Think of Docker as a tool that allows you to package an application with all of its dependencies (libraries, system tools, code, runtime) into a standardized unit called a **container**. This container will run the same way anywhere Docker is installed, eliminating the classic "but it worked on my machine!" problem.

Here is a curated list of essential Docker commands, organized from basic to more advanced, to get you up and running.

The Docker Cheat Sheet for an Intern

We'll break this down into logical sections.

1. The Basics: Docker Lifecycle

The core workflow is: **Build** an image -> **Run** a container from that image.

Command	Description	Example
`docker --version`	Check your Docker installation version.	`docker --version`
`docker info`	Display system-wide information about Docker.	`docker info`
`docker run <image>`	**Pull an image** (if not found locally) and **start a container** from it. This is the most common command.	`docker run hello-world`
`docker ps`	List **running** containers.	`docker ps`
`docker ps -a`	List **all** containers (including stopped ones). Crucial for debugging.	`docker ps -a`
`docker stop <container>`	Stop a running container gracefully. You can use the container ID or name.	`docker stop peaceful_kepler`
`docker start <container>`	Start a stopped container.	`docker start peaceful_kepler`
`docker rm <container>`	**Remove a stopped container.** Keep your system clean!	`docker rm peaceful_kepler`
`docker rm $(docker ps -aq)`	**Force remove all containers** (running or stopped). **Use with caution!**	`docker rm -f $(docker ps -aq)`
`docker images`	List all images on your local machine.	`docker images`
`docker rmi <image>`	Remove a specific image from your local machine.	`docker rmi nginx`

2. Running Containers Interactively

This is how you *learn* and *debug*. You get a shell inside the container.

Command	Description	Example
`docker run -it <image> <shell>`	Run a container **interactively** (`-i`) and allocate a pseudo-TTY (`-t`), then run a shell.	`docker run -it ubuntu /bin/bash`
`docker exec -it <container> <shell>`	Run a command **inside a already-running** container.

Essential for debugging. | `docker exec -it my_nginx /bin/bash` |
| `exit` | (From inside the container's shell) Exit the shell, which stops the container if it was started with `-it`. | `exit` |

3. Managing Ports and Volumes (The "How to Connect" Part)

Containers are isolated. These commands break that isolation in a controlled way to allow networking and data persistence.

Command	Description	Example
`docker run -p <host_port>:<container_port>`	**Map a port** from the host machine to the container. How you access web apps.	`docker run -p 8080:80 nginx`
`docker run -v <host_path>:<container_path>`	**Mount a volume**. Link a directory on your host to a directory in the container. Data persists even if the container is deleted.	`docker run -v /my/data:/data ubuntu`
`docker run --name <name>`	Give your container a custom name instead of a random one (like `peaceful_kepler`).	`docker run --name my_nginx nginx`

4. Building Your Own Images with `Dockerfile`

This is where the real power is. You define your environment as code.

First, create a file named `Dockerfile` (no extension) in your project directory.

Dockerfile Instruction	Description
`FROM <base_image>`	The base image to start from (e.g., `ubuntu`, `python`, `node`).
`COPY <src> <dest>`	Copy files from your host machine into the image.
`RUN <command>`	Execute a command **during the build process** (e.g., `RUN apt-get update`).
`WORKDIR <path>`	Set the working directory for any `RUN`, `CMD`, `ENTRYPOINT` instructions.
`EXPOSE <port>`	Document which port the container listens on (doesn't actually publish it).
`CMD ["executable", "param1"]`	The default command to run **when the container starts**.

Command	Description	Example
`docker build -t <name:tag> <path>`	**Build an image** from a `Dockerfile`. `-t` tags it with a name. The `.` is the build context (current directory).	`docker build -t my-app:1.0 .`

5. Viewing Logs and Inspecting

Command	Description	Example
`docker logs <container>`	Fetch the logs of a container.	`docker logs my_nginx`
`docker logs -f <container>`	**Follow** the logs (like `tail -f`). Great for real-time debugging.	`docker logs -f my_nginx`
`docker inspect <container/image>`	Return low-level information (JSON) on a container or image. Lots of details!	`docker inspect my_nginx`

Your First Practical Task

Let's put it all together. We'll run an Nginx web server, customize it, and see it work.

1. **Run a basic container:**

   docker run -d --name my-web-server -p 8080:80 nginx

   * `-d` runs it in detached mode (in the background).
   * `--name` gives it a friendly name.
   * `-p 8080:80` maps your machine's port 8080 to the container's port 80.

2. **Verify it's working:**
   * Open your browser and go to `http://localhost:8080`. You should see the "Welcome to nginx!" page.
   * Run `docker ps` to see your container running.

3. **Look at the logs:**

   docker logs my-web-server

   You should see the Nginx access logs.

4. **Get a shell inside the container:**

   docker exec -it my-web-server /bin/bash

   Now you are *inside* the container! You can run commands like `ls /usr/share/nginx/html` to see the web files. Type `exit` to leave.

5. **Clean up:**

   docker stop my-web-server
   docker rm my-web-server

Pro Tips for Your Internship:

1. **Always Clean Up:** Get in the habit of stopping and removing containers you don't need. Use `docker ps -a` weekly to check for forgotten containers.
2. **Read the Logs:** 90% of debugging "why didn't my container start?" is solved by running `docker logs <container_name>`.
3. **Start Simple:** Use official images from Docker Hub (like `nginx`, `redis`, `python`) before building your own complex ones.
4. **Understand the Dockerfile:** The `Dockerfile` is the recipe. Mastering it is key to mastering Docker.

This is your foundation. Play with these commands, break things, and fix them. Once you're comfortable here, we'll move on to the next big topic: **Docker Compose** (for running multi-container apps).

Level 2: The Docker Mindset of a Senior Engineer

1. The Theory: Immutability and the Single Concern Principle

A container is not a tiny VM. This is the most important mental model shift.

* **Immutable Infrastructure:** A running container is **immutable**. You don't SSH into it and `apt-get upgrade` something. If you need a change, you rebuild the *image* (the blueprint) from the `Dockerfile`, redeploy a new container from that updated image, and kill the old one. This guarantees consistency and rollbacks are trivial.
* **Single Concern:** Each container should do **one thing and do it well** (e.g., run your app, run a database, run a cache). This makes them easy to scale, debug, and secure.

2. Advanced Command Usage: Efficiency and Insight

These commands separate juniors from seniors. They're about getting deep insight and saving time.

Command & Flags	Why a Senior Engineer Uses It
`docker system df`	**Quick diagnosis of disk usage.** Shows how much space your images, containers, and volumes are *really* using. The first command I run when my disk is full.
`docker logs --tail 50 -f <container>`	**Tail the last 50 lines and follow.** You don't always need the entire log history since the dawn of time. Get to the recent events fast.
`docker exec -it <container> sh`	**Use `sh` instead of `bash`.** Many minimalist base images (like `alpine`) don't have `bash`. `sh` is more universally available and gets the job done.
`docker inspect --format='{{.NetworkSettings.IPAddress}}' <container>`	**Extract specific information in a usable format.** `docker inspect` outputs a huge JSON blob. Using `--format`

(Go templates) lets you grep for specific details (IP, status, log path) programmatically. |
| `docker run --rm <image>` | **Run a container and automatically remove it when it exits.** Perfect for one-off tasks or testing. Prevents container clutter. `docker run --rm -it ubuntu /bin/bash` |
| `docker run -e VARIABLE=value` | **Inject environment variables at runtime.** This is how you make your containerized application configurable for different environments (dev, staging, prod) without rebuilding the image. Critical for 12-factor apps. |

3. Image Building: The Art of the Dockerfile

This is where great DevOps engineers are made. A bad image is bloated, slow, and insecure.

The Golden Rules of Dockerfile Creation:

1. **Use a `.dockerignore` file:** Exactly like `.gitignore`. Prevents sending unnecessary files (like `node_modules`, local `.env` files, build logs) to the Docker daemon. Makes builds faster and more secure.
2. **Leverage the Build Cache:** Order your commands from **least frequently changed to most frequently changed**. This is the most important optimization.
```dockerfile
# BAD: Code is copied first. Every code change breaks the cache for apt-get!
COPY . /app
RUN apt-get update && apt-get install -y python3

# GOOD: Dependencies are installed first. Code, which changes often, is last.
RUN apt-get update && apt-get install -y python3
COPY . /app
```
3. **Chain `RUN` commands:** Each `RUN` instruction creates a new layer. Combine them to reduce image size.
```dockerfile
# BAD: Creates multiple layers, increasing image size.
RUN apt-get update
RUN apt-get install -y python3
RUN rm -rf /var/lib/apt/lists/*

# GOOD: One layer. The cleanup happens in the same layer as the install.
RUN apt-get update && \
    apt-get install -y python3 && \
    rm -rf /var/lib/apt/lists/*
```
4. **Use Small Base Images:** Don't start with `ubuntu` if you can use `debian:slim`. Don't use `debian` if you can use `alpine`. smaller image = faster downloads, smaller attack surface, faster startup.
```dockerfile
FROM python:3.9-slim  # Instead of `python:3.9`
# or
```

```
FROM alpine:3.16
```
5. **Run as a Non-Root User:** By default, containers run as root. This is a security risk.
```dockerfile
RUN groupadd -r myuser && useradd -r -g myuser myuser
USER myuser
COPY --chown=myuser:myuser . /app
```

4. Beyond Single Containers: The Ecosystem

Docker alone isn't DevOps. It's a component.

* **Docker Compose:** You'll use this daily for local development. It defines and runs multi-container applications with a single `docker-compose.yml` file. It's the next thing you must learn.
* **Orchestration (Kubernetes, Docker Swarm):** This is how you run containers in production. Docker is the tool to *build* the container, but an orchestrator is what manages thousands of them across a cluster of machines, handling scaling, networking, and self-healing. This is the ultimate destination.

5. Debugging Theory: The Three-Step Process

When a container misbehaves:

1. **Inspect the Logs:** `docker logs -f <container>`. 80% of issues are solved here.
2. **Inspect the Configuration:** `docker inspect <container>`. Is the command correct? Are the environment variables set? Are the ports mapped correctly?
3. **Shell In:** `docker exec -it <container> sh`. This is your last resort. Check the filesystem. Is the file where it's supposed to be? Can you run the binary manually? This confirms if the problem is in the runtime environment or the application itself.

Your Senior-Level Task

1. **Build an optimized image:** Take a simple Python app (a Flask hello world). Build it three ways:
   * `FROM python:3.9`
   * `FROM python:3.9-slim`
   * `FROM python:3.9-alpine` (you may need to install build tools)
   Compare the final image sizes (`docker images`). See the dramatic difference?
2. **Break the cache:** Intentionally structure a `Dockerfile` poorly (copy code first, then install dependencies). Make a change to your code and rebuild. Notice how the build is slow. Now fix the `Dockerfile` and rebuild again. See how it uses the cache and is nearly instantaneous?
3. **Inject configuration:** Run your app with `docker run -e FLASK_ENV=production` and have the application code read that variable to change its behavior.

This mindset—thinking in terms of immutable, single-concern units, optimizing for security and size, and understanding the broader ecosystem—is what makes a DevOps engineer truly valuable. Welcome to the next level. Let me know when you're ready to talk about Docker Compose and container orchestration.