Container Network - Docker Stop vs Kill

Differentiating Docker stop vs Kill

Differentiating Docker stop vs kill
This lab scenario provides a comprehensive overview of the differences between Docker's stop and kill commands, including their behavior, use cases, and practical examples. Understanding these differences is crucial for proper container management, especially when ensuring that applications shut down gracefully.

Introduction
In Docker, both stop and kill commands are used to terminate a running container, but they work differently:

docker stop: Sends a SIGTERM signal first, allowing the process to clean up before termination. If the process doesn't terminate within a specified timeout (default 10 seconds), Docker sends a SIGKILL signal to forcefully stop the process.

alt text

docker kill: Sends a SIGKILL signal immediately, terminating the process without allowing any clean-up.

alt text

Using docker stop
Step 1: Run a Container with a Process that Handles SIGTERM
Run a container with an Ubuntu image, simulating a process that handles the SIGTERM signal:

docker run --name graceful-termination -d ubuntu:latest /bin/bash -c "trap 'echo SIGTERM received; exit 0' SIGTERM; while :; do echo 'Running'; sleep 1; done"
Breakdown of the Command
docker run:

The primary Docker command to create and start a new container.
--name graceful-termination:

Names the container graceful-termination for easier management and reference.
-d:

Runs the container in detached mode, meaning it will run in the background.
ubuntu:latest:

Specifies the Docker image to use, in this case, the latest version of the Ubuntu image.
/bin/bash -c:

Starts a Bash shell within the container to execute the following script.
"trap 'echo SIGTERM received; exit 0' SIGTERM; while :; do echo 'Running'; sleep 1; done":

trap 'echo SIGTERM received; exit 0' SIGTERM: Sets up a trap to catch the SIGTERM signal. When this signal is received, the message "SIGTERM received" is printed and the script exits with a status of 0, indicating a graceful shutdown.
while :; do echo 'Running'; sleep 1; done: An infinite loop that prints "Running" every second. This keeps the container running until it is stopped.
Step 2: Stop the Container
Use the docker stop command to terminate the container:

docker stop graceful-termination
Step 3: Check the Container Logs
Examine the logs to verify how the process inside the container handled the SIGTERM signal:

docker logs graceful-termination
Expected Logs
Running
Running
Running
SIGTERM received
The logs should show that the process received the SIGTERM signal, executed the trap command, printed "SIGTERM received," and exited gracefully.

alt text

Using docker kill
Step 1: Run a Container with a Process that Would Handle SIGTERM
Run a similar container as before:

docker run --name force-termination -d ubuntu:latest /bin/bash -c "trap 'echo SIGTERM received; exit 0' SIGTERM; while :; do echo 'Running'; sleep 1; done"
Step 2: Kill the Container
Use the docker kill command to terminate the container:

docker kill force-termination
Step 3: Check the Container Logs
Examine the logs to verify the behavior of the process inside the container when terminated by SIGKILL:

docker logs force-termination
Expected Logs
Running
Running
Running
The logs should show continuous "Running" messages without any indication of "SIGTERM received," demonstrating that the process was abruptly terminated by the SIGKILL signal.

alt text

Summary of Differences

Signal Sent:

stop: Sends SIGTERM first, then SIGKILL if the process doesn't terminate within the timeout.
kill: Sends SIGKILL immediately.
Graceful Shutdown:

stop: Allows the process to handle the signal and perform clean-up operations.
kill: Does not allow any handling or clean-up.
Use Case:

stop: Preferred for gracefully shutting down applications.
kill: Used for forcefully terminating unresponsive containers.
By examining the container logs, we can clearly see the difference in behavior. The stop command allows the application to handle the termination signal, whereas the kill command results in immediate termination without any clean-up.

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

Sure! Let's break this down in the **simplest way possible**, starting from the **very basics**, and also correct any unclear parts.

🔍 What are `docker stop` and `docker kill`?

Both commands are used to **stop a running container**, but they do it in **different ways**.

🚦 Basics of Linux Signals (Important to Understand This!)

When you run a program (like in a Docker container), you can send it a **signal** to tell it to do something — like stop, pause, etc.

Two Important Signals:

- **SIGTERM (Signal Terminate)**
  - 🔹 A gentle way to ask a program to stop.
  - 🔹 Gives the program a chance to **clean up**, save data, or log something before shutting down.
- **SIGKILL (Signal Kill)**
  - 🔹 A **forceful** way to stop a program — it **instantly shuts it down**, no questions asked.
  - 🔹 The program doesn't get a chance to clean up.

🛑 `docker stop` — Gentle Shutdown

What it does:

1. Sends **SIGTERM** to the container.
2. Waits for a few seconds (default is 10 seconds).

3. If the program **still doesn't stop**, it sends **SIGKILL** to force-stop it.

Why it's useful:

- It gives your app a chance to **save work** or **shut down properly**.

💀 `docker kill` — Forceful Shutdown

What it does:

- **Immediately sends SIGKILL**.
- The container stops **instantly** — **no cleanup**, no warning.

Why it's used:

- When the container is **stuck** or **unresponsive**, and you want to stop it right away.

👨‍🔬 Lab Example (With Explanation)

Let's try both commands with a container that can handle `SIGTERM`.

🔧 Step-by-Step: Using `docker stop`

Step 1: Run a container that listens for `SIGTERM`

```bash
docker run --name graceful-termination -d ubuntu:latest /bin/bash -c "trap 'echo SIGTERM received;
```

🔍 What this does:

- `trap 'echo SIGTERM received; exit 0' SIGTERM`:
  If it receives a SIGTERM, it prints `"SIGTERM received"` and exits cleanly.
- `while :; do echo 'Running'; sleep 1; done`:
  Keeps printing "Running" every second — keeps the container alive.

Step 2: Stop the container

```bash
docker stop graceful-termination
```

Step 3: View the logs

```bash
docker logs graceful-termination
```

🧾 Expected Output:

```sql
Running
Running
Running
SIGTERM received
```

✅ This shows it received SIGTERM and shut down **gracefully**.

🔧 Step-by-Step: Using `docker kill`

Step 1: Run another container (same as before)

```bash
docker run --name force-termination -d ubuntu:latest /bin/bash -c "trap 'echo SIGTERM received; exi
```

Step 2: Kill the container

```bash
docker kill force-termination
```

Step 3: Check the logs

```bash
docker logs force-termination
```

🧾 Expected Output:

```sql
Running
Running
Running
```

❌ No "SIGTERM received" because it never got SIGTERM — it was killed **instantly** with SIGKILL.

📊 Summary Table: `stop` vs `kill`

Feature	`docker stop`	`docker kill`
Signal Sent	First SIGTERM, then SIGKILL	SIGKILL only
Graceful Shutdown	✅ Yes	❌ No

Feature	`docker stop`	`docker kill`
Cleanup Allowed	✅ Yes	❌ No
Use When	App should shut down properly	App is stuck or unresponsive
Default Timeout	10 seconds (can be changed)	No timeout, instant kill

🧠 Final Thoughts

- Always try `docker stop` first — it's polite!
- Use `docker kill` only when necessary.
- Understanding these helps in **managing containers safely**, avoiding **data loss or corruption**.

Let me know if you want a visual diagram or a short cheat sheet for this too!