is safe and disappears when the container stops.**
✅ **Using `tmpfs` keeps our microservice fast and secure.**

🔹 **Real-life use?**
This is used in **banking, password management, and secure APIs** where sensitive data should not be

stored permanently.

🚀 **Next Step?**
Try experimenting with different memory sizes and access controls using `tmpfs-size` and `tmpfs-mode`!

I corrected a few small mistakes and made everything super simple. Let me know if you need more explanations! 😊