

# AI-03024 Docker FAQ

Owner	© Rigel Arcayan
Tags	Project
Created time	@September 23, 2025 6:32 AM

**What is Docker? Unpack the creation of Docker based applications. Enumerate and explain the main structure, sub structure of each component and how files, data, networking, db and static items collected, stored and served.**

Docker is an open-source platform that uses OS-level virtualization to package and run applications in isolated, lightweight environments called containers. Containers are different from traditional virtual machines (VMs) because they share the host operating system's kernel, making them much more efficient and portable.

## Creation of Docker-Based Applications 🛠️

The process of creating a Docker-based application, often called "dockerizing," involves a few key steps:

- **Writing a Dockerfile:** This is a text file that contains all the commands needed to build a Docker image. It acts as a blueprint, specifying everything from the base operating system to the application's dependencies and how to run the application.
- **Building a Docker Image:** You use the Docker command `docker build` to execute the instructions in the Dockerfile. Each instruction in the Dockerfile creates a new, read-only layer in the image. This layered approach is a key feature of Docker, as it allows for efficient sharing and caching of common layers, reducing storage space.
- **Running a Docker Container:** A Docker container is a runnable instance of a Docker image. You use the `docker run` command to start a container. When a container runs, a new, writable layer is added on top of the image's read-only layers. This is where all changes and new data are stored.

## Structure of a Docker Application 📦

A Dockerized application isn't a single file but a collection of interconnected components, all managed by the Docker Engine. The main components are:

- Docker Daemon (dockerd): This is the background service that runs on the host machine. It's the "brain" of Docker, responsible for building, running, and managing all Docker objects, including images, containers, networks, and volumes.
- Docker Client: This is the command-line interface (CLI) that users interact with. When you type a command like `docker run`, the client sends an API request to the Docker Daemon.
- Docker Registry: This is a centralized repository for storing and distributing Docker images. Docker Hub is the most common public registry, but you can also use private registries.

#### File, Data, and Static Item Management

##### Files and Data Storage

Docker has a "Copy-on-Write" (CoW) system for managing files within a container. This means that an image's layers are read-only. When a container is running and a file is modified or created, that change is written to the new, writable layer on top. This is great for a container's ephemeral nature, but for data you want to save permanently, Docker provides different storage options:

- Volumes: This is the recommended way to persist data. Volumes are stored in a dedicated, managed part of the host's filesystem, outside of the container's writable layer. They are independent of the container's lifecycle, meaning the data survives even if the container is removed. Volumes are perfect for databases, application data, and sharing data between multiple containers.
- Bind Mounts: These directly map a directory or file from the host machine into a container. This gives the container direct access to the host's filesystem, which is useful for tasks like sharing configuration files or working on code live during development. However, bind mounts are less portable and secure than volumes.

#### Networking

Docker provides several networking drivers to allow containers to communicate with each other and the outside world. The most common drivers are:

- **Bridge Network:** This is the default network for standalone containers. It creates a private, internal network on the host machine. Containers on the same bridge network can communicate with each other by using their container names or IP addresses.
- **Host Network:** This removes the network isolation between the container and the host. A container using the host network shares the host's IP address and can access services on the host machine, which can be useful for performance but sacrifices isolation.
- **Overlay Network:** This is used for networking across multiple Docker hosts and is crucial for multi-host applications like those managed by Docker Swarm.

#### Database and Static Content Serving

- **Databases:** To ensure data persistence, databases are almost always run in a container with a volume mounted to store the database files. This ensures that the data is not lost when the container is stopped, updated, or replaced.
- **Static Items:** Static files like HTML, CSS, and JavaScript can be served in a few ways:
  - **Inside the Container:** You can copy the static files into the Docker image as part of the build process. A web server like NGINX or Apache is then used to serve these files from within the container.
  - **Bind Mounts:** For development, you can use a bind mount to map your static files from your host machine into the web server container. This allows for live updates as you make changes to your files without needing to rebuild the image.
  - **Dedicated Static Server:** In production, it's common to have a separate container (often an NGINX container) that is specifically responsible for serving static files, which can improve performance and security.

### **List the common docker generic commands I need to know and read about.**

Here are the most common Docker commands you should know, broken down by function.

#### 1. Image Management

- `docker pull <image-name>[:tag]` 📦
  - Purpose: Downloads a Docker image from a registry (like Docker Hub) to your local machine.
  - Example: `docker pull nginx:latest` downloads the latest version of the NGINX web server image.
- `docker images` 🖼️
  - Purpose: Lists all the Docker images currently stored on your local machine.
- `docker rmi <image-name>[:tag]` 🗑️
  - Purpose: Removes a local Docker image. Use this to free up disk space.
  - Example: `docker rmi hello-world` removes the hello-world image.
- `docker build -t <image-name>[:tag] .` 🛠️
  - Purpose: Builds a new Docker image from a Dockerfile in the current directory (.). The `-t` flag tags the image with a name and optional tag.
  - Example: `docker build -t my-app:1.0 .` builds an image from the Dockerfile and names it my-app with the 1.0 tag.

## 1. Container Lifecycle

- `docker run <image-name>[:tag]` 🚀
  - Purpose: This is the most fundamental command. It creates and starts a new container from a specified image. It's often used with flags to customize the container's behavior.
  - Example: `docker run -d -p 8080:80 nginx` runs a container in detached mode (`-d`), maps port 80 inside the container to port 8080 on the host machine (`-p`), and uses the nginx image.
- `docker ps` 📄
  - Purpose: Lists all currently running containers.
  - Flags: Use `docker ps -a` to list all containers, including those that have been stopped.

- `docker stop <container-id or name>` 🛑
  - Purpose: Gracefully stops a running container. Docker sends a SIGTERM signal to the container's main process, allowing it to shut down cleanly.
- `docker start <container-id or name>` ▶️
  - Purpose: Starts a container that has been previously stopped.
- `docker rm <container-id or name>` ❌
  - Purpose: Removes a stopped container. You can't remove a running container unless you use the -f (force) flag.

## 1. Interacting with Containers

- `docker exec -it <container-id or name> <command>` 🏃
  - Purpose: Runs a command inside a running container. This is extremely useful for debugging.
  - Flags: -i keeps STDIN open, and -t allocates a pseudo-TTY, which allows you to interact with the shell.
  - Example: `docker exec -it my-app-container bash` opens a bash shell inside the container named my-app-container.
- `docker logs <container-id or name>` 📖
  - Purpose: Fetches the logs from a container, which is crucial for troubleshooting.
  - Flags: `docker logs -f <container-id>` follows the log output in real time.
- `docker cp <source-path> <destination-path>` 📁
  - Purpose: Copies files or directories between a container and the local filesystem.
  - Example: `docker cp my-app-container:/usr/src/app/log.txt .` copies a file from the container to your current directory.

## 1. System & Cleanup

- `docker info` ⓘ

- Purpose: Displays system-wide information about your Docker installation, including the number of images, containers, and volumes.
- docker system prune ✨
  - Purpose: Cleans up your system by removing all unused Docker data, including stopped containers, unused networks, and dangling images. This is the single best command for freeing up disk space.

**If I want to start over and redo my docker project, list the commands that I need to know/run without losing my personal data.**

To completely restart a Docker project without losing your personal data, you need to be careful about what you delete. The key is to remove containers and images but preserve your volumes, which is where your persistent data is stored. Here are the commands you should run:

- Stop all running containers:  
`docker stop $(docker ps -a -q)`  
 This command finds the IDs of all running and stopped containers (`docker ps -a -q`) and passes them to the `docker stop` command.
- Remove all stopped containers:  
`docker rm $(docker ps -a -q)`  
 This command removes all containers that are not currently running.
- Prune all unused containers, networks, and images (without removing volumes):  
`docker system prune`  
 This is a powerful command that safely cleans up dangling and unused resources. By default, it will not remove volumes, which is exactly what you want. You will be prompted to confirm the removal.  
 If you want to be more specific or aggressive with cleanup, you can use these individual commands:
- To remove all unused networks:  
`docker network prune`
- To remove all stopped containers:  
`docker container prune`

- To remove dangling images (images with no tag and not used by any container):  
docker image prune
  - To remove all unused images (not just dangling ones):  
docker image prune -a
- After running these commands, your Docker environment will be clean of all project-related containers and images, but the data you stored in your volumes will be safe and sound, ready for you to rebuild your project from a clean slate.

Clarify the following about docker

1. A db container can be accessed by other app containers
2. A network container can be accessed by other app container
3. When a db container is running, I can self-manage the inside db
4. When I turn of compose down, a container, my db stay safe and I can clone the folder where it's being saved
5. My data in docker project is safe just like a non docker setup which I can backup at will.
6. Most companies now runs on docker apps?

Great set of clarifying questions 👍 Let's walk through each point carefully and address what's true, what needs nuance, and how it works in practice.

### **1. A DB container can be accessed by other app containers**

✅ True.

- As long as they're on the same Docker network, containers can communicate using their service name as the hostname.
- Example: If your docker-compose.yml defines db and wordpress in the same network, WordPress can connect to db:3306 just like a normal database.

### **2. A network container can be accessed by other app containers**

⚠️ Needs clarification.

- There's no such thing as a dedicated "network container" in Docker — instead, you create Docker networks.
- Multiple containers attached to the same network can talk to each other.
- If you meant "reverse proxy container" (like Nginx/Traefik), yes, it can route traffic to other app containers.

### **3. When a DB container is running, I can self-manage the inside DB**

 True.

- You can exec into it:

```
docker exec -it mydb mysql -u root -p
```

- Or connect via an external DB client (like DBeaver, HeidiSQL, pgAdmin).
- It behaves just like a "normal" database install.

### **4. When I turn off**

**docker-compose down**

**, my DB stays safe and I can clone the folder where it's being saved**

 Partially true.

- If you're using named volumes or bind mounts, your data is safe when containers are removed.
- But if you don't configure persistence (e.g., just `docker run postgres` without a volume), the DB data will be lost once the container is removed.
- Cloning the folder where the volume is stored is a valid backup method.

### **5. My data in a Docker project is safe just like a non-Docker setup which I can backup at will**

 Yes, if persistence is configured.

- With volumes or bind mounts, Docker databases are just as safe as bare-metal installs.
- You can back them up by:
  - Dumping (`mysqldump`, `pg_dump`)

- Copying the data volume (docker cp, or volume folder)

## 6. Most companies now run on Docker apps?

✅ Largely true, with nuance.

- Docker (or containerization in general) is the industry standard for deploying modern apps.
- Most enterprises use Docker, but often wrapped inside Kubernetes (or OpenShift, AWS ECS, etc.) for scaling and orchestration.
- Some still run apps on bare metal or VMs, especially legacy systems, but the trend is heavily toward containers.

✅ Direct Answer Summary:

1. Yes, DBs are accessible across containers via networks.
2. Docker networks allow app-to-app communication.
3. You can manage the DB inside as usual.
4. Data persists only if volumes/bind mounts are used.
5. With persistence, Docker DBs are as safe as traditional setups.
6. Yes, containers (Docker/Kubernetes) dominate in modern companies.

👉 Would you like me to draw you a simple diagram of how app containers, DB containers, and Docker networks fit together (so you can visualize the relationships)?

## List 22 most used/popular docker apps in 2025

Here are ~22 of the most-used / popular apps (services) that people often run in Docker (especially in production or self-hosted setups) as of 2025, plus some commentary. If you want, I can break them into categories (DB, cache, media, dev tools, etc.).

# Most Popular Docker Apps & Images (2025)

These are common Docker images / apps used frequently across companies, developers, hobbyists, and self-hosters.

#	App / Image	Category / What It Does
1	NGINX	Web server / reverse proxy / load-balancer. Common for fronting apps, proxies, static file serving.
2	Redis	In-memory key/value store, cache, sometimes pub/sub.
3	PostgreSQL	Relational database.
4	MySQL / MariaDB	Relational databases, very common.
5	MongoDB	NoSQL document store.
6	Elasticsearch	Search & analytics engine.
7	RabbitMQ	Message broker queues, decoupling services.
8	Kafka	Distributed streaming / event log.
9	Zookeeper	Used often alongside Kafka (for coordination), other distributed systems.
10	Tomcat	Java servlet container / app server.
11	Logstash / Kibana	Part of the ELK (or more broadly) stack for logs processing, visualization.
12	MinIO	S3-compatible object storage. (Often used for file storage in Docker setups)
13	Nacos	Service discovery / config server (especially in microservices / Java ecosystems).
14	Portainer	UI / dashboard for managing Docker containers. Very popular among self-hosters and small/medium setups.
15	Traefik	Dynamic reverse proxy / load balancer especially good with microservices and autoscaling / certificate management.
16	Plex	Media streaming server. People often run Plex in Docker for home media servers.

17	Jellyfin	Open-source self-hosted media streaming alternative.
18	Sonarr / Radarr / Lidarr etc ("arr" apps)	Media automation: managing TV / Movies / Music download-and-organize workflows. Often used together in media stacks.
19	Overseerr / Ombi / Jellyseerr	Media request / catalog / UI to allow users to request movies/shows.
20	qBittorrent / Transmission / other torrent / download clients	For self-hosted downloading of media. Frequently dockerized.
21	Tautulli	Monitoring usage/statistics of media servers (especially Plex).
22	Home / Dashboard apps (Heimdall, Homarr, Flame, Organizr, etc.)	They provide dashboard/start-pages to access your other services/containers etc. Often run in Docker.

## Notes & Trends

- The mix above includes both backend infrastructure (DBs, caches, brokers), media/self-hosted apps, and tooling for management / dashboard.
- According to Docker's 2025 State of App Development survey, the most used data stores in Docker setups are MongoDB, MySQL/MariaDB, and close behind that is Amazon RDS; PostgreSQL used to be more dominant but this year shift happened.
- In many self-hosting / homelab communities, "arr" stack + media streaming + dashboards are standard. (i.e. Sonarr, Radarr, Plex/Jellyfin, request apps, etc.)

If you like, I can also pull up a list specific to Philippine / Southeast Asia usage so it's more local-relevant. Do you prefer that?

### List 10 similar apps like Notion but self-hosted and dockerized and have a good iOS app.

Here are some self-hosted / Docker-friendly Notion-like apps that have good iOS app support. Some trade-offs (e.g. features, maturity) are included. I didn't find 10 perfect ones, but these are among the best options as of 2025.

## ✓ Self-Hosted + Docker + iOS Apps (or good mobile support)

#	App	iOS App?	Self-host / Docker Support	Notes / Trade-offs
1. Standard Notes	✓ Has official iOS app.	✓ Full self-hostable backend with Docker; mobile + desktop clients point to your own server.	Simpler than Notion (fewer "blocks / layouts"), more focused on notes + encryption; plugin ecosystem helps.	
2. Joplin	✓ iOS app exists.	✓ Self hosted options (Joplin Server or via sync targets like WebDAV, Nextcloud etc.). Docker setups exist.	More "note / notebook" oriented; less of Notion's database-block paradigm but strong markdown + attachments etc.	
3. AppFlowy	✓ Has iOS app.	✓ Supports self-hosting (via Supabase or AppFlowy Cloud self-hosted) with Docker etc.	Fairly new; some features still developing; syncing and collaborative editing may be less mature than Notion.	
4. Logseq	✓ Yes, iOS app.	Partially self-hostable: works "local first" on devices; for sharing / syncing, one needs setups	Sync setups require more manual configuration; UI less polished in some areas; not	

	(Git, Syncthing, NAS etc). There is some Docker involvement for supporting the web/client side.	all block-database features as in Notion.	
--	---	---	--

## ! Other potential ones / borderline cases

These are not fully matching all three (self-host + Docker + strong iOS app), but worth considering:

- Trilium Notes — very powerful for hierarchical knowledge bases; good desktop & web; mobile is less strong, and official iOS app is not as polished or may be missing.
- BookStack, Wiki.js, Outline — good for docs/wiki, decent web/mobile view, but official native iOS apps are either missing or limited (often just web-views, not fully native).
- Standard Notes is probably the strongest match overall for privacy + self-host + mobile.

If you want, I can try to find 6–10 that exactly match all your criteria (self-hosted, Docker, good iOS native app, and as feature-rich as possible) so you can pick the best. Do you want me to list those with ranking / comparison?

Explain :

```
curl https://raw.githubusercontent.com/standardnotes/server/main/.env.sample > .env
```

Sure, Rigel! Here's a breakdown of what this command does:

```
curl https://raw.githubusercontent.com/standardnotes/server/main/.env.sample > .env
```

### 🧠 What It Means

- `curl` : A command-line tool used to fetch data from a URL.
  - `https://raw.githubusercontent.com/standardnotes/server/main/.env.sample` : This is the direct URL to a sample environment configuration file ( `.env.sample` ) from the Standard

Notes server repository on GitHub.

- `> .env`: This redirects the output of the `curl` command into a new file named `.env`.

### What It Does

- Downloads the `.env.sample` file from the Standard Notes GitHub repo.
  - Saves it locally as `.env`, which is typically used to configure environment variables for a local development or deployment setup.

### Why It's Useful

- The `.env.sample` file contains placeholder values or examples of required environment variables.
  - Renaming it to `.env` allows your application (like a Node.js or Docker-based server) to read actual configuration values from it.
  - You can then edit `.env` to include your own secrets, API keys, database URLs, etc.

### Example Use Case

If you're setting up the Standard Notes server locally, this command gives you a starting point for configuration. You'd follow it up by editing `.env` to match your own setup:

## Example edits inside `.env`

```
DATABASE_URL=postgres://user:password@localhost:5432/standardnotes
NODE_ENV=development
```

Would you like help customizing that `.env` file for a local Docker or cloud deployment? I can walk you through it step-by-step.




**Explain :**

**`docker compose pull docker compose up -d`**

**Is `docker compose pull` need to run one time per project. What happens post `pull` command and `up` command**

Great question, Rigel! Let's break this down clearly and precisely, with a systems mindset:

## Command Breakdown

1. `docker compose pull`
  - Purpose: Downloads the latest versions of all container images defined in your `docker-compose.yml` file.
    - When to run:
      -  Before first-time setup (initial deployment).
      -  When updating to newer image versions (e.g., after upstream changes).
      -  Not required every time unless you want to refresh or update the images.
1. `docker compose up -d`
  - Purpose: Starts all services defined in `docker-compose.yml` in detached mode (`d`), meaning they run in the background.
    - What it does:
      - Checks if the required images are available locally.
      - If not, it will pull them automatically (unless you've already run `pull`).
  - Creates and starts containers based on those images.
    - Sets up networks, volumes, and dependencies as defined.

---

## Typical Workflow

# Step 1: Get the latest images (optional if already pulled)

```
docker compose pull
```

# Step 2: Start the services

```
docker compose up -d
```

## What Happens After Each Command

Command	Action	Result
<code>pull</code>	Fetches container images from remote registry	Local cache of latest images
<code>up -d</code>	Builds (if needed), creates, and runs containers	Services are live in background

---

## Protocol Tip for Legacy Systems

If you're designing a protocol for deployment or handoff:

- Include `docker compose pull` in your update checklist, not in every run.
    - Use `docker compose up -d` as your standard launch command.
    - For version pinning, specify image tags in `docker-compose.yml` to avoid unintended updates.
- 

Would you like a visual flow map or checklist block for Docker deployment protocols? I can tailor it to your archive or onboarding guides. Ingat, Rigel!

### **Will pull overwrites my db data if I'm using the db previously? How to prevent my data from being overwritten?**

Excellent safeguard question, Rigel — and crucial for any protocol involving persistent services. Here's the full breakdown:

---

 Does `docker compose pull` Overwrite Your Database?

No, `docker compose pull` does not touch your data.

- It only downloads updated container images from the registry.
    - It does not modify volumes, bind mounts, or any persistent storage.
    - Your database data is safe as long as it's stored in a volume or mounted directory outside the container.
- 

 What Could Overwrite Your Data?

The real risk comes from:

- Container recreation ( `docker compose up --force-recreate` )
    - Volume recreation (if volumes are not named or are removed)
    - Bind mount misconfiguration (e.g., mounting a blank folder over your data)
    - Database initialization scripts that run on container startup and overwrite existing data
- 

 How to Prevent Data Loss

Here's your protocol-grade checklist:

Use Named Volumes

services:

db:

image: postgres

volumes:

- db\_data:/var/lib/postgresql/data

volumes:

db\_data:

- Named volumes ( `db_data` ) persist across container rebuilds.
  - You can inspect them with `docker volume ls` and `docker volume inspect db_data` .

#### ✓ Avoid `--force-recreate` Unless Necessary

- This flag recreates containers even if nothing changed.
  - Use it only when you're sure your volumes are safe.

#### ✓ Backup Before Pulling or Upgrading

- Use `pg_dump` , `mysqldump` , or volume snapshot tools.
  - Example: `docker exec db_container pg_dumpall -U postgres > backup.sql`

#### ✓ Audit Init Scripts

- Some images auto-run setup scripts if they detect an empty data directory.
  - If your volume is misconfigured or missing, this can trigger a destructive re-init.

---

 Protocol Block: Safe Upgrade Flow

## Step 1: Backup your data

```
docker exec db_container pg_dumpall -U postgres > backup.sql
```

## Step 2: Pull updated images

```
docker compose pull
```

## Step 3: Start services safely

```
docker compose up -d
```

---

Would you like a visual flow map or decision matrix for safe container upgrades and volume handling? It could slot beautifully into your Mission Manual's Docker section. Ingat, Rigel!