NEWS · AUGUST 21, 2026

Install Docker Desktop and Run Your First Container (1/5)

By Dr. Holger Flick

Picture the moment. You're deep in a Delphi client/server feature and you finally need a real backend to develop against — a Postgres database, say, or a Redis cache, or a message broker. On your own Windows dev machine. Right now.

So you go find the installer. You run it. It registers a Windows service, edits your `PATH`, drops config files in three places, opens a port, and asks you to pick a superuser password you'll forget by Thursday. It works — until you need version 15 for one project and 17 for another, and now two services fight over the same port. And when the project ends, uninstalling leaves orphaned services, stray directories, and a registry you don't dare touch. Every seasoned desktop developer knows this particular dread.

There is a better way, and it's the whole reason this series exists. With one command you can have a real, running Postgres server. With one more, it's gone — cleanly, completely, as if it were never there. No services, no `PATH` surgery, no uninstall mess. That's the promise of **containers**, and this post gets you set up to make good on it.

If you just finished the Networking for Delphi Developers series, this is the sequel I promised you at its doorway.

That series left you fluent in IP addresses, ports, DNS, TCP, HTTP, and web services. You'll use every bit of it here — but first, the tooling and three ideas that make the rest of the series easy.

> **This series: Docker for Delphi Developers**

1. **Install Docker Desktop and run your first container — this post** — what a container, image, and registry actually are 2. Run Postgres in a container, talk to it from Delphi — a real database in one command, FireDAC creates and queries a table 3. Images, tags, layers, and registries — where images come from and how they're built up 4. Docker Compose: a database and a backend in one file — multi-service apps and service-name networking 5. Isolation architecture: web client ’ backend ’ database — private networks and who's allowed to reach the data

Three ideas before a single command

Before we install anything, let me hand you the mental model. Docker has exactly three concepts you must hold in your head, and every one of them maps cleanly onto something you already know as a Delphi developer. Get these three and the commands become obvious rather than magic.

The three are **image**, **container**, and **registry**. We'll take them one at a time.

An image is a class

An image is a read-only template: a packaged, frozen recipe containing an application and everything it needs to run — the program, its libraries, its default configuration, right down to a minimal slice of a Linux filesystem. It doesn't *do* anything on its own. It just sits there, a blueprint waiting to be brought to life.

If that sounds familiar, it should: **an image is a class.** You declare it once, it holds no live state, and it exists to be instantiated.

A container is an object

A container is what you get when you *run* an image: a live, running instance with its own state, its own memory, its own isolated view of the filesystem and network. You can start many containers from one image, exactly the way you construct many objects from one class. Each is independent; you can throw one away without touching the image or the others.

So the analogy every Delphi developer already owns is the whole thing in one line.

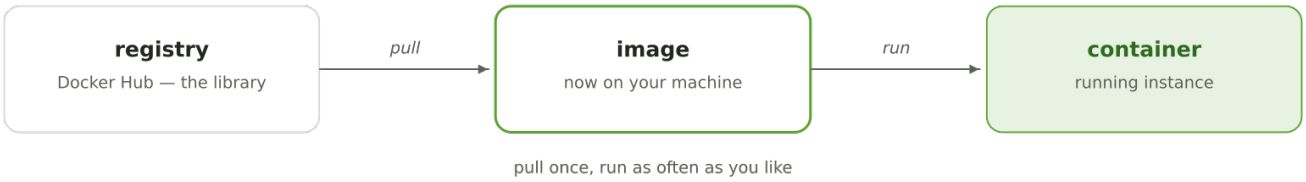
The one analogy that unlocks Docker: image is to container as class is to object

Read it left to right: one image on the left, several live containers in the middle, and the Delphi vocabulary on the right. `docker run` is the constructor. Once that clicks, the rest of Docker is just method names.

A registry is where images come from

You didn't write the Postgres image, and you never will. Somebody published it, and you *pull* it. The place you pull from is a registry — a server that stores and serves images. The default public one is Docker Hub, a vast library of ready-made images: Postgres, Redis, nginx, and thousands more, maintained and versioned so you don't have to be.

Think of a registry as a package repository for whole applications rather than source libraries — closer to Delphi's GetIt or a NuGet feed in spirit, but what it ships is a complete, runnable server. Here's how the three ideas fit together.



Pull an image from a registry, then run it into a container — the whole loop

The image is pulled from the registry to your machine once; from there you run it into containers whenever you need them. Keep this loop in mind — it's literally what the next command does.

The one contrast worth drawing: a container is not a VM

You may be wondering how this differs from a virtual machine — the technology that runs a whole guest operating system, kernel and all, on top of your own. The honest one-line answer: a container is a **lightweight, isolated box that shares the host's kernel** instead of booting its own, so it starts in a fraction of a second and carries only the application, not a whole OS. A VM virtualizes an entire computer; a container isolates a single application. That's the only VM comparison this series needs — we're not here to teach virtualization, just to place containers next to a thing you already know.

Getting Docker onto Windows

The tool we install is Docker Desktop — the official application that bundles the Docker engine, the command-line tools, and a management dashboard into one Windows installer. On Windows it runs its Linux containers using WSL 2, the Windows Subsystem for Linux version 2 — a genuine, Microsoft-supplied Linux kernel that Windows runs for you in the background. (Docker Desktop can alternatively use the Hyper-V backend, but WSL 2 is the recommended default and what we'll set up.)

> **Check the requirements first**
> Per Docker's Windows install documentation, the WSL 2 backend needs 64-bit **Windows 10 22H2** (build 19045) or **Windows 11 23H2** (build 22631) or higher — the Enterprise, Pro, or Education editions — a 64-bit processor with SLAT, at least **8 GB of RAM**, hardware virtualization enabled in the BIOS/UEFI, and

> **WSL version 2.1.5 or later**. Confirm the current requirements at the link before you begin, since they move over time.

The setup is a short, ordered procedure. Follow it top to bottom.

1. Enable WSL 2

Open PowerShell **as Administrator** (right-click ’ Run as administrator) and run the single install command, then restart when prompted. Per Microsoft's WSL install guide, this one command enables the required Windows features and installs a default Linux distribution:

```
wsl --install
```

New distributions installed this way are set to WSL 2 by default. If WSL was already present from an older setup, make WSL 2 the default explicitly:

```
wsl --set-default-version 2
```

2. Install Docker Desktop

Download the installer from the official Docker Desktop page, run `Docker Desktop Installer.exe`, and on the configuration page make sure the **WSL 2 backend** option is selected. Complete the wizard and accept the subscription agreement.

3. Launch it and confirm the engine is up

Start Docker Desktop from the Start menu and wait for the whale icon in the system tray to stop animating — that means the engine is running. Then open a terminal (PowerShell or Windows Terminal) and verify the CLI can talk to it:

```
docker version
```

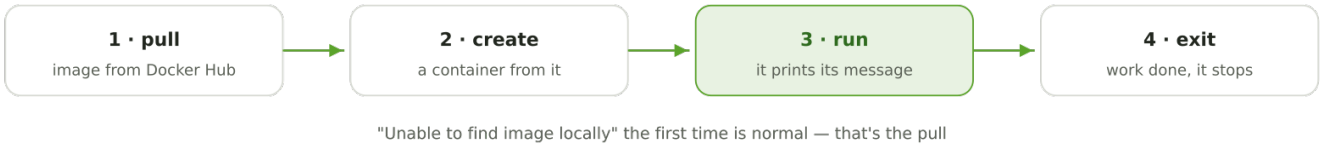
You should see both a **Client** and a **Server** section. If the Server section is present, the engine is live and you're ready.

Your first container: `hello-world`

Now the moment the whole install was for. Run Docker's tiny official test image, which exists for exactly this purpose:

```
docker run hello-world
```

You'll see a short message that begins with "Hello from Docker!" — but the interesting part is what Docker did to produce it. It's the three-idea loop, live, in order.



What one docker run does: pull from the registry, create, run, exit

Reading the flow: because you'd never run `hello-world` before, Docker couldn't find the image locally, so it **pulled** it from Docker Hub (idea three). It then **created** a container from that image (idea two, the object from the class) and **ran** it. The container's only job was to print its greeting, so once printed, it **exited**. That "Unable to find image locally" line you see the first time isn't an error — it's the pull happening, and it won't repeat, because the image now lives on your machine.

A handful of commands to build confidence

The container exited, but it didn't vanish — and neither did the image. A few short commands let you see and tidy what `docker run` left behind. Each does one clear thing.

The table names each command and exactly what it shows or does.

Command	What it does
`docker ps`	Lists **running** containers only. Right after `hello-world`, this is empty — it already exited.
`docker ps -a`	Lists **all** containers, including stopped ones. Here you'll find your `hello-world` container sitting in `Exited` state.
`docker images`	Lists the images on your machine — the `hello-world` image you just pulled will be one of them.
`docker rm <name-or-id>`	Removes a stopped container by its name or ID (copy either from `docker ps -a`). This is the "clean when done" half of the promise.

Try them in sequence: run `docker ps` (empty), then `docker ps -a` (there's your exited container, with an auto-assigned name like `sleepy_bell`), then `docker images` (there's the pulled image), then `docker rm` with that container's name to remove it. That last command is the point of the whole exercise — the container disposes cleanly, and after Part 2 you'll do the same to a whole Postgres server.

> **Two flags worth knowing early**
> You don't need these yet, but they're the two you'll reach for constantly. Adding `--rm` to `docker run` tells Docker to auto-delete the container the moment it exits — no manual `docker rm` needed. And `-d` (detached) runs a container in the background instead of tying up your terminal, which is exactly how you'll start a long-lived server like Postgres in the next post.

Honest talk: licensing and free alternatives

One thing I insist on before you build anything on a tool: know what it costs and what else exists. Docker Desktop is free for a lot of people — and for some organizations it isn't — and there are excellent free, open-source alternatives. Here's the fair, complete picture.

Per Docker's subscription terms, **Docker Desktop is free** for personal use, education, non-commercial open-source projects, and small businesses — defined as **fewer than 250 employees* and *less than \$10 million in annual revenue**. Larger organizations and government entities need a paid subscription (Pro, Team, or Business). That's a genuinely generous free tier that covers most individual developers and small shops outright — and if you're at a larger company, the paid plans fund the maintenance of the tooling and the vast public image library you'll lean on all series long.

It's also worth being precise about *what* is licensed. The paid requirement is for **Docker Desktop, the convenient bundled application**. The underlying Docker Engine and the `docker` CLI are open source (Apache 2.0), and the container format itself is an open standard governed by the Open Container Initiative (OCI) — which is precisely why the alternatives below can run the very same images.

> **Free and open-source alternatives**
> If Docker Desktop's license doesn't fit your situation, two mature open-source tools run the same OCI containers and images — the exact ones from Docker Hub — so everything in this series still applies. Rancher Desktop (open source, by SUSE) gives you container and Kubernetes tooling on Windows, macOS, and Linux, and can even use the Docker/Moby engine so the familiar `docker` command works unchanged. Podman Desktop (open source, a CNCF project) is a vendor-neutral GUI for building and running OCI containers, also cross-platform. Choose by fit: Docker Desktop for the polished all-in-one default and commercial support, Rancher or Podman Desktop when you want a fully open-source stack. None is a consolation prize — they're all first-rate tools built on the same open standard.

Takeaways

You installed the tooling and, more importantly, you now own the three ideas the entire rest of this series stands on. An **image** is a read-only template — a class. A **container** is a running instance of it — an object, born from `docker run`. A **registry** like Docker Hub is where images come from — you pull once and run as often as you like. And a container isn't a virtual machine: it's a lightweight box that shares your host's kernel, so it's fast to start and clean to remove.

> A container is an object; an image is its class; `docker run` is the constructor. Once that clicks, Docker stops being a mystery and starts being a tool.

That "clean to remove" property is the promise from the very top of this post, and next time we cash it in for real. In Part 2: a real Postgres database in one command, running on your Windows machine with nothing installed natively — and a Delphi app, using FireDAC over the modern RTL, creating a table, inserting rows, and reading them back. See you there.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —