Chapter 5: What Is a Docker Image?

Introduction

In this chapter, we'll take a deep dive into Docker images. The goal is to give you a clear, practical understanding of what Docker images are, how to work with them through essential operations, and what's happening behind the scenes when you use them.

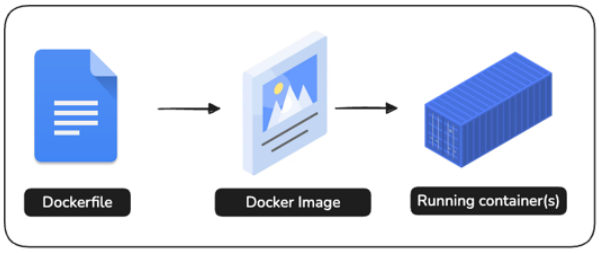

Figure #: From Dockerfile to Container

This diagram shows the standard Docker workflow for creating a running application. It starts with a Dockerfile inside your project, which contains all the instructions needed to build an image. In the previous chapter, you learned how the Docker Engine works at a low level — the CLI, the Engine API, the Daemon, BuildKit, and Buildx. That foundation now becomes essential, because those components are exactly what turn your Dockerfile into a real image.

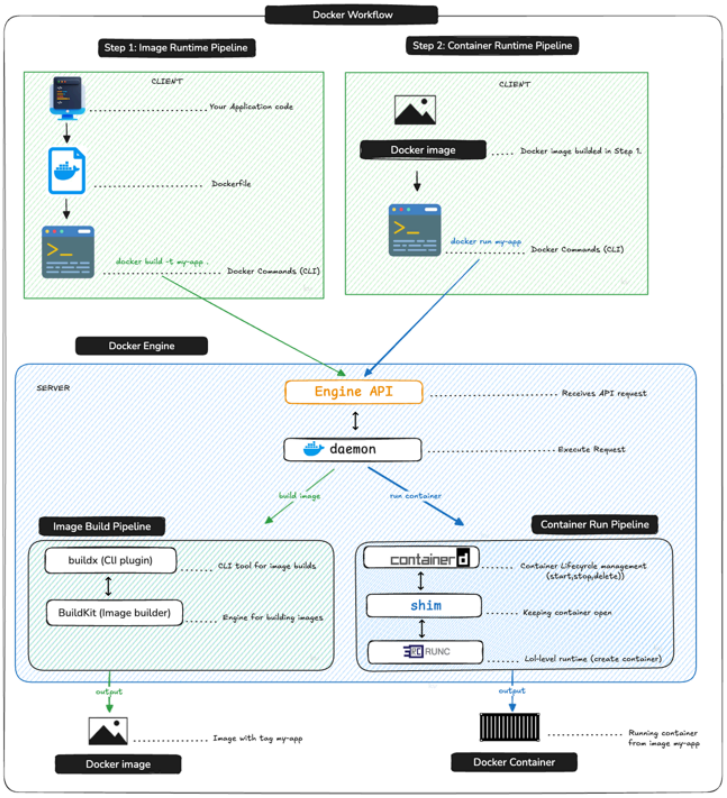

Figure 17: Docker Workflow

In the Docker Workflow, you can see the full path from source code to a running application.
When you run a build command and your project includes a Dockerfile, the Docker CLI hands the request off to the Docker API, which then passes each instruction to the Docker Daemon for execution. The Daemon passes the job to BuildKit, which reads your Dockerfile line by line, builds each layer, and produces the final Docker image — a portable blueprint of your application.

In Step 2 of the figure, you can see that running `docker run my-app` takes the image builded in Step 1 and starts a live instance of it—a Docker container. This is the moment your application actually comes to life.

From there, the workflow becomes simple: **Dockerfile → Image → Container.**

Understanding this pipeline shows you exactly why each part of the Docker Engine exists and how they work together to turn your code into a running application.

Later in the book, we'll take this even further by building image for real front-end React.js sample applications — including a full production-ready setup deployed to AWS.

> **Dockerfile**
> The Dockerfile is the recipe that tells Docker exactly how to build an image — layer by layer, step by step. Whenever you run `docker build`, Docker reads the instructions in this file and uses them to create a new image from scratch.It's the *recipe* that defines how an image is built — layer by layer, step by step. Every time you run `docker build`, Docker reads your Dockerfile and creates a new image based on those instructions. We will explore the Dockerfile in depth in next chapters.

Docker images - The deep dive

An **image** is like a blueprint, while a **container** is the actual app running from that blueprint. You can think of images as instructions, and containers as the real, working result of those instructions.

- **Images** exist at build time. They define everything your app needs: code, dependencies, and configuration.
- **Containers** exist at run time. They are live, running instances of your app created from an image.

You can use commands like `docker run` to start one or more containers from the same image. Once a container is running, it depends on the image it was created from. Because of this, Docker won't let you remove an image if there are still containers using it. To delete an image, you must first stop and remove all of its containers.

This simple relationship is important: images are reusable blueprints, and containers are temporary instances from this image. You can start, stop, and remove containers freely, but the image stays the same until you decide to build, update or delete it.

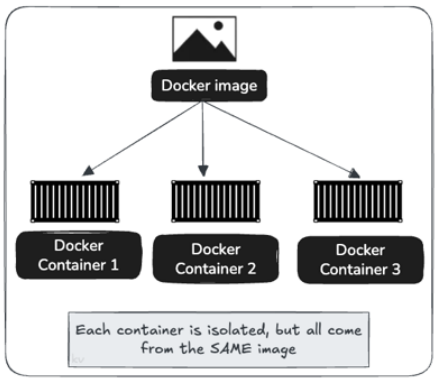

Figure 18: mage as Blueprint → Many Running Containers

Images Are Usually Small

A container is designed to run a single application or service. Because of this, it only needs the application code and its required dependencies — nothing more. As a result, images are usually small and stripped of all unnecessary parts.

For example, the official Alpine Linux image is only about 7 MB. It's so small because it doesn't include multiple shells, package managers, or other extras. In fact, many images don't include a shell or a package manager at all — if the application doesn't need it, it's not included.

Another important detail is that images don't contain a kernel. Containers share the kernel of the host machine they run on. This means the only operating system parts included in an image are minimal: a few essential filesystem components and other basic constructs.

Windows-based images, however, tend to be much larger than Linux-based ones. This is simply due to the way Windows is built. It's common for Windows images to be several gigabytes in size, which makes them slower to push and pull compared to lightweight Linux images.

Pulling Images

When you first install Docker, your host has no images stored locally. On Linux, the local image repository is usually found under:

```
/var/lib/docker/<storage-driver>
```

On Mac or Windows with Docker Desktop, everything runs inside a virtual machine, but the concept is the same.

To check what images you currently have, use:

```
docker images
```

If no images are listed, you'll need to **pull** them. Pulling is the process of downloading images from a registry (like Docker Hub) onto your host.

For example, let's pull the official Node.js image based on Alpine Linux:

```
docker pull node:24.7.0-alpine

```

You'll see output similar to this:

```
24.6.0-alpine3.22: Pulling from library/node
b5d25b35c1db: Pull complete
6970efae6230: Pull complete
fea4afd29d1f: Pull complete
Digest:
sha256:9a4c2b52a16e4a2f4f98b1f7d10d8fcd8a184e94e4bcd1e3af9d16f07b75d2a1
Status: Downloaded newer image for node:24.7.0-alpine
docker.io/library/node:24.7.0-alpine
```

Now, if we check our images again:

```
docker images
```

Output:

```
REPOSITORY   TAG                IMAGE ID       CREATED       SIZE
node         24.6.0-alpine3.22  44dd6f223004   10 days ago    234MB

```

As you can see, the `node:24.7.0-alpine` image is now stored locally, ready to be used for containers. Notice how small it is — 55.9 MB is the **compressed transfer size** (what you download).
The 234 MB you saw is the uncompressed size on disk after Docker extracts the image layers. — thanks to being based on Alpine Linux.

On the other hand, Windows-based images are often much larger. This makes them slower to download and push compared to lightweight Linux images.

Image Registry

When we pulled an image earlier, we had to specify which image to download. To understand this better, let's look at how image naming works.

Images are stored in central locations called **registries**. Registries make it easy to share, access, and manage images.

The most common registry is **Docker Hub** (Docker Hub Container Image Library | App Containerization), but there are also:

- **Third-party registries** (like GitHub Container Registry (GHCR) , AWS Elastic Container Registry (ECR) ,Google Artifact Registry, Azure Container Registry (ACR))
- **Private registries** (used inside organizations for security and control).

By default, the Docker client uses **Docker Hub** when you pull or push images. That's why, unless you specify otherwise, Docker assumes your image lives on Docker Hub.

Image naming and tagging

Every Docker image has two main parts: a **repository name** and a **tag**, written in the format:

```
<repository>:<tag>

```

Pulling Official Images

If the image comes from an official repository on Docker Hub, you just provide the repository name and tag. For example, let's pull the official Node.js image:

```
docker image pull node:24.7.0-alpine

```

- `node` → the repository (official Node.js images).
- `24.6.0-alpine3.22` → the tag (this version of Node.js built on Alpine Linux 3.22).

You can also pull the default `latest` tag:

```
docker image pull node:latest
```

Or simply:

```
docker image pull node
```

Since no tag is provided, Docker assumes you want `latest`. But be careful — `latest` **doesn't always mean newest**. For example, while `node:latest` might point to a stable release, the newest version might be tagged differently (like `node:current`).

Best practice

Always use **explicit version tags** (e.g. `node:24.7.0-alpine`) instead of vague ones like `latest` or even `node:24`.

1. **Reproducible Builds**
   - With a fixed tag, the same Dockerfile will build the same image tomorrow, next week, or next year.
   - Floating tags (`latest`, `node:24`) can silently change upstream, introducing new bugs or incompatibilities without you touching your code.
2. **Predictable Deployments**
   - CI/CD pipelines and production clusters rely on consistency.
   - If one server pulls `node:24` today and another pulls it next week, they might end up with different runtimes. That's a recipe for "works on my machine."
3. **Easier Debugging & Rollbacks**
   - Pinning lets you trace exactly which environment your app was built against.
   - If a regression happens, you can roll back to a known-good version without guessing what "latest" meant last Tuesday.
4. **Security & Compliance**

- In regulated or enterprise environments, you often need to prove exactly which base image was used.
- Pinning a tag makes audits and vulnerability scans precise and trustworthy.

5. **Layer Caching & Build Performance**
   - Docker layer caching works best when the base image tag is stable.
   - Using moving tags like `latest` can invalidate caches unnecessarily, slowing down builds.

Pulling from Personal Repositories

Images in personal or organization repositories require a namespace. For example:

```
docker image pull myusername/react-app:dev

```

This pulls the image tagged `dev` from the `react-app` repository owned by the user `myusername`.

Pulling from Other Registries

Not all images live on Docker Hub. For example, if your team uses Google Container Registry (GCR), you must include the registry domain name:

```
docker image pull gcr.io/myproject/node:24.6.0
```

This pulls version `24.6.0` of Node.js from your project's private GCR repository.

Images with Multiple Tags

A single image can have multiple tags pointing to the same image ID. For example, you could pull all tags of your custom Node.js image:

```
docker image pull -a myusername/react-app
docker image ls
```

Output might look like this:

```
REPOSITORY              TAG              IMAGE ID       SIZE
myusername/react-app    latest           d5e1e48cf932   700MB
myusername/react-app    v2               d5e1e48cf932   700MB
myusername/react-app    v1               6852022de69d   680MB

```

Here, both `latest` and `v2` share the same **IMAGE ID**, meaning they're just two tags for the same image.

Again, this shows why you shouldn't blindly rely on `latest`. It could point to an older version. Always tag and pull explicitly.

Images and layers

A Docker image is made up of **read-only layers**. Each layer represents the result of one instruction in your Dockerfile (like `FROM`, `RUN`, or `COPY`). Docker stacks these layers together and exposes them as a single, unified image.

- Layers are **cached**, so if nothing changes in a step, Docker reuses the existing layer. This makes builds faster.
- Layers are **shared**, meaning different images can reuse the same base layers (e.g., node:24.7.0-alpine), which saves disk space.

When you pull an image, Docker downloads it layer by layer. That's why you'll see messages like *"Pulling fs layer"* for each step. Inspecting those layers helps you understand how an image is constructed and where optimizations can be made.

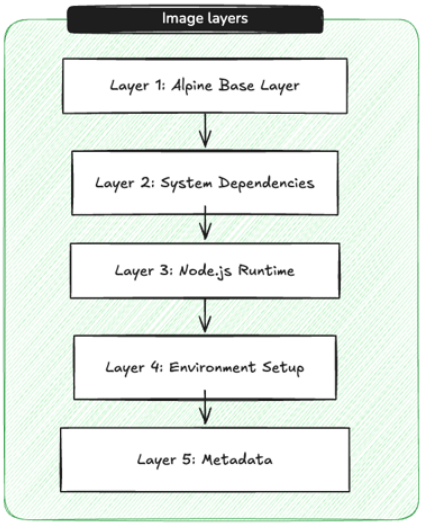

Figure 19: Docker image layers

Here's roughly what you can expect to see:

1. **Alpine Base Layer**
   - Small Linux distribution (`alpine:3.22`).
   - Contains only the bare minimum (busybox, musl, apk).
   - Keeps the image tiny (a few MB).
2. **System Dependencies**
   - Installs required tools for Node.js (like `serve` , `python3` , `vite`).
   - These are needed to compile native npm packages.
3. **Node.js Runtime**
   - Adds the Node.js binaries (24.7.0-alpine).
   - Includes `npm` and sometimes `corepack` for managing `yarn` and `pnpm` .
4. **Environment Setup**
   - Sets environment variables like `NODE_VERSION` and `PATH` .
   - Ensures `/usr/local/bin/node` is available globally.
5. **Final Metadata Layer**
   - Adds labels, default `CMD ["node"]` , and other metadata.
   - This is what makes the image behave like "just run Node."

When running:

```
docker pull node:24.7.0-alpine
```

Docker outputs something like:

```
24.6.0-alpine3.22: Pulling from library/node
Digest:
sha256:51dbfc749ec3018c7d4bf8b9ee65299ff9a908e38918ce163b0acfcd5dd931d9
Status: Image is up to date for node:24.7.0-alpine
```

```
4 docker.io/library/node:24.7.0-alpine
```

You won't always see *exact* human-friendly descriptions, but behind the scenes each "Pulling fs layer" corresponds to one of the steps above.

To actually inspect the layers, you can run:

```
1 docker history node:24.7.0-alpine
```

Which gives output like:

```
1  IMAGE          CREATED       CREATED BY
   SIZE      COMMENT
2  51dbfc749ec3   2 weeks ago   CMD ["node"]
   0B        buildkit.dockerfile.v0
3  <missing>      2 weeks ago   ENTRYPOINT ["docker-entrypoint.sh"]
   0B        buildkit.dockerfile.v0
4  <missing>      2 weeks ago   COPY docker-entrypoint.sh /usr/local/bin/
   # …   20.5kB    buildkit.dockerfile.v0
5  <missing>      2 weeks ago   RUN /bin/sh -c apk add --no-cache --
   virtual …   5.47MB    buildkit.dockerfile.v0
6  <missing>      2 weeks ago   ENV YARN_VERSION=1.22.22
   0B        buildkit.dockerfile.v0
7  <missing>      2 weeks ago   RUN /bin/sh -c addgroup -g 1000 node
   && …   161MB     buildkit.dockerfile.v0
8  <missing>      2 weeks ago   ENV NODE_VERSION=24.6.0
   0B        buildkit.dockerfile.v0
9  <missing>      6 weeks ago   CMD ["/bin/sh"]
   0B        buildkit.dockerfile.v0
10 <missing>      6 weeks ago   ADD alpine-minirootfs-3.22.1-
   aarch64.tar.gz …   9.17MB    buildkit.dockerfile.v0
```

Every time you use `node:24.7.0-alpine`, you're not downloading "one big thing." You're pulling **a stack of layers**: Alpine → system deps → Node.js runtime → metadata.
This layering is what makes Docker fast and efficient, because cached layers are reused across projects.

Multi-architecture images

Docker started out simple: one image, one platform. But as the ecosystem grew, developers needed images that could run on different CPU architectures (like **x64** and **ARM**) and operating systems (**Linux** and **Windows**).

The problem? A single image tag could no longer mean the same thing everywhere. Without a solution, you'd have to worry about picking the right variant yourself — breaking the smooth Docker workflow.

Architecture vs. Platform

- **Architecture** = CPU type (e.g., `x64`, `ARM`, `PowerPC`, `s390x`).
- **Platform** = Operating system (`Linux`, `Windows`) or OS + architecture together (e.g., `linux/arm64`).

The Solution: Multi-Arch Images

With **multi-architecture images**, a single tag (like `node:24.7.0-alpine`) can represent **different builds** for different platforms and architectures.

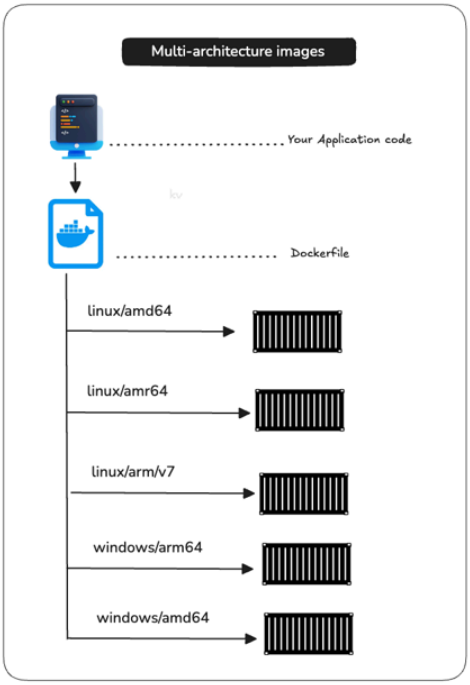

Figure 20: multi-architecture images

Take the official `node:24.7.0-alpine` image. Behind this single tag, there are different builds: Linux on x64, Linux on ARM64 (for Apple Silicon/M1/M2 chips), and others. When you pull the image, Docker figures out which one matches your system and fetches only that build.

How Docker Chooses the Right Node.js Image

Let's say you're on a MacBook with Apple Silicon (ARM64). When you pull `node:24.7.0-alpine`, Docker does the following:

1. Downloads the **manifest list** for the image from Docker Hub.
2. Finds the entry for **Linux/ARM64**.
3. Retrieves the manifest for that platform.
4. Downloads and assembles the layers for the ARM64 variant of Node.js.

The same process works on a Linux server running x64 — Docker automatically pulls the x64 version instead.

Example: Same Command, Different Platforms

Linux on ARM64 (Apple Silicon / Raspberry Pi):

```
docker run --rm node:24.7.0-alpine node -v
v24.6.0
```

Linux on x64 (standard server/PC):

```
docker run --rm node:24.7.0-alpine node -v
v24.6.0
```

Both commands are identical. Docker just gives you the right binary for your platform.

Inspecting the Node Image Manifest List

You can check which platforms are supported with:

```
docker manifest inspect node:24.7.0-alpine | grep 'architecture\|os'

```

The output (shortened) shows multiple supported platforms:

```
  "architecture": "amd64",
  "os": "linux"
  "architecture": "arm64",
  "os": "linux",
  "architecture": "s390x",
  "os": "linux"
```

This means the `node:24.7.0-alpine` image supports at least Linux/x64 and Linux/ARM64.

Building Your Own Multi-Arch Images

As a front-end developer, you might package your React or Next.js app into a Docker image. With `docker buildx`, you can publish a multi-arch image so it runs on both ARM (MacBooks, Raspberry Pi) and x64 (servers, CI/CD). For example:

```
docker build --platform linux/amd64,linux/arm64 -t my-frontend:latest .

```

This creates one tag (`my-frontend:latest`) that supports both architectures — no more juggling different builds.

That's why you can run:

```
docker run node:24.7.0-alpine node -v
```

on your MacBook, your Windows WSL, or your Linux CI server, and it just works. The manifest list makes sure you always get the right Node.js binary for your environment.

Delete Images

At some point in your development workflow, your local machine will accumulate many Docker images. Each time you run `docker pull` or build a new image with `docker build`, Docker stores it on your host. Over time, this can consume a significant amount of disk space. Cleaning up unused images is a normal part of Docker maintenance.

Docker provides the `docker rmi` command for this purpose. The name `rmi` comes from **remove image**. When you remove an image, Docker will:

- **Un-tag the image**: the friendly name like `node:24.7.0-alpine` is removed.
- **Delete the image layers**: the actual underlying data stored on your machine is deleted.
- **Preserve shared layers**: if another image depends on the same layers, Docker keeps them until no image needs them.

> **Important**
>
> If an image is in use by a running container, Docker will refuse to delete it. This prevents you from accidentally breaking a container that depends on that image. To fully clean up, you need to stop and remove the container before deleting the image.
>
> ***To stop a running container, you can use***: docker stop <container_id_or_name>

Deleting an Image by ID

Every Docker image has a unique identifier, known as the **IMAGE ID**. You can find it with:

```
$ docker images
REPOSITORY   TAG                    IMAGE ID       CREATED
SIZE
node         24.6.0-alpine3.22      44dd6f223004   2 weeks ago
234MB

```

Here, the Node.js image we pulled has the ID `44dd6f223004`.

To delete the image, run either of the following commands:

```
$ docker image rm 44dd6f223004
```

or, equivalently:

```
$ docker rmi 44dd6f223004
```

Sample output:

```
Untagged: node:24.7.0-alpine
Deleted: sha256:44dd6f2230041eede4ee5e7...
```

What happens here?

1. Docker first **removes the tag** `node:24.7.0-alpine`.
2. It then **removes the layers** associated with this image.
3. If those layers are shared with another image, they won't be deleted until all references are gone.

Now, if you run `docker images` again, the image is no longer listed.

This method is handy when scripting automation pipelines because IDs are always unique and precise.

Deleting an Image by Name

Typing long IDs every time you want to remove an image can be tedious. That's why most developers prefer to delete images by their **repository name and tag**:

```
$ docker rmi node:24.7.0-alpine
```

or, using the modern way with `docker image rm <image>`:

```
$ docker image rm node:24.7.0-alpine
```

This works the same way as deleting by ID. However, be aware that if multiple tags point to the same image ID (e.g., `node:24.7.0-alpine` and `node:latest`), Docker will only remove the tag you specified. The underlying layers will remain until all tags are gone.

This approach is more human-friendly and is what you'll typically use in day-to-day development.

Deleting Multiple Images at Once

React.js developers often test with multiple Node.js versions (for example, Node 22 and Node 24). You may want to clean them all up in one go.

By name:

```
$ docker rmi node:24.7.0-alpine node:22-alpine3.19

```

By ID:

```
$ docker rmi 44dd6f223004 3f5ef9003cef
```

Docker will process each one in sequence. If one cannot be deleted because it's in use, Docker will skip it but still remove the others.

Output looks like this:

```
Untagged: node:24.7.0-alpine
Deleted: sha256:44dd6f223004...
Untagged: node:22-alpine3.19
Deleted: sha256:3f5ef9003cef...

```

Deleting All Images at Once

Over time, your Docker host may accumulate a large number of images — some you still use, others left over from experiments or old builds. Docker gives you two main ways to clean them up, depending on whether you want a **safe cleanup** or a **full reset**.

Option 1: Safe Cleanup with `docker image prune -a`

The command:

```
$ docker image prune -a
```

What it does:

- Removes **unused images** (images not referenced by any container).
- Leaves behind any images that are still in use by existing containers.
- Frees up disk space without breaking anything currently running.

This is the option you'll use most often in day-to-day development. For example, if you've been switching between multiple Node.js base images (`node:22` , `node:24` , etc.) and don't need the old ones anymore, this command will clean them up safely.

Option 2: Full Reset with `docker rmi $(docker images -q) -f`

Sometimes, though, you want a **completely clean slate** — for example, if you've been experimenting heavily, your disk is full, or you want to reset your machine before a fresh workshop or demo. In that case, you can wipe **all images** at once:

```
$ docker rmi $(docker images -q) -f
```

Here's what's happening:

- `docker images -q` → lists only image IDs.
- `$(...)` → passes that list into the next command.
- `-f` → forces deletion, even if the image has multiple tags.

Sample run:

```
Untagged: node:24.7.0-alpine
Deleted: sha256:44dd6f223004...
Untagged: node:22-alpine3.19
Deleted: sha256:3f5ef9003cef...

```

Checking again:

```
$ docker images
REPOSITORY   TAG   IMAGE ID   CREATED   SIZE
```

Empty. Your host is clean.

⚠ On **Windows**, this command works in PowerShell but not in CMD.

How to fix the Error Image Is in Use ?

If you try deleting an image that's powering a container, you'll see:

```
$ docker rmi node:24.7.0-alpine
Error response from daemon: conflict: unable to remove repository
reference "node:24.7.0-alpine" (must force) - container abc123 is using
its referenced image
```

Fix: Stop and remove the container first:

```
$ docker ps
$ docker stop abc123
$ docker rm abc123
$ docker rmi node:24.7.0-alpine
```

Image Debugging

Debugging Docker images is often necessary when containers fail to run, dependencies are missing, or the environment isn't behaving as expected. This section provides practical ways to inspect and troubleshoot Docker images and their containers.

1. Run a Container with an Interactive Shell

One of the simplest ways to debug an image is to run a container interactively. This allows you to explore the filesystem, check installed packages, and manually test commands.

Start a container from the image and drop into a shell:

```
docker run -it --rm <image_name> /bin/sh

```

If the image includes **bash** (common in Debian/Ubuntu-based images):

```
docker run -it --rm <image_name> /bin/bash

```

- The `-it` flag ensures you get an interactive terminal.
- The `--rm` flag removes the container once you exit, so you don't leave unused containers behind.
- Useful for checking directory structure (`ls` , `cd`), testing binaries, or verifying configuration files.

2. Debug an Existing Running Container

If the container is already running and you need to peek inside, use `docker exec` to attach to it. This is especially useful if you want to debug a live service without restarting it.

```
docker exec -it <container_id_or_name> /bin/sh

```

or:

```
docker exec -it <container_id_or_name> /bin/bash
```

- Allows you to inspect logs, environment variables, or running processes.
- Example: running `ps aux` inside can show you if your service is actually running.
- You can combine this with `docker logs <container_id>` to troubleshoot startup errors.

3. Inspect Image Metadata

The `docker inspect` command gives you full details about an image or container in JSON format.

```
docker inspect <image_name>
```

This output includes:

- **Entrypoint and CMD**: What command runs when the container starts.
- **Environment Variables**: Defaults baked into the image.
- **Volumes**: Directories marked for persistence.
- **Exposed Ports**: What ports the container listens on.

Understanding these details helps you check if your container is starting with the correct configuration.

4. Explore Image Layers

Docker images are built in layers, each representing a command in the Dockerfile. If something breaks, looking at the history helps you pinpoint where.

```
docker history <image_name>
```

You'll see:

- The `Dockerfile` command that created each layer.
- The size of each layer.
- Timestamps of creation.

This is useful for:

- Finding which step installed (or failed to install) a dependency.
- Identifying unnecessarily large layers (e.g., a forgotten cache).
- Verifying that your `COPY` or `RUN` commands are applied as expected.

5. Export and Explore Filesystem

Sometimes you want to dig deeper into the container's contents outside of Docker. You can export the container filesystem and explore it locally:

```
docker export <container_id> > container.tar
```

Then extract the `.tar` file with:

```
tar -xf container.tar
```

This allows you to:

- Inspect files even if the container no longer runs.
- Compare configurations between builds.
- Use your local tools (e.g., IDE, grep) to search inside the container's filesystem.

6. Common Debugging Tips

Here are a few additional tricks for everyday debugging:

- **Check Environment Variables**:
  Inside the container, run:

```
printenv
```

  or

```
echo $ENV_VAR_NAME
```

  to confirm configuration values.

- **Verify Installed Packages**:
  Use the package manager:

```
apt list --installed | grep <package_name>
```

```
yum list installed | grep <package_name>
```

  This confirms if a dependency is missing.

- **Test Network Access**:
  Run:

```
curl http://google.com
```

  to confirm internet or service connectivity.

- **Rebuild Without Cache**:
  If you suspect Docker's cache is causing problems, rebuild clean:

```
docker build --no-cache -t <image_name> .
```

- **Check Startup Logs**:

```
docker logs <container_id>
```

  This is often the first step when a service fails to start.

Essential Docker Image Commands

Command	Description	React.js Developer Use Case
`docker image build -t my-`	Build an image from a `Dockerfile` in the	Package your React app (after `npm run`

`react-app:dev .`	current directory. The `-t` flag adds a name/tag.	`build`) into an image for local testing or deployment.
`docker image ls`	List all images currently on your host.	Quickly check which Node.js and React app images you already have.
`docker image pull node:24.7.0-alpine`	Download an image from a registry (default: Docker Hub).	Get the official Node.js base image to build and run your React app.
`docker image rm my-react-app:dev`	Remove one or more images from your host.	Clean up old builds or outdated Node.js images taking up disk space.
`docker image prune -a`	Remove unused images (`-a` removes all unused, not just dangling).	Free up disk space after lots of rebuilds or switching Node.js versions.
`docker image tag my-react-app:dev myusername/my-react-app:v1.0.0`	Add a new tag pointing to the same image.	Version your React app image for releases in CI/CD.
`docker image push myusername/my-react-app:v1.0.0`	Push an image to a registry.	Share your React app image with teammates or deploy to production.
`docker image inspect my-react-app:dev`	Show detailed JSON about an image: layers, config, env, ports.	Verify what's inside your React app image before deploying.

`docker image history node:24.7.0-alpine`	Show how an image was built, layer by layer.	Troubleshoot large image sizes or learn how Node.js images are structured.
`docker image save my-react-app:dev -o react-app.tar`	Save an image as a tar archive.	Export your React app image to move between environments (useful in offline setups).
`docker image load -i react-app.tar`	Load an image from a tar archive.	Import your saved React app image into another system without pulling from a registry.
`docker image import my-app.tar`	Create an image directly from a tarball filesystem.	Rarely used — but handy for restoring app files as an image without a Dockerfile.

Chapter Summary

In this chapter, we took a deep dive into Docker images — the blueprints for containers. We learned that images package everything needed to run an app: your code, dependencies, the runtime (like Node.js), and just enough of an operating system to stay lightweight and portable.

We saw that images are built in layers, making them efficient to reuse and share across projects. We compared images (blueprints) to containers (running instances) and explored why lightweight images like Alpine are common in front-end development, while Windows images are much larger.

We also looked at registries like Docker Hub, third-party cloud registries, and private registries, and we covered how image naming and tagging works. Best practice: always pin exact versions (e.g., `node:24.7.0-alpine`) instead of relying on `latest`. This ensures reproducible builds, predictable deployments, and easier debugging.

We examined how Docker supports multi-architecture images, allowing the same tag to run seamlessly on different platforms (Linux, ARM, Windows).

Finally, we worked with the key image management commands:

- Pulling (`docker pull`)
- Listing (`docker images`)
- Deleting by ID or name (`docker rmi`)
- Removing multiple or all images at once
- Handling errors when an image is still in use

Next: Choosing the Right Node.js Image

Not all Node.js images are created equal.
In the next chapter, we'll look at how to choose the right Node.js base image for your project — whether you need the smallest possible image for production or a more complete one for development.

You'll learn when to use `alpine`, `slim`, or full images, how version tags work, and how to balance image size, speed, and compatibility for your React or front-end builds.
