AFRIDYNE SYSTEMS · TECHNICAL GUIDE

Docker for Beginners: Containers, Compose and Windows Apps

Run applications in containers, from first install to drop-in kits, and find out whether Docker can run Windows programs.

Updated November 2026 · Linux (Arch, Debian, Ubuntu) · Docker Engine and Compose v2 · about 45 minutes

Docker lets you run an application in a sealed box called a **container**, with everything it needs already packed inside. You do not install the application onto your computer in the usual way, and when you no longer want it you delete the box and nothing is left behind. This guide takes you from nothing to running real services, explains the “drop-in” kits you will see on GitHub, and answers a question we are often asked: *can Docker run Windows programs?*

> **What Docker is not:** it is not a virtual machine, and it does not make an insecure application safe. A container shares your computer’s kernel, so it starts in seconds and uses little memory, but it is a convenience and an isolation layer, not a security guarantee. Section 6 covers the habits that keep it safe.

What Docker is, in six words

Almost everything in Docker comes down to these terms. Come back to this table whenever a word stops making sense.

Word	What it means
image	A read-only package: the application plus everything it needs. Think of a recipe with all the ingredients included. You download it, you do not change it.
container	A running copy of an image. Think of the meal cooked from the recipe. You can start, stop and delete containers freely.
volume	A folder that lives outside the container, so your data survives when the container is deleted or replaced. Without one, the data goes with the container.
port	A door from the container to your computer, written `host:container`. `8080:80` means “what the container serves on 80, show on my port 8080”.
Compose	A small text file (`compose.yaml`) that describes one or more containers, so you start the whole set with one command instead of typing a long one.
registry	Where images are downloaded from. The default is Docker Hub. Anyone can publish there, so check who made an image before you run it.

1. Install

On **Arch Linux**:

terminal

```
sudo pacman -S docker docker-compose
sudo systemctl enable --now docker.service
```

On **Debian or Ubuntu**, use Docker’s own repository rather than the older version in the distribution’s archive. The current steps are on docs.docker.com/engine/install. They end with installing `docker-ce` and `docker-compose-plugin`.

> **Using Linux Mint?** Docker’s steps ask for your release name, and they want the **Ubuntu** one that Mint is built on, not Mint’s own name (such as *wilma*). Mint 22 is based on `noble` and Mint 21 on `jammy`. To see yours, run `grep UBUNTU_CODENAME /etc/os-release`.

Now check that it works. This downloads a tiny test image and runs it:

terminal

```
sudo docker run --rm hello-world
```

If you see “Hello from Docker!”, the engine is running. To avoid typing `sudo` every time, add yourself to the `docker` group, then log out and back in:

terminal

```
sudo usermod -aG docker $USER
```

> **The docker group is the same as root.** Anyone in it can start a container that reads or changes any file on the machine. Add only people you would give the administrator password to. On a shared or business machine, keep using `sudo docker` instead.

2. Your first container

Run the nginx web server and open it in a browser:

terminal

```
docker run -d --name web -p 8080:80 nginx
```

Then visit `http://localhost:8080`. Taking the command apart: `-d` runs it in the background, `--name web` gives it a name you can refer to, `-p 8080:80` opens the port door, and `nginx` is the image. If you do not have the image yet, Docker fetches it first.

The commands you will use daily:

terminal

```
docker ps                 # what is running
docker logs -f web        # watch its log live (Ctrl+C to leave)
docker exec -it web sh    # open a shell inside it
docker stop web           # stop it
docker rm web             # delete the container (the image stays)
```

> **Try this, it explains volumes:** start a container, create a file inside it with `docker exec`, then `docker rm -f` the container and start a new one. The file is gone. Containers are disposable by design, which is why anything you want to keep must be in a volume (next section).

3. Compose: one file instead of a long command

Long `docker run` commands are hard to remember and easy to get wrong. Compose puts the same information in a file you can read, edit and keep. Here is a real, useful example: **Uptime Kuma**, a friendly dashboard that tells you when a website or server goes down. Make a folder, then create a file called `compose.yaml` in it:

compose.yaml

```
services:
  uptime-kuma:
    image: louislam/uptime-kuma:1
    container_name: uptime-kuma
    restart: unless-stopped
    ports:
      - "127.0.0.1:3001:3001"
    volumes:
      - ./data:/app/data
```

Reading it line by line: `image` is what to run, `restart: unless-stopped` brings it back after a reboot, `ports` opens the door (only on this computer, because of the `127.0.0.1`), and `volumes` keeps its data in a `data` folder next to the file. Start it from that folder:

terminal

```
docker compose up -d
```

Open `http://localhost:3001` and create the admin account. To let other computers on your network reach it, remove the `127.0.0.1:` part, but read section 6 first. The image tag (`:1`) is the major version; check the project’s page for the current one before relying on this for anything important.

Everyday Compose commands, always run from the folder holding the file:

terminal

```
docker compose ps          # status
docker compose logs -f      # live logs
docker compose pull         # fetch newer images
docker compose up -d        # apply changes / update
docker compose down         # stop and remove the containers (data in ./data is kept)
```

Never add `-v` to `docker compose down` casually. It deletes the named volumes as well, which means your data. A plain `down` keeps it.

4. Drop-in containers

“Drop-in” is not an official Docker term. It is what people call a ready-made folder you drop next to your project or onto a server, fill in a few settings, and start. Nothing is built from scratch. A good example is drop-in-docker-php, a starter kit for PHP developers. Its README describes the whole routine, and it is the same routine you will meet in most kits:

1. **Add the folder to your project.** Copy it in, or add it as a Git submodule so you can pull updates.
2. **Copy the template settings.** Files named like `tmp.env` become `.env`, and each `tmp.*.env` file in the `env` folder loses its `tmp.` prefix.
3. **Edit the settings.** Ports, the database password, and where your source code lives, written relative to the compose file.
4. **Start it.** The kit wraps Compose in a `Makefile`, so `make up` starts everything, `make down` stops it and `make shell` opens a shell in the app container.

What you get is a whole working stack with one command. In this kit that is a PHP application, a web server in front of it, a MySQL database, Adminer to look at the database, and MailHog to catch outgoing test email. Installing all of that by hand on your own computer would take an afternoon, and it would stay behind when the project ended. In a container it comes and goes cleanly, and a colleague gets exactly the same setup by cloning the project.

Before you run any drop-in kit, spend five minutes on this checklist:

- **Read the README and the compose file.** They are short, and they show every port opened and every folder shared.
- **Check where the images come from.** Official or well-known publishers only for anything touching real data.
- **Change every default password** in the `.env` files before the first start.
- **Keep `.env` out of Git.** It holds your passwords. Share the `tmp.*` templates, not the real files.

More services that run well as containers in a home or small office, each one a compose file away:

Service	What it does for you
Uptime Kuma	Tells you when a website, server or router stops answering. Section 3 above.
Pi-hole	Network-wide advert and tracker blocking. See our Pi-hole guide; note it needs port 53, which can clash with the system resolver.
Nextcloud	Your own file sync and sharing, in the style of Dropbox, kept on your hardware.
Jellyfin	A media server for your own films and music, to any device at home.
Vaultwarden	A small self-hosted password manager compatible with Bitwarden apps.

5. Can Docker run Windows applications?

This is the question we are asked most, and the answer is **not directly**. A container borrows the kernel of the computer it runs on. On a Linux machine that kernel is Linux, so ordinary Docker containers can only run Linux software. A Windows program needs Windows. There are four honest routes, from simplest to heaviest:

Route	How it works	Good for	Watch out for
Wine (no Docker)	Translates Windows calls on the fly. Installed like any Linux program, no Windows needed.	Many everyday programs, some games. No Windows licence.	Not every program works. Check the Wine AppDB first.

Route	How it works	Good for	Watch out for
Windows in a container	A complete Windows runs inside a container using Linux's KVM virtualization, with a screen in your browser or over RDP.	Programs that must have real Windows.	Needs KVM, ideally 8 GB of RAM or more (the default is 4 GB), 32 GB+ of free disk and your own Windows licence.
WinApps or WinBoat	Runs that Windows in the background and shows each program as its own window on your Linux desktop.	Daily use of a few Windows programs, such as Office, without a dual boot.	Fiddly setup, and slow on older hardware. WinBoat is a young project, so expect rough edges.
Windows containers	Docker's native Windows mode, on a Windows host only.	Server software on Windows machines.	Not built for desktop programs with a window and a mouse.

If you are on **Windows 11** already, you can simply run your Windows programs. Docker Desktop on Windows is for running Linux containers, using WSL 2, so you do not need any of this section. To work out which of your programs need any of it, see Windows Programs on Linux.

Windows in a container, step by step

The most popular option is the open-source dockur/windows project. It downloads Windows, installs it automatically and gives you a desktop in your browser. First, check that your Linux machine can do hardware virtualization. Turn it on in the BIOS or UEFI if it is off (look for Intel VT-x or AMD-V, or "SVM"):

terminal

```
ls -l /dev/kvm          # the file must exist
lscpu | grep -i virtualization
```

On Debian or Ubuntu you can also install `cpu-checker` and run `kvm-ok`. If `/dev/kvm` is missing, stop here: this route will not work on that machine. Docker Desktop on Linux or macOS does not provide it either. Then make a folder and a `compose.yaml`:

compose.yaml

```
services:
  windows:
    image: dockurr/windows
    container_name: windows
    environment:
      VERSION: "11"
      RAM_SIZE: "8G"
      CPU_CORES: "4"
      DISK_SIZE: "64G"
    devices:
      - /dev/kvm
      - /dev/net/tun
    cap_add:
      - NET_ADMIN
    ports:
      - "127.0.0.1:8006:8006"
      - "127.0.0.1:3389:3389/tcp"
      - "127.0.0.1:3389:3389/udp"
    volumes:
      - ./windows:/storage
      - ./shared:/shared
    restart: unless-stopped
    stop_grace_period: 2m
```

Run `docker compose up -d`, then open `http://localhost:8006` to watch Windows install itself. It downloads several gigabytes the first time. When it reaches the desktop, install your program as you would on any PC. Anything in your `shared` folder appears inside Windows as a `Shared` folder, which is the easy way to move files in and out.

For daily use, connect with a Remote Desktop (RDP) client instead of the browser. It is faster and supports the clipboard. On Linux, `xfreerdp` or Remmina will do. The project's default login, if you did not set your own, is user `Docker` with password `admin`: change it with the `USERNAME` and `PASSWORD` settings before you use this for anything real.

Licensing. The project downloads Windows for you but does not give you a licence. You are responsible for holding a valid Windows licence and following Microsoft's terms, especially in a business. If a program is for work, check this before you build anything around it.

Be realistic about speed. This is a full Windows running on your computer, so expect it to use several gigabytes of memory all the time, and to be slow for heavy graphics or full-screen games. For one or two office programs it is perfectly usable.

What about Photoshop?

Photoshop is the hardest case, so be cautious. It is a heavy program, and Windows in a container is the weakest setup for heavy programs. Three things matter:

- **No graphics card.** By default the container cannot use your graphics card, so Windows draws everything in software. Photoshop uses the graphics card for many of its tools, so some are switched off and others run slowly.
- **Memory.** Adobe recommends 16 GB of RAM or more for Photoshop. The container gets only the share you give it with `RAM_SIZE`, and your own Linux desktop still needs its share.
- **Disk.** Photoshop uses a large "scratch" area on disk while you work. Raise `DISK_SIZE` well above the 64 GB default, and keep the Windows folder on a fast drive (an SSD).

It is fine for opening a file and making light edits, and frustrating for a full working day. For regular professional use the realistic choices are a dual boot with Windows, a full virtual machine with the graphics card passed through (an advanced setup that needs suitable hardware), or switching to a Linux-native tool such as GIMP or Krita. Photoshop is also a paid subscription that needs its own Adobe sign-in, on top of the Windows licence. Adobe does not support it under Wine, and recent versions are unreliable there.

6. Keep it safe

Docker bypasses your firewall for published ports. This surprises almost everyone. When a container publishes a port, Docker adds its own network rules, and they can take effect ahead of a simple firewall such as `ufw`. A rule that says "block 3001" may not stop anyone reaching it. The reliable fix is to publish only to the local machine, as every example in this guide does:

terminal

```
ports:
  - "127.0.0.1:3001:3001"     # only this computer
  # - "3001:3001"             # every computer that can reach this one
```

If a service needs to be reachable from elsewhere, put it behind a reverse proxy (such as NGINX) or a VPN. Our WireGuard guide is a good way to reach home services securely from outside.

- **Back up your volumes.** Containers are disposable, your `data` folders are not. Include them in your backup plan.
- **Update on purpose.** Containers do not update themselves. Run `docker compose pull` followed by `docker compose up -d` now and then, and read release notes for anything important.
- **Be careful with the Docker socket.** If a guide tells you to mount `/var/run/docker.sock` into a container, that container can control everything on the machine. Do it only for tools you trust.
- **Pin versions for anything important.** `image: nginx:1.27` is predictable, `image: nginx` changes under you.
- **Tidy up now and then.** Old images and stopped containers use real disk space (see the cheat sheet).

When things go wrong

What you see	Likely cause	What to do
port is already allocated	Something else on this computer already uses that port.	Change the left-hand number: `8081:80`. Check what has it with `sudo ss -tlnp`.
permission denied … docker.sock	Your user is not in the `docker` group.	Use `sudo`, or add yourself to the group (see the warning in section 1) and log in again.
Container starts, then exits	A setting is wrong or missing, a password is empty, or a file is not readable.	`docker compose logs` almost always says why. Read the last 20 lines.

What you see	Likely cause	What to do
Works here, not from another computer	The port is published to 127.0.0.1 only, or a firewall or router blocks it.	Decide whether you really want it exposed. If yes, see section 6.
Settings and data vanished	The container was removed and the data was never in a volume.	Add a `volumes:` entry. Restore from backup if you have one.
no space left on device	Old images and stopped containers have piled up.	`docker system df` to see, `docker system prune` to clean.
/dev/kvm: no such file	Hardware virtualization is off or not available (Windows in a container).	Enable VT-x or AMD-V in the BIOS. In a virtual machine, switch on nested virtualization.

If you are stuck, the best three pieces of information to have ready are the output of `docker compose ps`, the last lines of `docker compose logs`, and your `compose.yaml` with the passwords removed.