

How to Use the Postgres Docker Official Image



Tyler Charboneau

Postgres is [one of the top relational, multi-model databases](#) currently available. It's designed to power database applications — which either serve important data directly to end users or through another application via APIs. Your typical website might fit that first example well, while a finance app (like PayPal) typically uses APIs to process `GET` or `POST` database requests.

Postgres' object-relational structure and concurrency [are advantages over alternatives like MySQL](#). And while no database technology is objectively the best, Postgres shines if you value extensibility, data integrity, and open-source software. It's highly scalable and supports complex, standards-based SQL queries.

Posted




Oct 5, 2022



Post Tags

-  Docker
-  Docker community
-  Docker Desktop
-  Docker Official Images
-  Popular Topics

Categories

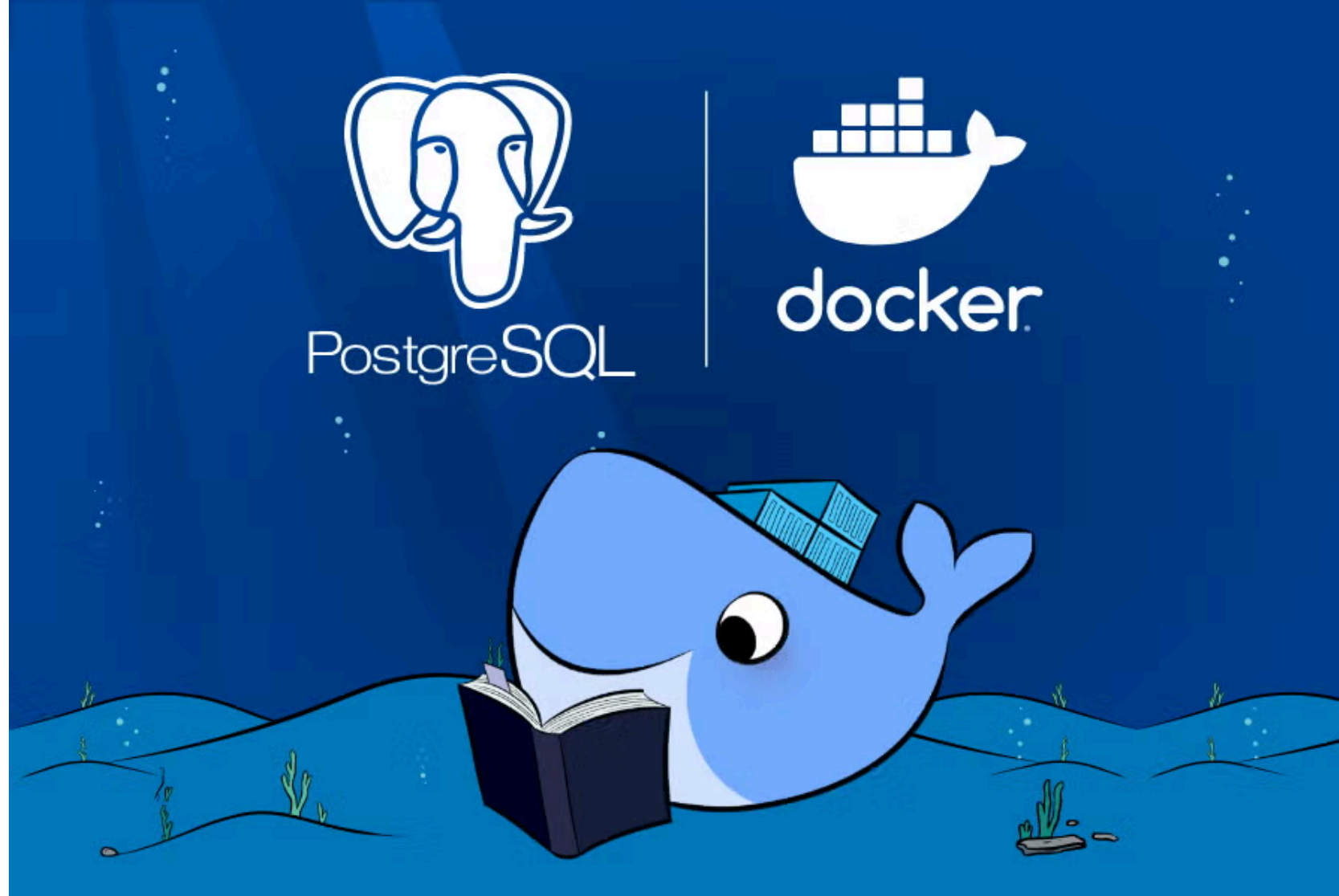
-  Community
-  Company
-  Engineering

The [Postgres Docker Official Image \(DOI\)](#) lets you create a Postgres container tailored specifically to your application. This image also handles many core setup tasks for you. We'll discuss containerization, and the Postgres DOI, and show you how to get started.

In this tutorial:

- [Why should you containerize Postgres?](#)
- [What's the Postgres Docker Official Image?](#)
 - [Can you deploy Postgres containers in production?](#)
- [How to run Postgres in Docker](#)
 - [Enter a quick pull command](#)
 - [Start a Postgres instance](#)
 - [Using Docker Compose](#)
- [Extending your Postgres image](#)
 - [1. Environment variables](#)
 - [2. Docker secrets](#)
 - [3. Initialization scripts](#)
 - [4. Database configuration](#)
- [Important caveats and data storage tips](#)
- [Jumpstart your next Postgres project today](#)

Why should you containerize Postgres?



Since your Postgres database application can run alongside your main application, containerization is often beneficial. This makes it much quicker to spin up and deploy Postgres anywhere you need it. Containerization also separates your data from your database application. Should your application fail, it's easy to launch another container while shielding your data from harm.

This is simpler than installing Postgres locally, performing additional configuration, and starting your own background processes. Such workflows take extra time, require deeper technical knowledge, and don't adapt well to changing application requirements. That's why Docker containers come in handy — they're approachable and tuned for rapid development.

What's the Postgres Docker Official Image?

Like any other Docker image, the Postgres Docker Official Image contains all source code, core dependencies, tools, and libraries your application needs. The Postgres DOI tells your database application how to behave and interact with data. Meanwhile, your Postgres container is a running instance of this standard image.

Specifically, Postgres is perfect for the following use cases:

- Connecting Docker shared volumes to your application
- Testing your storage solutions during development
- Testing your database application against newer versions of your main application or Postgres itself

The PostgreSQL Docker Community maintains this image and added it to [Docker Hub](#) due to its widespread appeal.

Can you deploy Postgres containers in production?

Yes! Though this answer comes with some caveats and depends on how many containers you want to run simultaneously.

While it's possible to use the Postgres Official Image in production, Docker Postgres containers are best suited for local development. This lets you use tools like Docker Compose to collectively manage your services. You aren't forced to juggle multiple database containers at scale, which can be challenging.

Launching production Postgres containers means using an orchestration system like Kubernetes to stay up and running. You may also need third-party components to supplement Docker's offerings. However, you can absolutely give this a try if you're comfortable with Kubernetes! Arctype's Shanika Wickramasinghe [shares one method for doing so](#).

For these reasons, you can perform prod testing with just a few containers. But, it's best to reconsider your deployment options for anything beyond that.

How to run Postgres in Docker

To begin, [download the latest Docker Desktop release](#) and install it. Docker Desktop includes the Docker CLI, Docker Compose, and supplemental development tools. Meanwhile, the Docker Dashboard (Docker Desktop's UI component) will help you manage images and containers.

Afterward, it's time to Dockerize Postgres!

Enter a quick pull command

Pulling the Postgres Docker Official Image is the fastest way to get started. In your terminal, enter

```
docker pull postgres
```

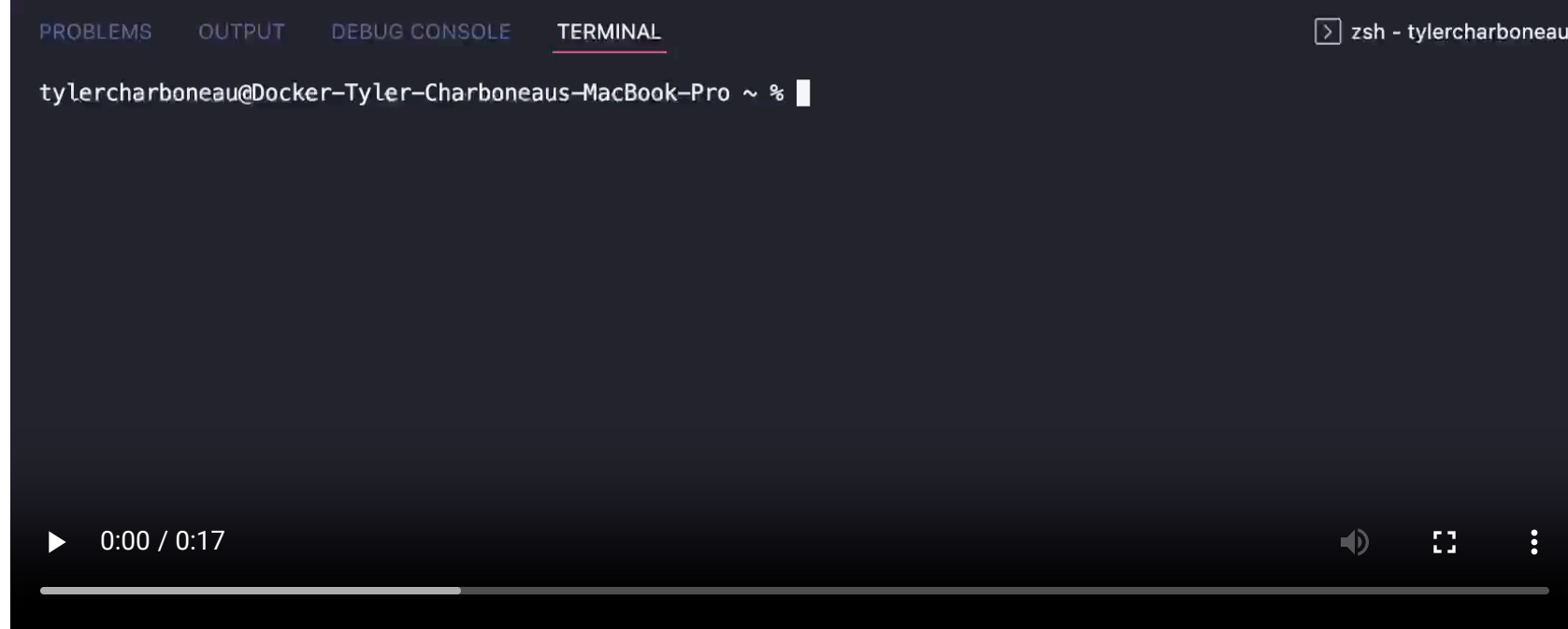
 to grab the latest Postgres version from Docker Hub.

Alternatively, you can pin your preferred version with a specific tag. Though we usually associate pinning with Dockerfiles, the concept is similar to a basic pull request.

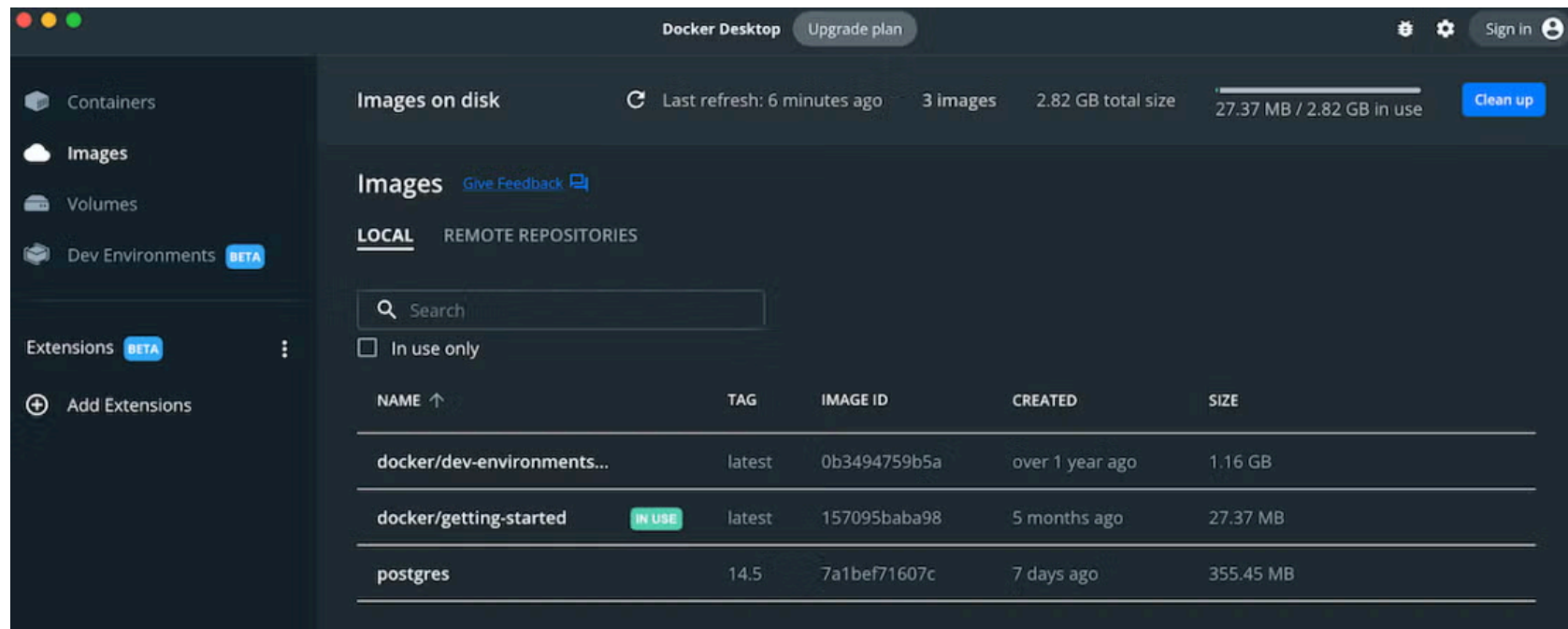
For example, you'd enter the `docker pull postgres:14.5` command if you prefer `postgres v14.5`.

Generally, we recommend using a specific version of Postgres. The `:latest` version automatically changes with each new Postgres release — and it's hard to know if those newer versions will introduce breaking changes or vulnerabilities.

Either way, Docker will download your Postgres image locally onto your machine. Here's how the process looks via the CLI:



Once the pull is finished, your terminal should notify you. You can also confirm this within Docker Desktop! From the left sidebar, click the **Images** tab and scan the list that appears in the main window. Docker Desktop will display your `postgres` image, which weighs in at 355.45 MB.



Postgres is one of the slimmest major database images on Docker Hub. But `alpine` variants are also available to further reduce your image sizes and include basic packages (perfect for simpler projects). You can learn more about Alpine’s benefits in [our recent Docker Official Image article](#).

Next up, what if you want to run your new image as a container? While many other images let you hover over them in the list and click the blue “Run” button that appears, Postgres needs a little extra attention. Being a database, it requires you to set environment variables before forming a successful connection. Let’s dive into that now.

Start a Postgres instance

Enter the following `docker run` command to start a new Postgres instance or container:

```
1 | docker run --name some-postgres -e POSTGRES_PASSWORD=mysecretpassword -d postgres
```

This creates a container named `some-postgres` and assigns important environment variables before running everything in the background. Postgres requires a password to function properly, which is why that’s included.

If you have this password already, you can spin up a Postgres container within Docker Desktop. Just click that aforementioned “Run” button beside your image, then manually enter this password within the “Optional Settings” pane before proceeding.

However, you can also use the Postgres interactive terminal, or `psql`, to query Postgres directly:

```
1 | docker run -it --rm --network some-network postgres psql -h some-postgres -U postgres
2 | psql (14.3)
3 | Type "help" for help.
4 |
5 | postgres=# SELECT 1;
6 |      ?column?
7 | -----
8 |              1
9 | (1 row)
```

Using Docker Compose

Since you're likely using multiple services, or even a database management tool, Docker Compose can help you run instances more efficiently. With a single YAML file, you can define how your services work. Here's an example for Postgres:

```
1  services:
2    db:
3      image: postgres
4      restart: always
5      environment:
6        POSTGRES_PASSWORD: example
7      volumes:
8        - pgdata:/var/lib/postgresql/data
9
10   adminer:
11     image: adminer
12     restart: always
13     ports:
14       - 8080:8080
15
16   volumes:
17     pgdata:
```

You'll see that both services are set to `restart: always`. This makes our data accessible whenever our applications are running and keeps the Adminer management service active simultaneously. When a container fails, this ensures that a new one starts right up.

Say you're running a web app that needs data immediately upon startup. Your Docker Compose file would reflect this. You'd add your `web` service and the `depends_on` parameter to specify startup and shutdown dependencies between services. Borrowing from our docs on [controlling startup and shutdown order](#), your expanded Compose file might look like this:

```
1  services:
2    web:
3      build: .
4      ports:
```



```
5     - "80:8000"
6     depends_on:
7         db:
8             condition: service_healthy
9     command: ["python", "app.py"]
10
11 db:
12     image: postgres
13     restart: always
14     environment:
15         POSTGRES_PASSWORD: example
16     healthcheck:
17         test: ["CMD-SHELL", "pg_isready"]
18         interval: 1s
19         timeout: 5s
20         retries: 10
21
22 adminer:
23     image: adminer
24     restart: always
25     ports:
26         - 8080:8080
```

To launch your Postgres database and supporting services, enter the `docker compose -f [FILE NAME] up` command.

Using either `docker run`, `psql`, or Docker Compose, you can successfully start up Postgres using the Official Image! These are reliable ways to work with “default” Postgres. However, you can configure your database application even further.

Extending your Postgres image

There are many ways to customize or configure your Postgres image. Let’s tackle four important mechanisms that can help you.

1. Environment variables

We've touched briefly on the importance of `POSTGRES_PASSWORD` to Postgres. Without specifying this, Postgres can't run effectively. But there are also other variables that influence container behavior:

- `POSTGRES_USER` – Specifies a user with superuser privileges and a database with the same name. Postgres uses the default user when this is empty.
- `POSTGRES_DB` – Specifies a name for your database or defaults to the `POSTGRES_USER` value when left blank.
- `POSTGRES_INITDB_ARGS` – Sends arguments to `postgres_initdb` and adds functionality
- `POSTGRES_INITDB_WALDIR` – Defines a specific directory for the Postgres transaction log. A transaction is an operation and usually describes a change to your database.
- `POSTGRES_HOST_AUTH_METHOD` – Controls the `auth-method` for `host` connections to `all` databases, users, and addresses
- `PGDATA` – Defines another default location or subdirectory for database files

These variables live within your plain text `.env` file. Ultimately, they determine how Postgres creates and connects databases. You can check out our [GitHub Postgres Official Image documentation](#) for more details on environment variables.

2. Docker secrets

While environment variables are useful, passing them between host and container doesn't come without risk. Docker secrets let you access and load those values from files already present in your container. This prevents your environment variables from being intercepted in transit over a port connection. You can use the following command (and iterations of it) to leverage Docker secrets with Postgres:

```
1 | docker run --name some-postgres -e POSTGRES_PASSWORD_FILE=/run/secrets/postgres-passwd -d postgres
```

Note: Docker secrets are only compatible with certain environment variables. [Reference our docs](#) to learn more.

3. Initialization scripts

Also called `init` scripts, these run any executable shell scripts or command-based `.sql` files once Postgres creates a `postgres-data` folder. This helps you perform any critical operations before your services are fully up and running. Conversely, Postgres will ignore these scripts if the `postgres-data` folder initializes.

4. Database configuration

Your Postgres database application acts as a server, and it's beneficial to control how it runs. Configuring your database not only determines how your Postgres container talks with other services, but also optimizes how Postgres runs and accesses data.

There are two ways you can handle [database configurations with Postgres](#). You can either apply these configurations locally within a dedicated file or use the command line. The CLI uses an entrypoint script to pass any Docker commands to the Postgres server daemon for processing.

Note: Available configurations differ between Postgres versions. The configuration file directory also changes slightly while using an `alpine` variant of the Postgres Docker Official Image.

Important caveats and data storage tips

While Postgres can be pretty user-friendly, it does have some quirks. Keep the following in mind while working with your Postgres container images:

- If no database exists when Postgres spins up in a container, it'll create a default database for you. While this process unfolds, that database won't accept any incoming connections.
- Working with a pre-existing database is best when using Docker Compose to start up multiple services. Otherwise, automation tools may fail while Postgres creates a default.
- Docker will throw an error if a Postgres container exceeds its 64 MB memory allotment.
- You can use either a `docker run` command or Docker Compose to allocate more memory to your Postgres containers.

Storing your data in the right place

Data accessibility helps Postgres work correctly, so you'll also want to make sure you're storing your data in the right place. This location must be visible to both Postgres and Docker to prevent pesky issues. While there's no perfect storage solution, [remember the following](#):

- Writing files to the host disk (Docker-managed) and using internal volume management is transparent and user-friendly. However, these files may be inaccessible to tools or apps outside of your containers.
- Using bind mounts to connect external data to your Postgres container can solve data accessibility issues. However, you're responsible for creating the directory and setting up permissions or security.

Lastly, if you decide to start your container via the `docker run` command, don't forget to mount the appropriate directory from the host. The `-v` flag enables this. Browse [our Docker run documentation](#) to learn more.

Jumpstart your next Postgres project today

As we've discovered, harnessing the Postgres Docker Official Image is pretty straightforward in most cases. Since many customizations are available, you only need to explore the extensibility options you're comfortable with. Postgres even supports extensions (like [PostGIS](#)) – which could add even deeper functionality.

Overall, Dockerizing Postgres locally has many advantages. Swing by Docker Hub and pull your first [Postgres Docker Official Image](#) to start experimenting. You'll find even deeper instructions for enhancing your database setup on [our Postgres GitHub page](#).

Need a springboard? Check out these Docker awesome-compose applications that leverage Postgres:

- Build a Go server with an [NGINX proxy and a Postgres database](#).
- Create a sample [Postgres and pgAdmin management setup](#).
- Build a React application with a [Rust backend and Postgres database](#).
- Create a Java application with [Spring Framework and a Postgres database](#).

[Docker](#), [Docker community](#), [Docker Desktop](#), [Docker Official Images](#), [Popular Topics](#)

Related Posts