Activity 7: Advanced Docker Concepts (Bonus)

🎯 **Objective:** Explore advanced Docker internals and ecosystem topics with deeper theory and hands-on practice. This activity is designed for students who want to understand how Docker really works under the hood. **No submission or grading**: this is a bonus lab for self-learning.

✅ Prerequisites

- Ubuntu 24.04 LTS EC2 instance with Docker installed and running.
- SSH access (with `secret-key.pem` and EC2 public IP).
- Comfort with Linux CLI and `sudo`.
- Curiosity to explore how Docker integrates with Linux kernel and OCI standards.

📂 Lab setup

SSH into your EC2 instance:

```shell
ssh -i secret-key.pem ubuntu@<EC2_PUBLIC_IP>
```

Switch to root for system-level inspection:

```shell
sudo -i
```

All commands below are to be run on the EC2 host.

📌 Docker Daemon & Engine API

Theory

Docker has two key components:

1. **Docker daemon (`dockerd`)**: long-running background process managing containers, images, networks, and volumes.
2. **Docker CLI (`docker`)**: a client tool that talks to the daemon.
3. **Docker API**: REST API exposed by the daemon over a Unix socket (`/var/run/docker.sock`) or optionally a TCP socket.

The CLI (`docker ps`, `docker run`, etc.) is just a wrapper that makes HTTP requests to the Docker API. For example, `docker ps` calls `GET /containers/json`.

This separation means **you can build your own Docker clients** using HTTP calls.

👉 Docs: [Docker Engine API]

Hands-on

Check daemon status:

```shell
ps aux | grep dockerd
systemctl status docker --no-pager
```

Inspect daemon logs:

```shell
journalctl -u docker -n 20
```

Curl the API directly:

```shell
# list running containers (like docker ps)
curl --unix-socket /var/run/docker.sock http://localhost/containers/json | jq
```

Create a container via API:

```shell
# Pull image
curl -s --unix-socket /var/run/docker.sock \
  -X POST \
  "http://localhost/images/create?fromImage=busybox&tag=latest"

#Create container
curl -s -X POST --unix-socket /var/run/docker.sock \
  -H "Content-Type: application/json" \
  -d '{"Image":"busybox","Cmd":["echo","hello-from-API"]}' \
  http://localhost/containers/create?name=api-test | jq

# Start container
curl -s --unix-socket /var/run/docker.sock \
  -X POST \
  http://localhost/containers/api-test/start
```

Check logs:

```shell
curl -s --unix-socket /var/run/docker.sock \
  "http://localhost/containers/api-test/logs?stdout=1&stderr=1&timestamps=0" |
jq -R -s .
```

📌 Container Internals: Namespaces, Cgroups, Capabilities

Theory

Linux kernel features make containers possible:

- **Namespaces**: isolate resources:
    - `pid` (process IDs),
    - `net` (network interfaces),
    - `mnt` (mount points),
    - `ipc`, `uts`, `user`. Each container has its own "view" of processes, networking, and filesystems.
- **Control groups (cgroups)**: control and account resources (CPU, memory, IO, pids). Cgroups ensure a container cannot exceed resource quotas.
- **Capabilities**: fine-grained kernel privileges (instead of full root). E.g., `CAP_NET_ADMIN`, `CAP_SYS_ADMIN`. By default, containers drop some dangerous capabilities for safety.

👉 Docs:

- [Namespaces in Linux]
- [Cgroups v2]
- [Linux capabilities]

Hands-on

Run a container and find its PID:

```shell
docker run -d --name ns-test busybox sleep 300
PID=$(docker inspect -f '{{.State.Pid}}' ns-test)
echo $PID
```

Check its namespaces:

```shell
ls -l /proc/$PID/ns
```

Check its cgroups:

```shell
sudo cat /proc/$PID/cgroup
```

Use `lsns` (if installed) to view namespace assignments:

```shell
sudo lsns -p $PID
```

Try a container with dropped capabilities:

```shell
docker run -it --rm --cap-drop ALL busybox sh -c "id; ps aux"

# Observe difference in capability bitmask
docker run --rm busybox sh -c 'echo "CapEff (no drop):"; awk "/CapEff/ {print
\$2}" /proc/1/status'
docker run --rm --cap-drop ALL busybox sh -c 'echo "CapEff (cap-drop):"; awk
"/CapEff/ {print \$2}" /proc/1/status'
```

Clean up:

```shell
docker rm -f ns-test
```

📌 Storage Drivers & Image Layers

Theory

Docker images and containers are built on **union file systems** using storage drivers.

- Default driver: **overlay2** (on Ubuntu ≥ 18.04).
- Each image consists of **read-only layers**.
- A container adds a **thin writable layer** on top.

This layering is what allows caching, efficient pulls, and small incremental changes.

👉 Docs: [Docker storage drivers]

Hands-on

Check driver in use:

```shell
docker info | grep "Storage Driver"
```

See disk usage:

```shell
docker system df
```

Inspect `/var/lib/docker/overlay2` (do not modify):

```shell
sudo ls /var/lib/docker/overlay2 | head
```

View image history:

```Shell
docker history alpine:3.18
```

Export a container's filesystem:

```Shell
docker export $(docker create alpine) | tar -tvf - | head
```

📌 Rootless Docker

Theory

Rootless Docker runs the Docker daemon and containers entirely without root privileges.

Why important?

- Extra security (mitigates daemon privilege escalation).
- Useful in shared systems without root access.

How it works:

- Uses **user namespaces** to map container root → host non-root UID.
- Requires `slirp4netns` for networking and `fuse-overlayfs` for storage.

👉 Docs: [Rootless mode]

Hands-on (conceptual only)

Check if your current daemon is rootless:

```Shell
docker info | grep Rootless
```

To install rootless Docker (don't run in production without care):

```Shell
curl -fsSL https://get.docker.com/rootless | sh
```

Then run the daemon as your user:

```Shell
systemctl --user start docker
export DOCKER_HOST=unix:///run/user/$UID/docker.sock
```

(We won't do full setup here; just awareness.)

📌 Docker Init (Scaffolding a Project)

Theory

Since **Docker 25.0**, a new command `docker init` was introduced to quickly scaffold a **starter `Dockerfile`** and `.dockerignore` for your project.

It analyzes your project's source code (Node.js, Python, Go, .NET, etc.) and generates a basic containerization setup.

👉 Docs: [docker init reference]

Why useful?

- Eases onboarding for beginners.
- Provides best-practice base images for common languages.
- Saves time writing boilerplate Dockerfiles.

Cautions:

- The generated Dockerfile is **generic**, not optimized (e.g., no multi-stage builds, no cache tuning).

- It may include unnecessary layers or packages.
- Treat it as a **starting point only**, not production-ready.
- Always review, optimize, and add security hardening before using in production.

Hands-on

1. Create a sample Node project:

```shell
mkdir ~/demo-node && cd ~/demo-node
npm init -y
echo "console.log('Hello Docker Init!')" > app.js
```

2. Run Docker init:

```shell
docker init
```

It will prompt you with questions like:

- Which language? (Node, Python, Go, etc.)
- Which package manager?
- Default port?

It then generates:

- `Dockerfile`
- `.dockerignore`

3. Inspect the generated Dockerfile:

```shell
cat Dockerfile
```

4. Build & run the scaffolded image:

```Shell
docker build -t demo-node .
docker run --rm demo-node
```

Example Output (Node.js)

```None
# Generated by docker init
FROM node:18
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

Why it matters

- For teaching/demo projects, `docker init` lowers the barrier.
- For production, it's a **teaching tool**: you should gradually evolve the Dockerfile to include **multi-stage builds, non-root users, healthchecks, and slim base images** (as we did in Activity 6).

📌 OCI Standards: Docker vs Containerd vs CRI-O

Theory

The **Open Container Initiative (OCI)** standardizes:

1. **Image format** (OCI Image Spec).
2. **Runtime format** (OCI Runtime Spec).

This lets multiple runtimes and tools interoperate.

- **Docker** (CLI + daemon + build + containerd + runc).
- **containerd**: lightweight core container runtime (used inside Docker and Kubernetes).
- **CRI-O**: Kubernetes-focused runtime, minimal.
- **runc**: reference implementation of the OCI runtime (low-level, executes containers).

👉 Docs:

- OCI Image Spec
- OCI Runtime Spec
- Containerd
- CRI-O

Hands-on

Check if containerd is running:

```
Shell
systemctl status containerd
```

Try `ctr` (containerd CLI):

```
Shell
sudo ctr images pull docker.io/library/alpine:3.18
sudo ctr run --rm -t docker.io/library/alpine:3.18 testctr sh -c "echo hello
from containerd"
```

📌 Kernel Interfaces: /proc, /sys/fs/cgroup, unshare, runc

Theory

Containers are just Linux processes with namespaces and cgroups.

- `/proc` shows process details and namespace links.
- `/sys/fs/cgroup` shows cgroup controllers and usage.
- `unshare` creates a new namespace manually.
- `runc` is the low-level binary that executes containers from OCI bundles (used under Docker).

👉 Docs:

- [runc OCI runtime]
- [man 2 unshare]

Hands-on

Inspect a container's namespaces:

```shell
docker run -d --name alpine1 alpine sleep 60
PID=$(docker inspect -f '{{.State.Pid}}' alpine1)
ls -l /proc/$PID/ns
```

Inspect cgroups:

```shell
cat /proc/$PID/cgroup
```

Try `unshare`:

Shell
```
sudo unshare --fork --pid --mount-proc bash
ps aux   # shows only processes inside new PID namespace
exit
```

Try `runc` (low-level run):

Shell
```
mkdir -p /tmp/bundle/rootfs
cd /tmp/bundle
docker export $(docker create alpine) | tar -C rootfs -xf -
runc spec
sudo runc run testcontainer
```

Clean up:

Shell
```
docker rm -f alpine1
```

🧹 Cleanup

Shell
```
docker ps -aq | xargs docker rm -f
docker system prune -af
```

📖 Suggested further reading

- Docker Engine overview
- Namespaces and cgroups deep dive (LWN)
- Rootless Docker guide
- OCI Specifications

🏆 Conclusion

This bonus activity gives you a “peek under the hood” of Docker: daemon & API, kernel integration, storage drivers, rootless mode, OCI ecosystem, and low-level runtimes.

The goal is not to memorize commands but to **connect the dots**: Docker is just tooling on top of Linux features (namespaces, cgroups, capabilities) and OCI standards.