1：

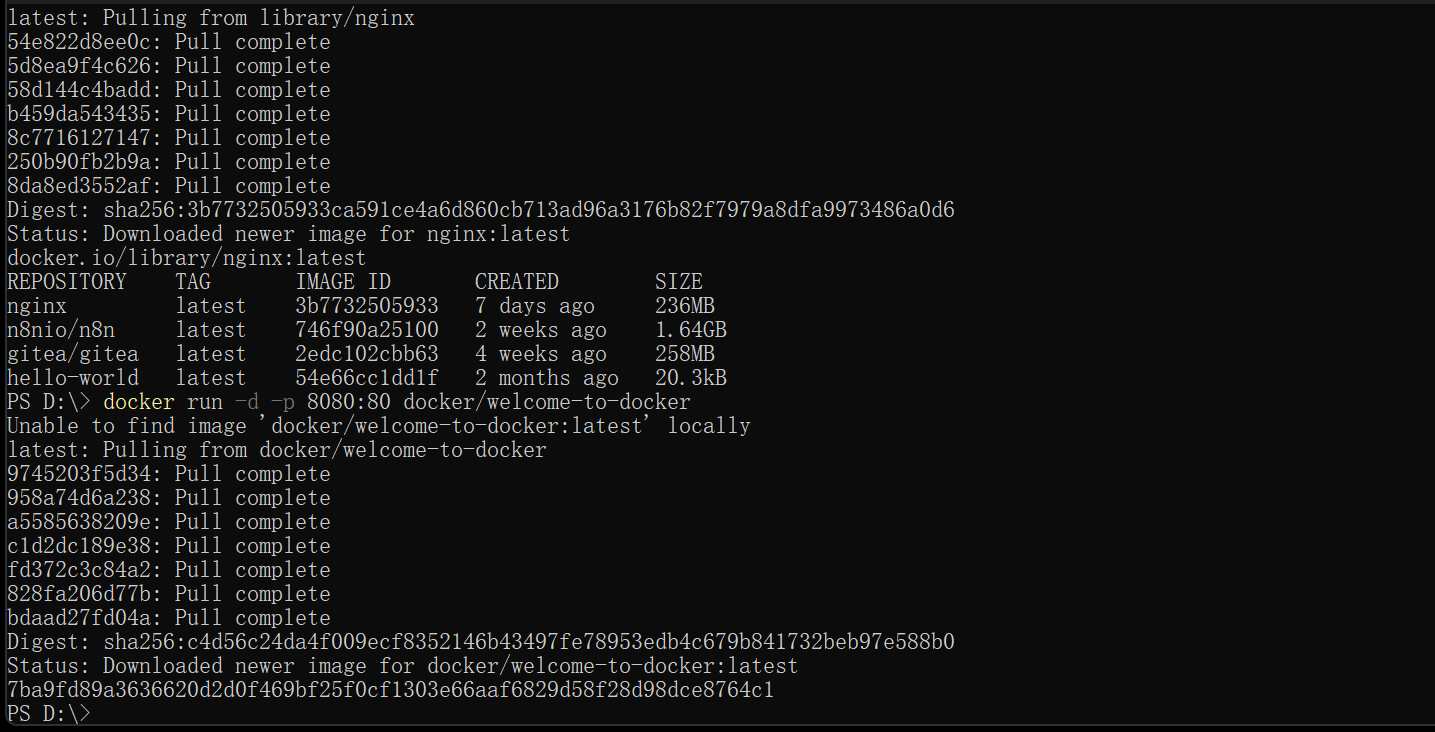
Imagine you're developing a killer web app that has three main components - a React frontend, a Python API, and a PostgreSQL database. If you wanted to work on this project, you'd have to install Node, Python, and PostgreSQL.

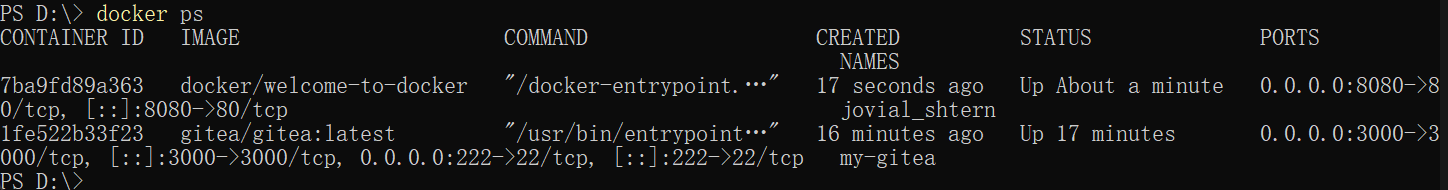
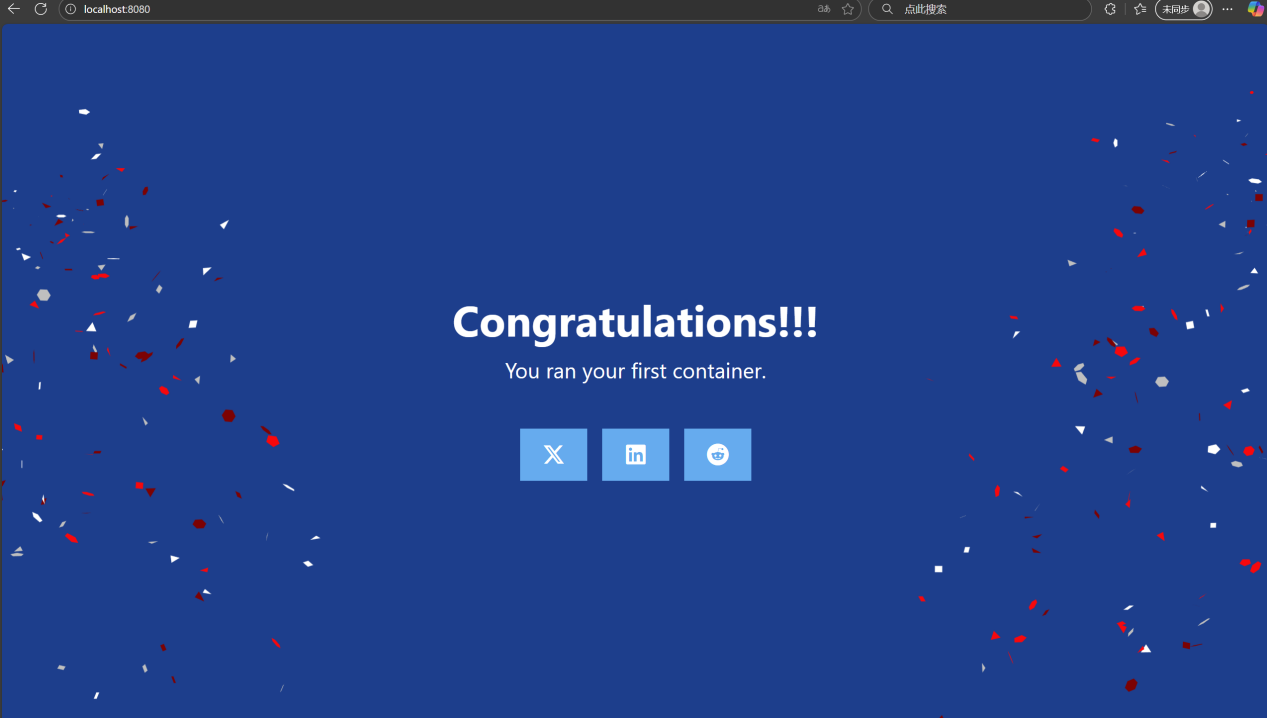
How do you make sure you have the same versions as the other developers on your team? Or your CI/CD system? Or what's used in production?

How do you ensure the version of Python (or Node or the database) your app needs isn't affected by what's already on your machine? How do you manage potential conflicts?

Enter containers!

What is a container? Simply put, containers are isolated processes for each of your app's components. Each component - the frontend React app, the Python API engine, and the database - runs in its own isolated environment, completely isolated from everything else on your machine.





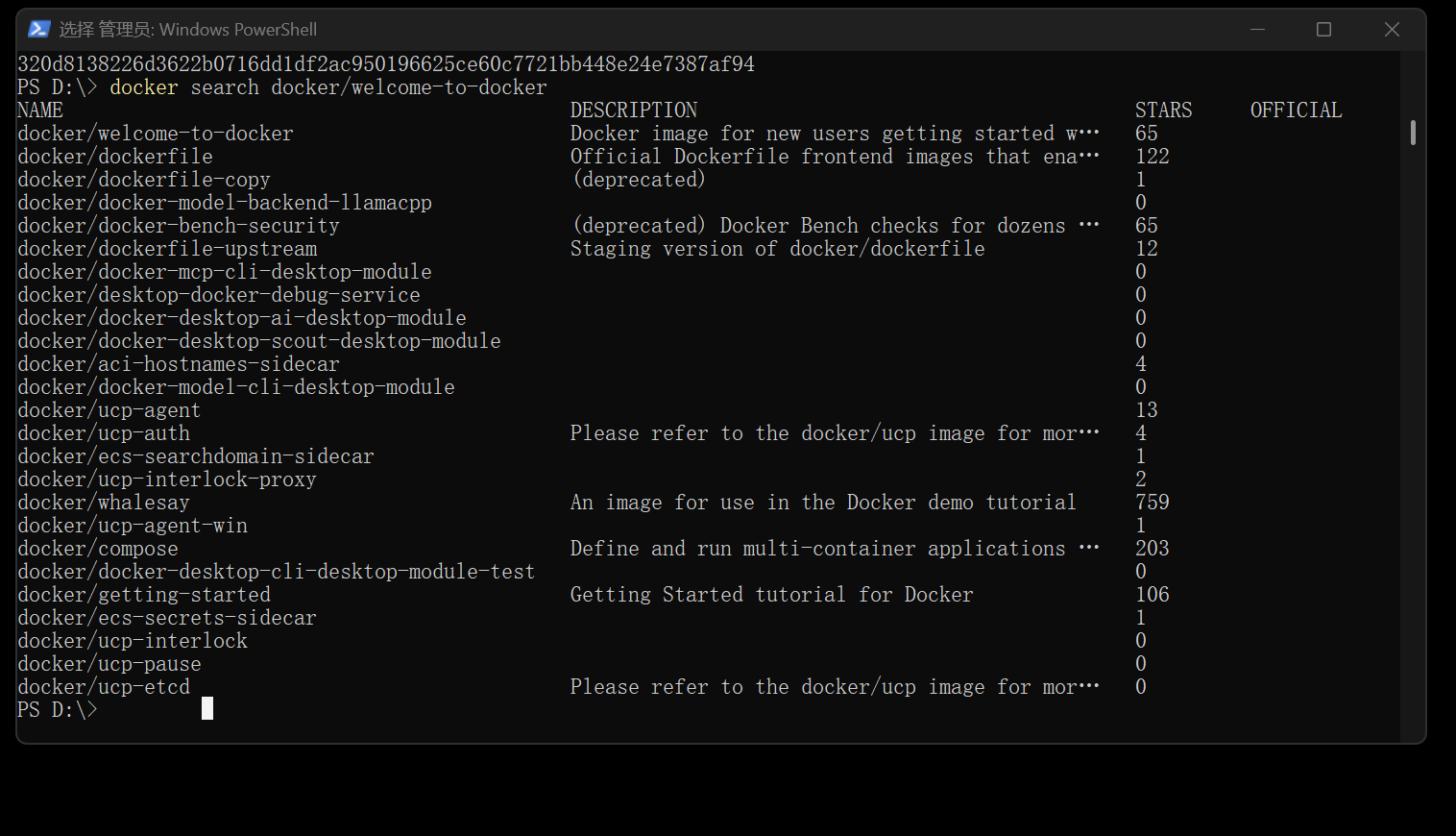


2:

Seeing as a [container](https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/) is an isolated process, where does it get its files and configuration? How do you share those environments?

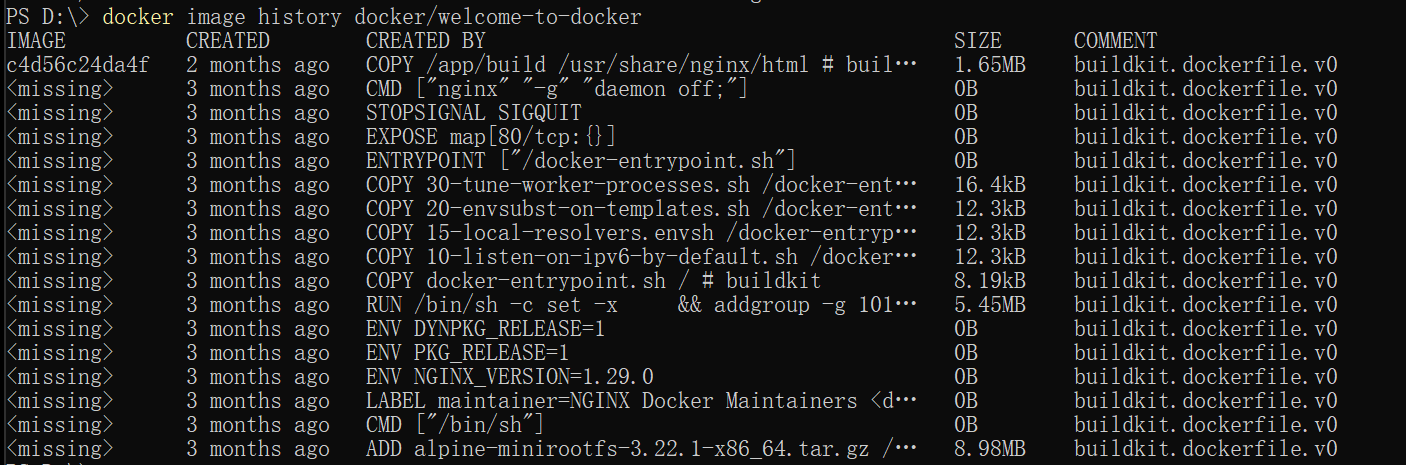
That's where container images come in. A container image is a standardized package that includes all of the files, binaries, libraries, and configurations to run a container.

For a [PostgreSQL](https://hub.docker.com/_/postgres) image, that image will package the database binaries, config files, and other dependencies. For a Python web app, it'll include the Python runtime, your app code, and all of its dependencies.





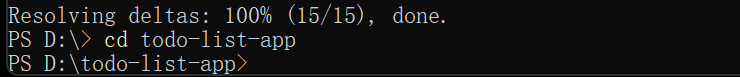
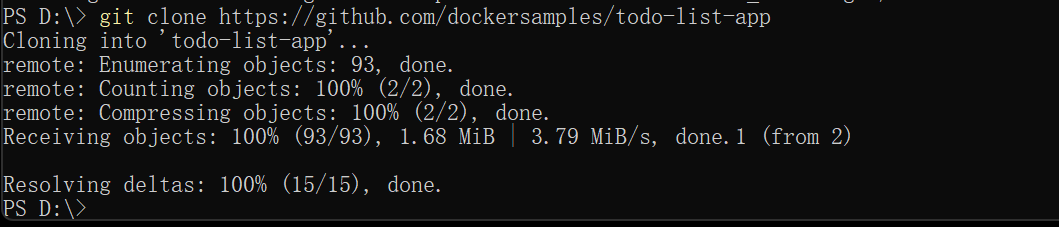


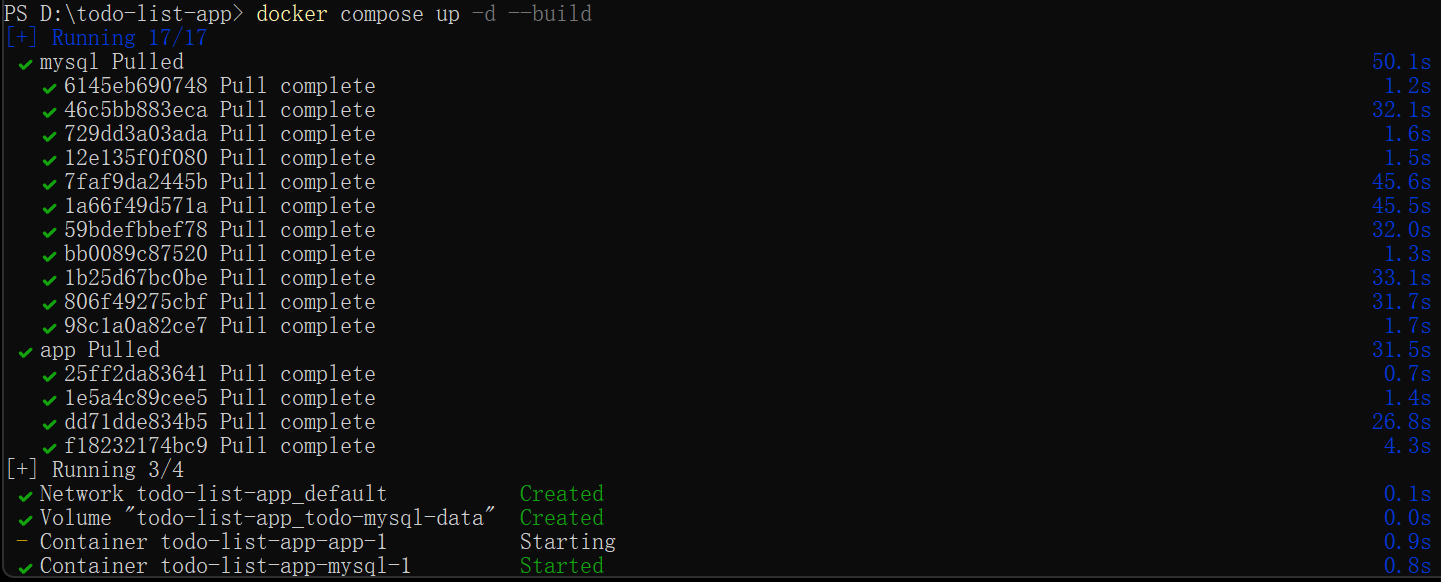


3:

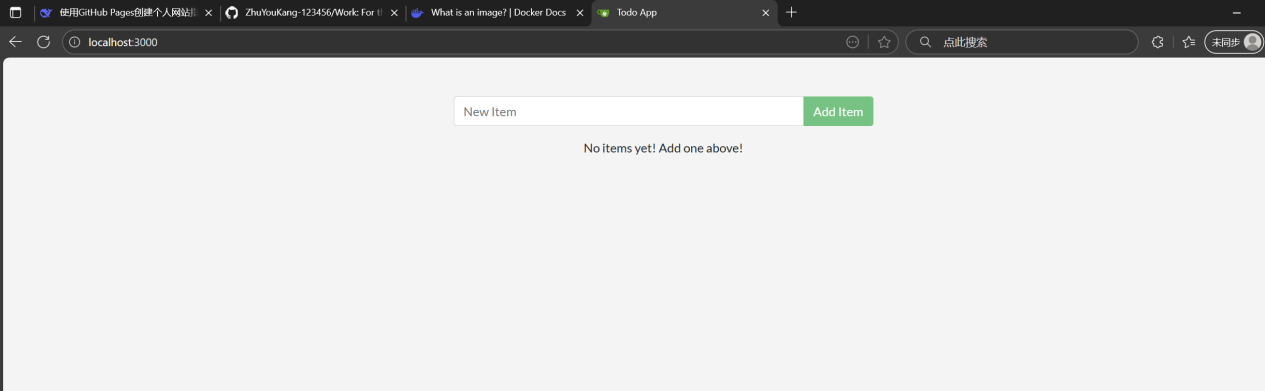
With Docker Compose, you can define all of your containers and their configurations in a single YAML file. If you include this file in your code repository, anyone that clones your repository can get up and running with a single command.

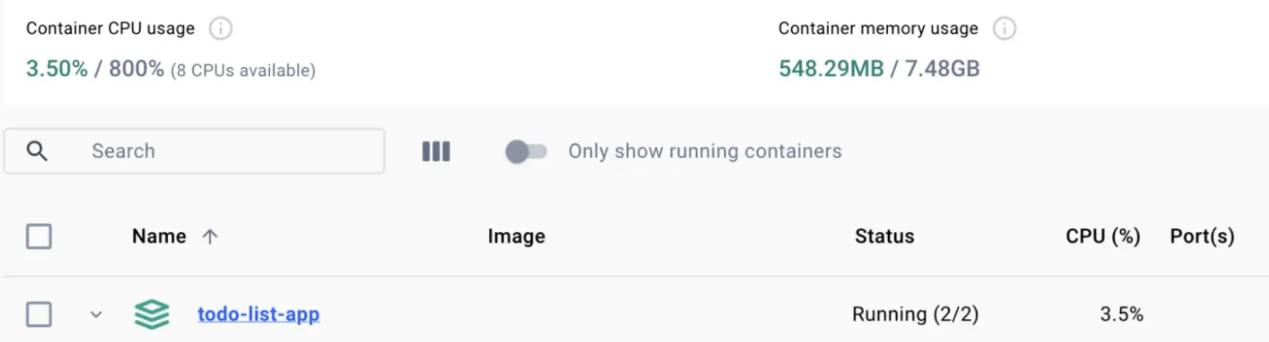
It's important to understand that Compose is a declarative tool - you simply define it and go. You don't always need to recreate everything from scratch. If you make a change, run docker compose up again and Compose will reconcile the changes in your file and apply them intelligently.

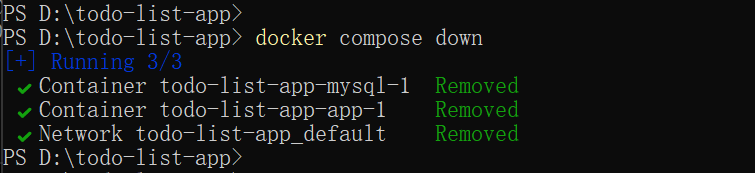


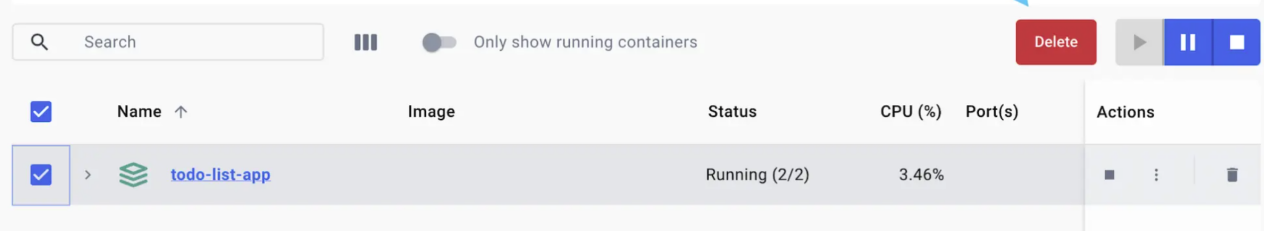




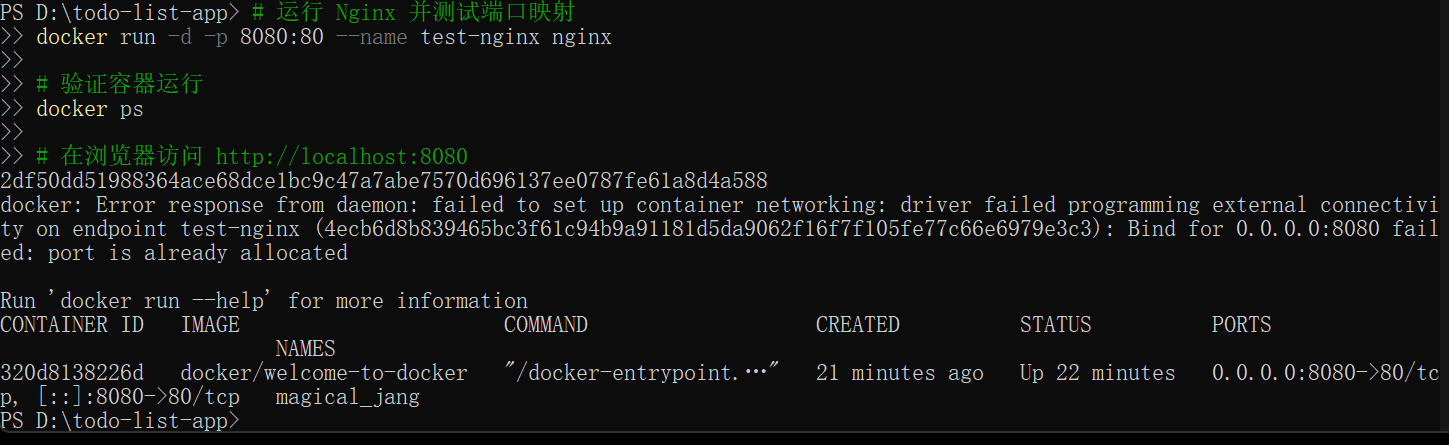


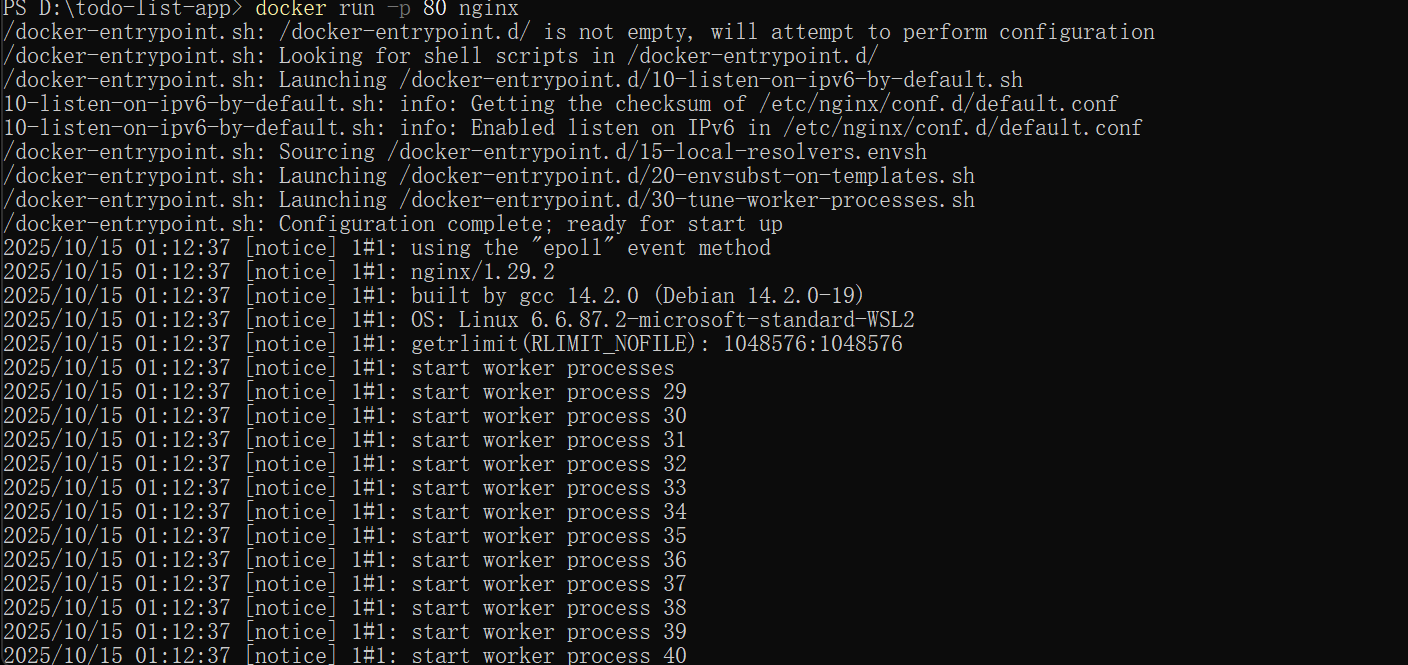


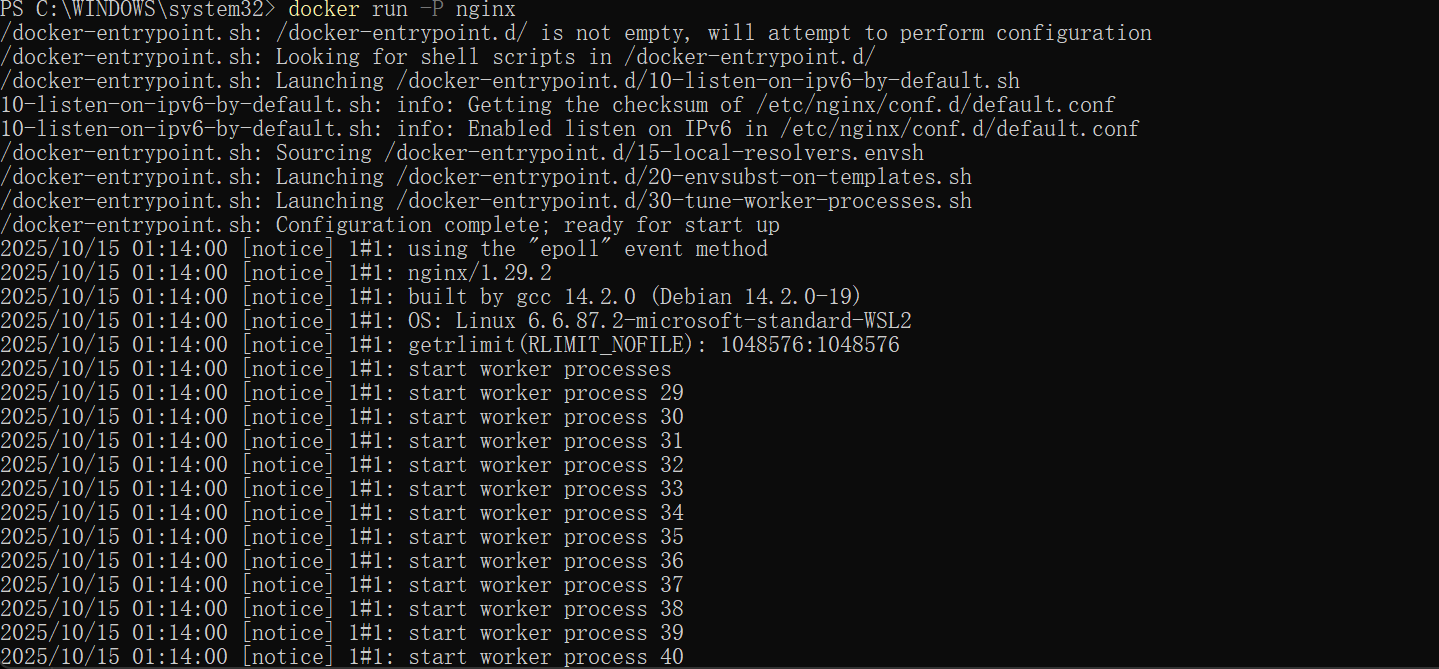




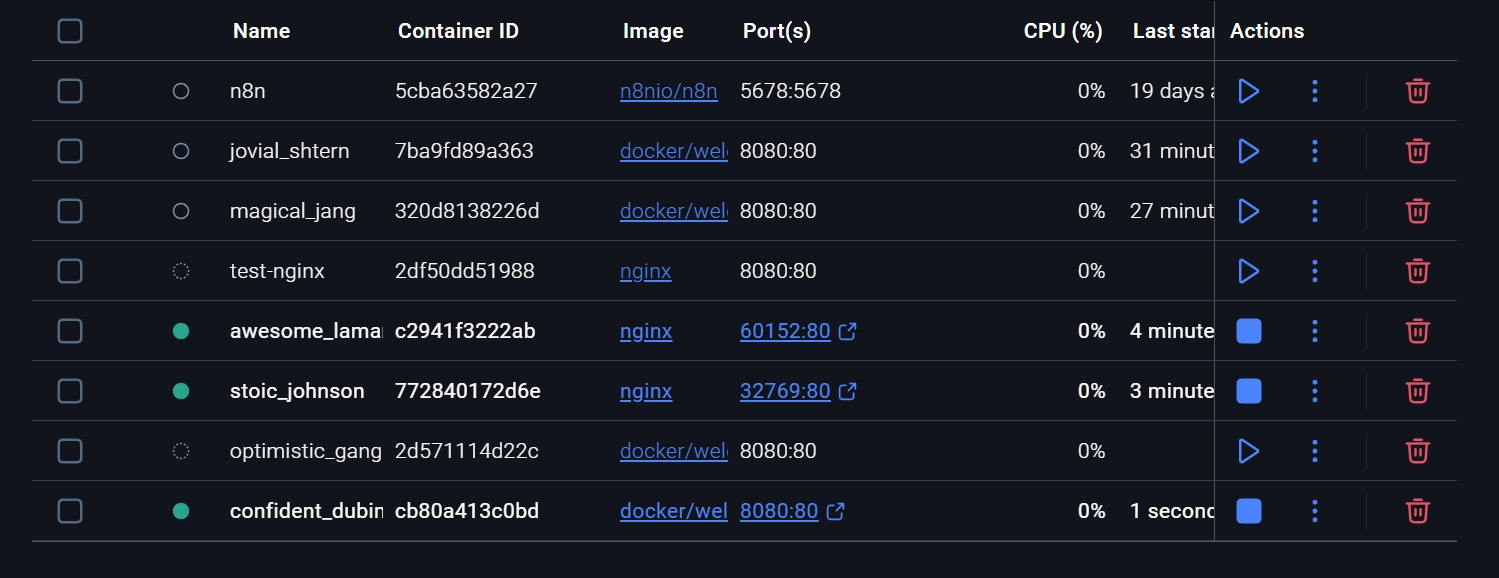
4:

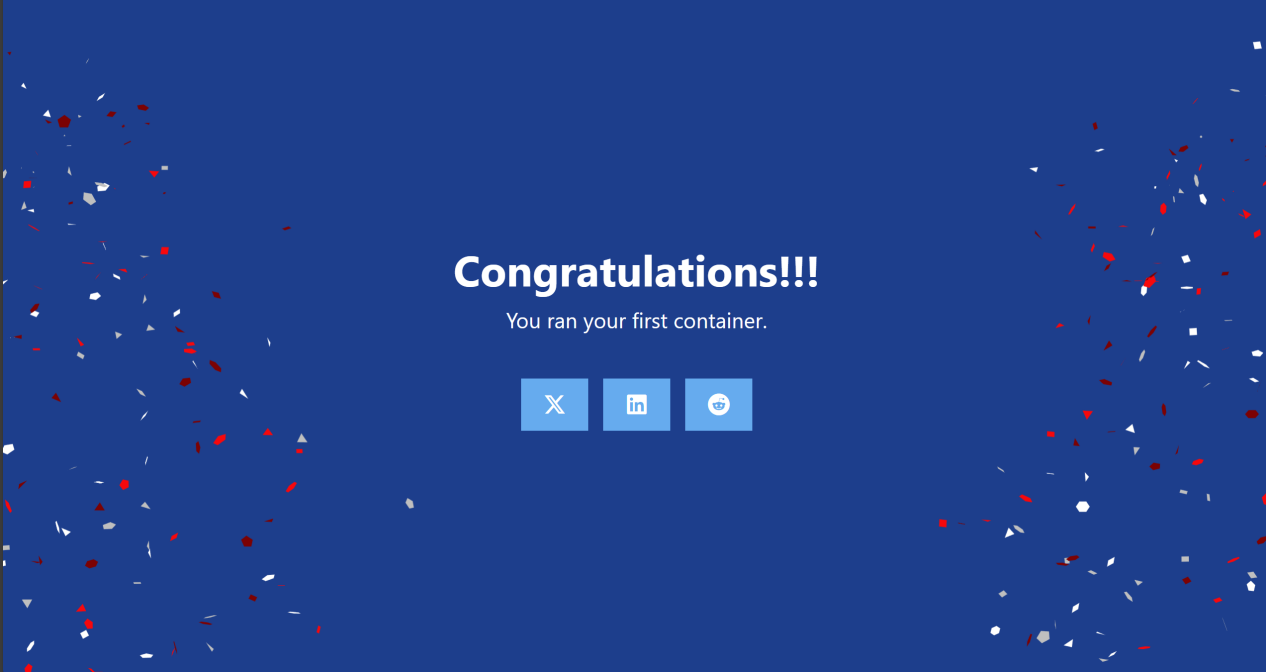
Publishing a port provides the ability to break through a little bit of networking isolation by setting up a forwarding rule. As an example, you can indicate that requests on your host’s port 8080 should be forwarded to the container’s port 80. Publishing ports happens during container creation using the -p (or --publish) flag with docker run.

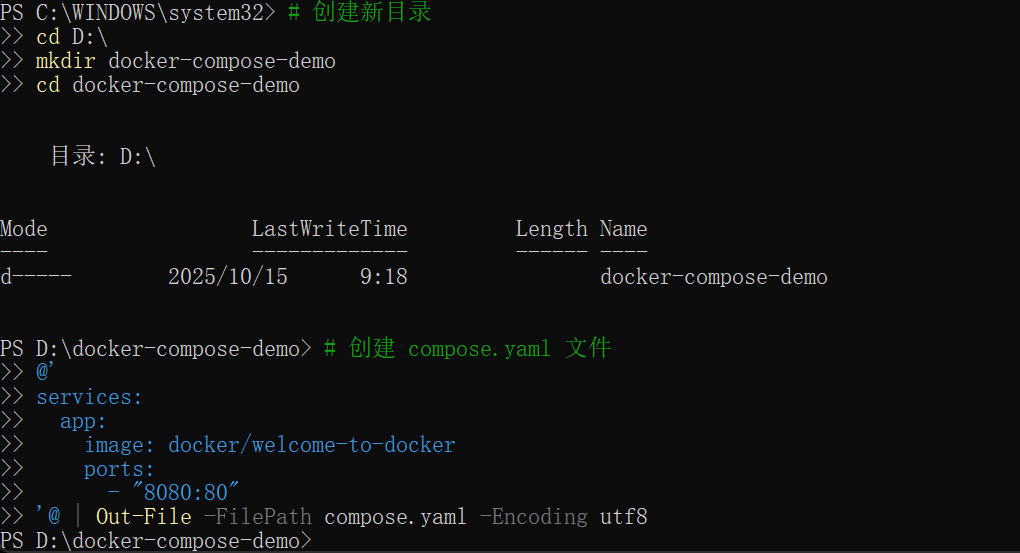


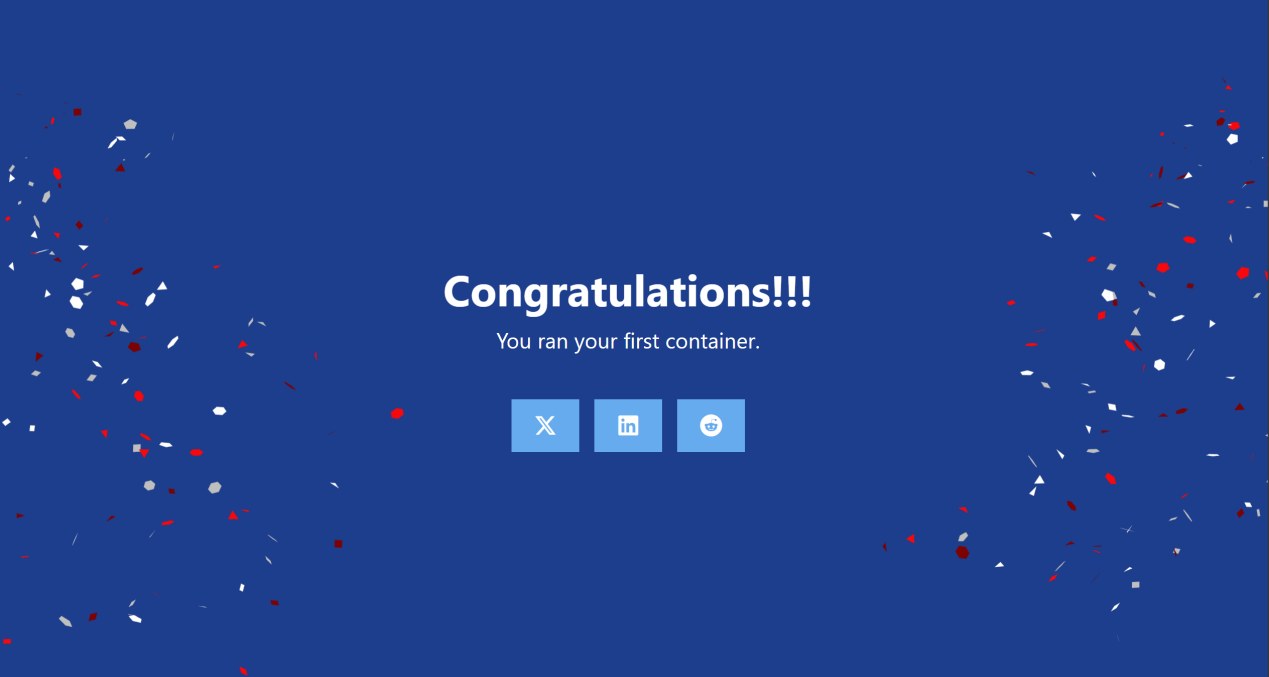








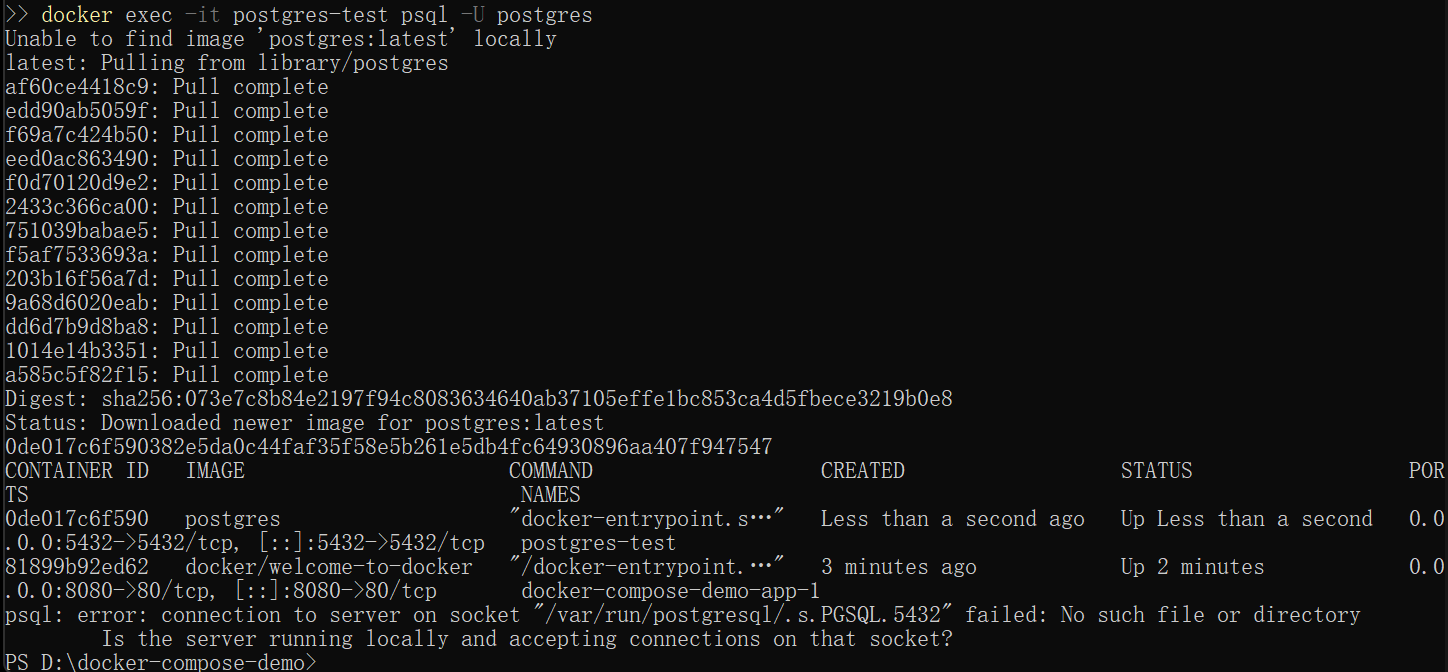


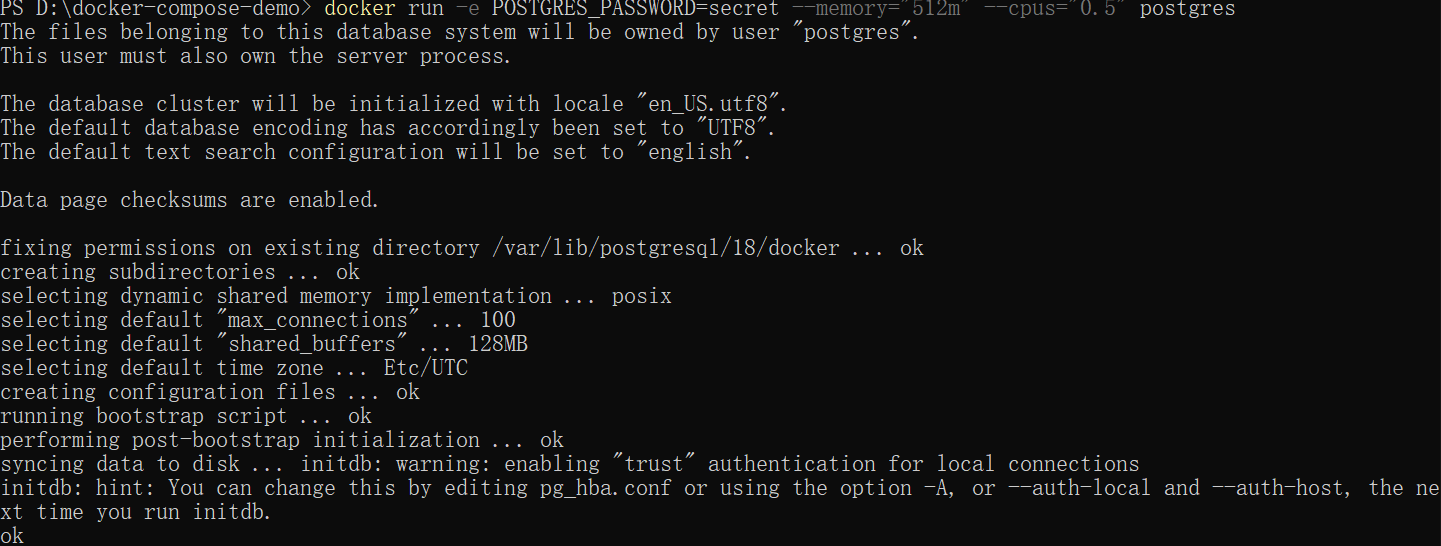


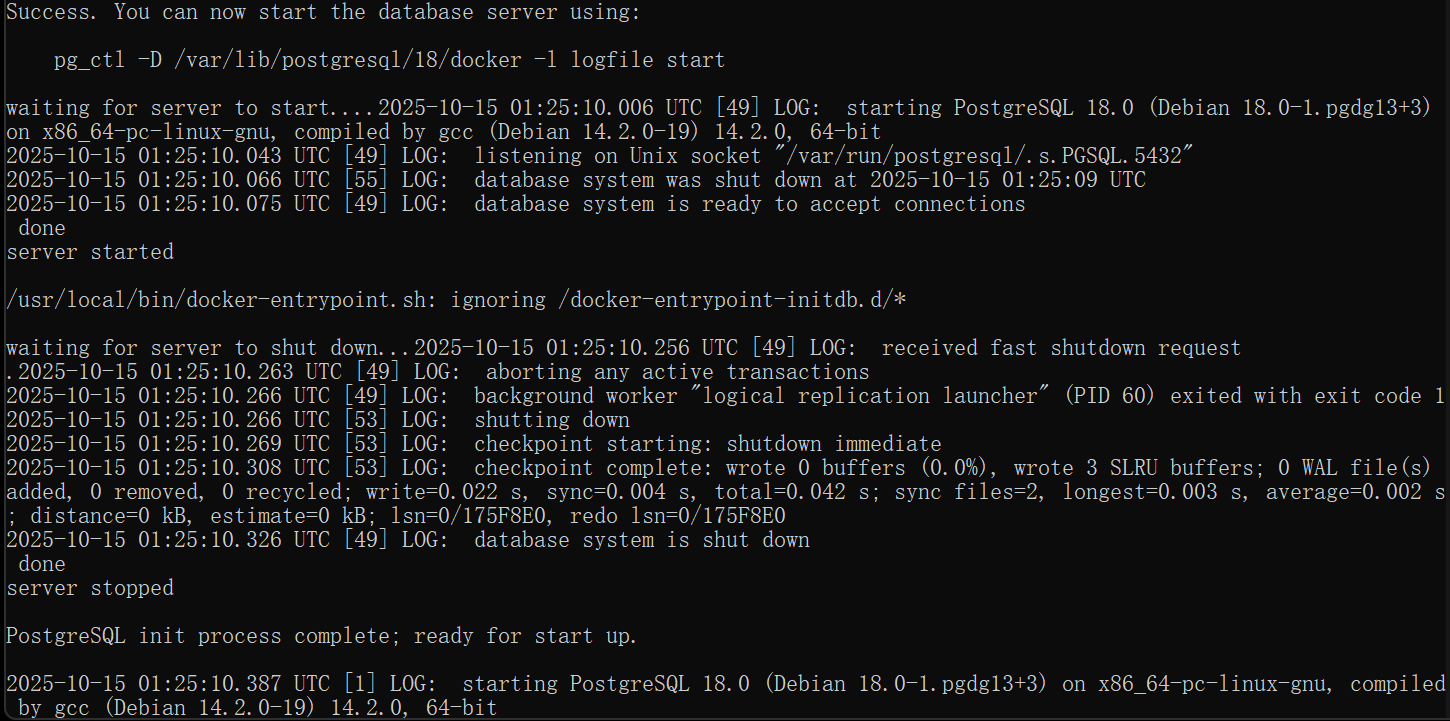
5:

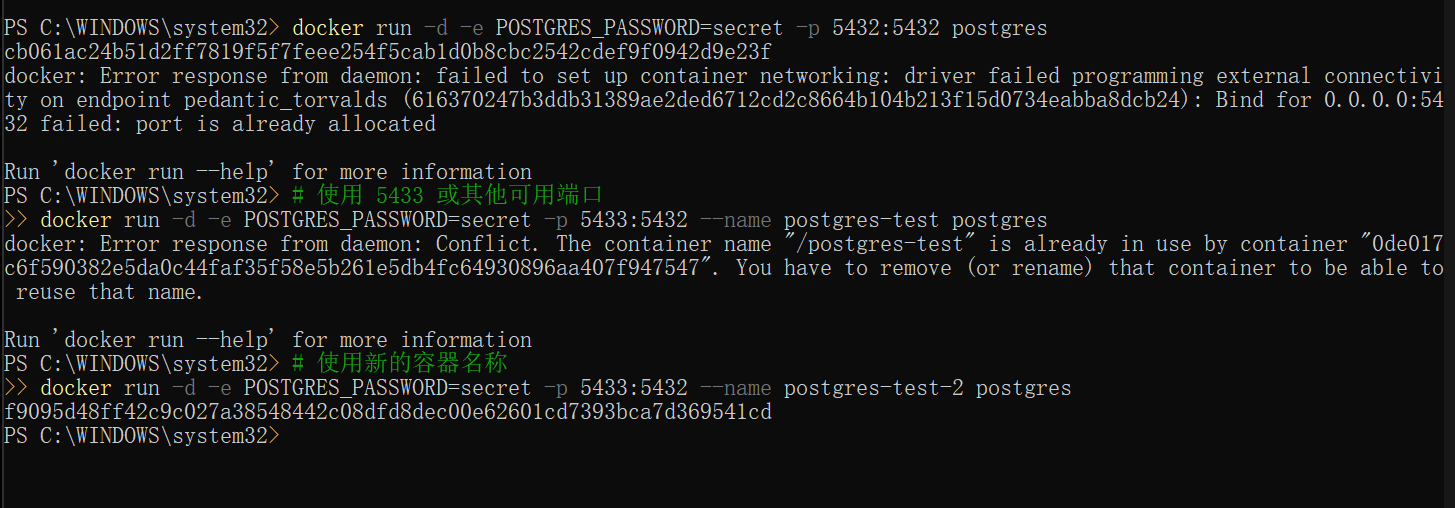
Docker container starts, it executes an application or command. The container gets this executable (script or file) from its image’s configuration. Containers come with default settings that usually work well, but you can change them if needed. These adjustments help the container's program run exactly how you want it to.

For example, if you have an existing database container that listens on the standard port and you want to run a new instance of the same database container, then you might want to change the port settings the new container listens on so that it doesn’t conflict with the existing container. Sometimes you might want to increase the memory available to the container if the program needs more resources to handle a heavy workload or set the environment variables to provide specific configuration details the program needs to function properly.

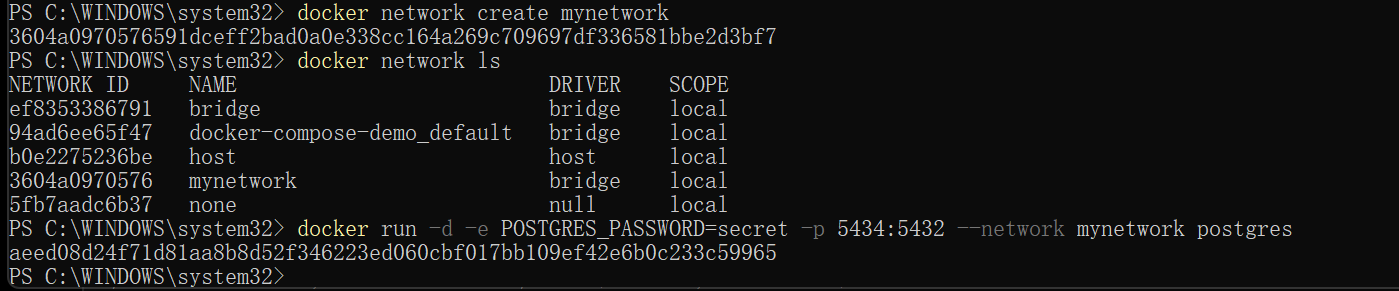




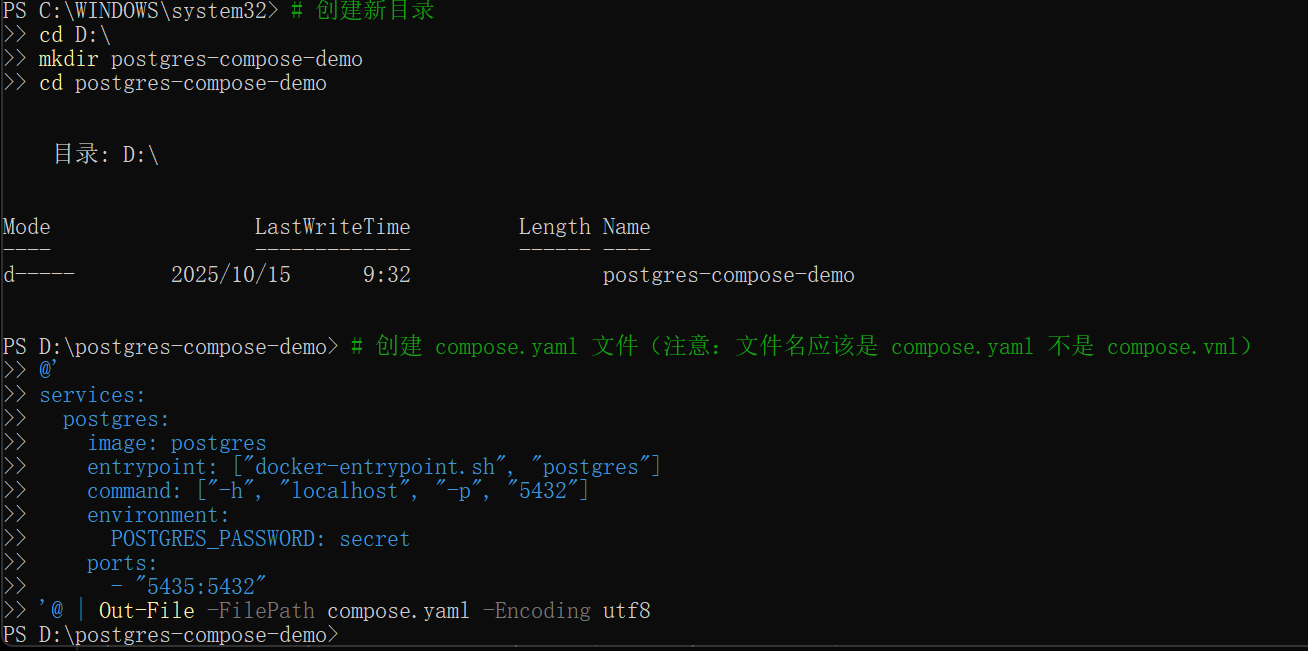


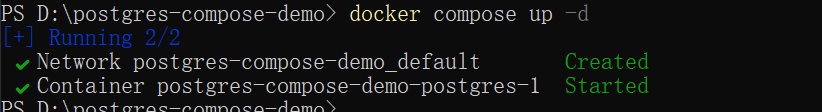


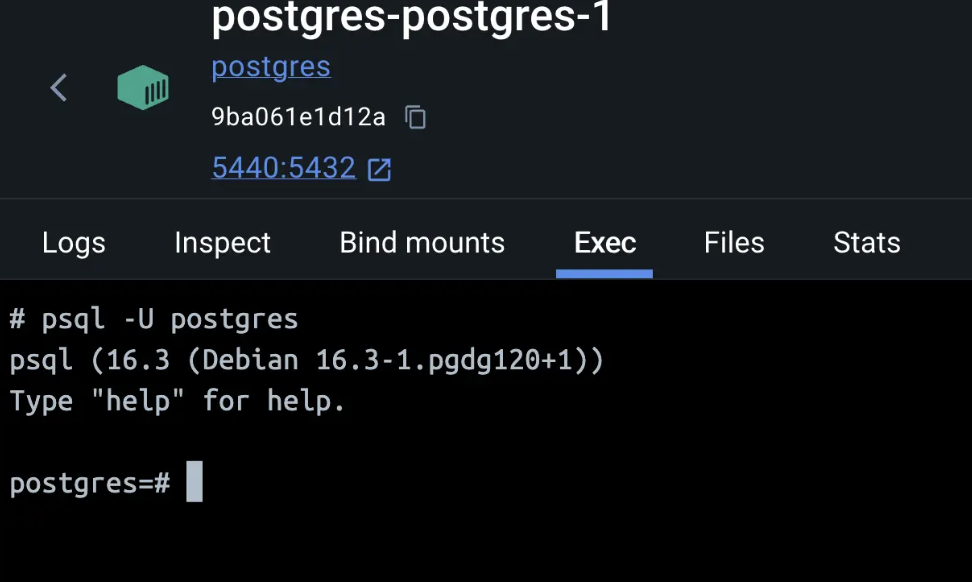


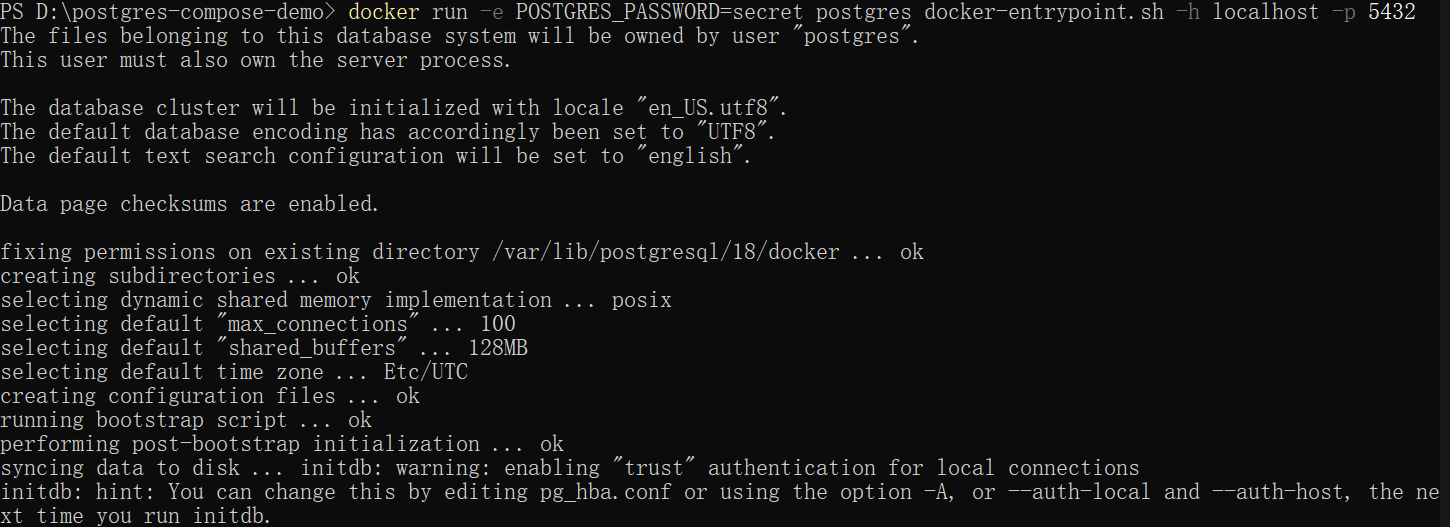








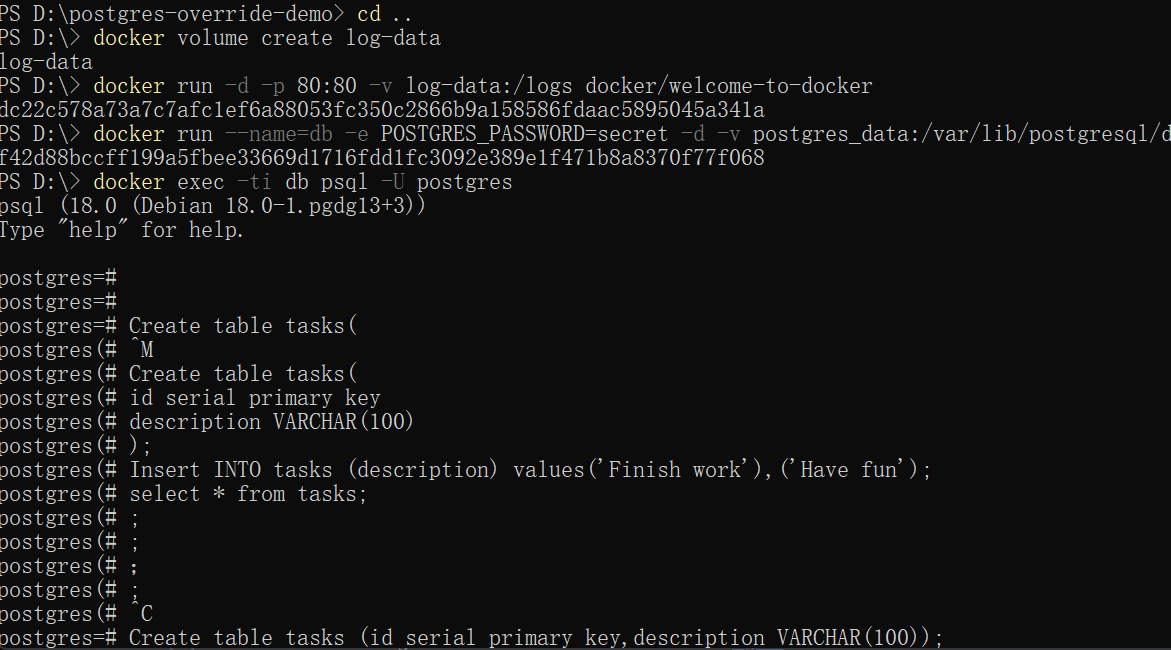


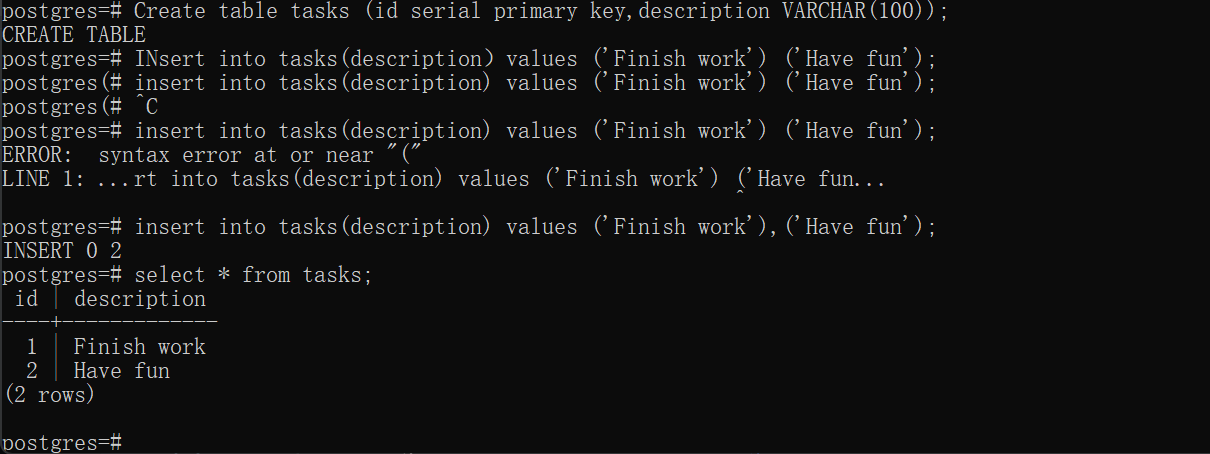


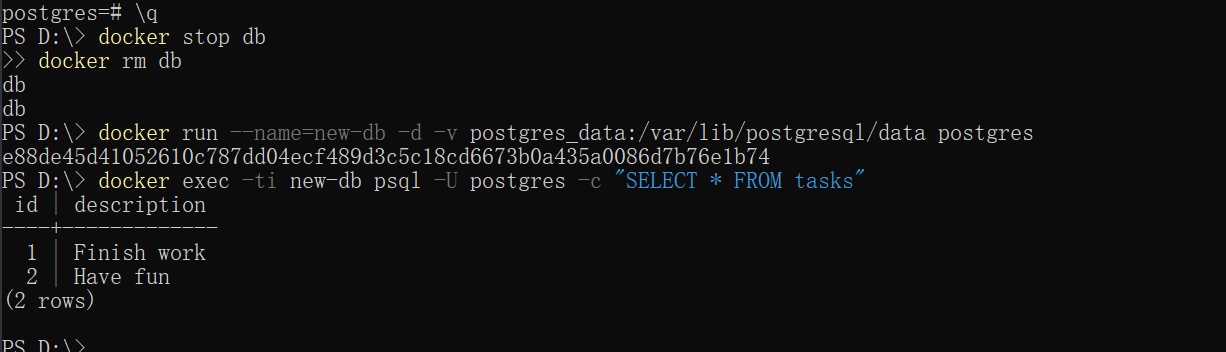
6：

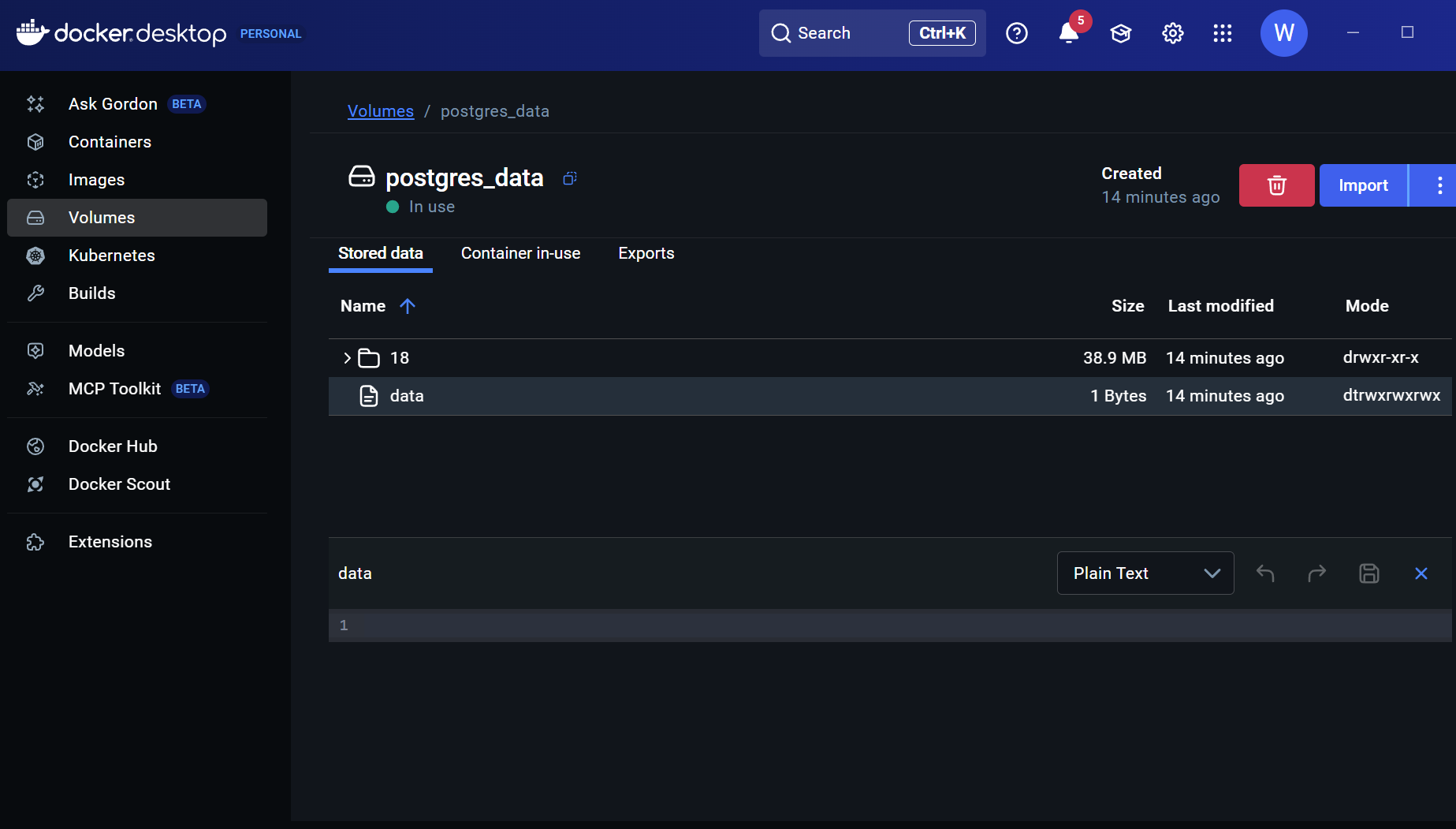
When a container starts, it uses the files and configuration provided by the image. Each container is able to create, modify, and delete files and does so without affecting any other containers. When the container is deleted, these file changes are also deleted.

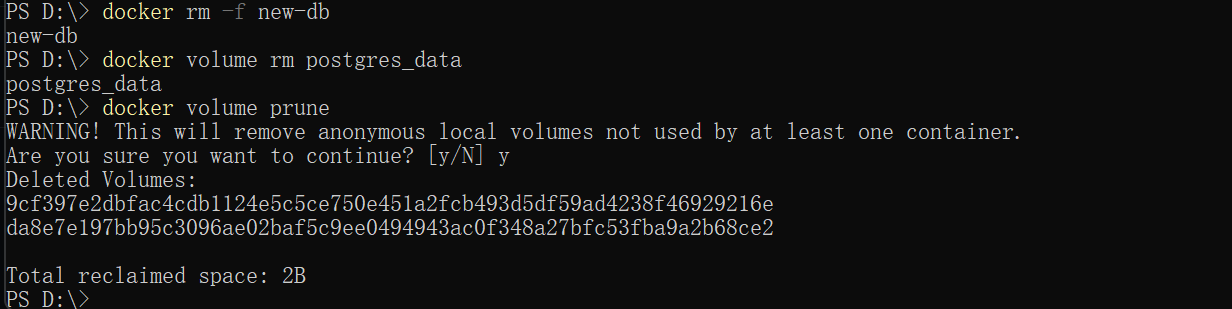
While this ephemeral nature of containers is great, it poses a challenge when you want to persist the data.







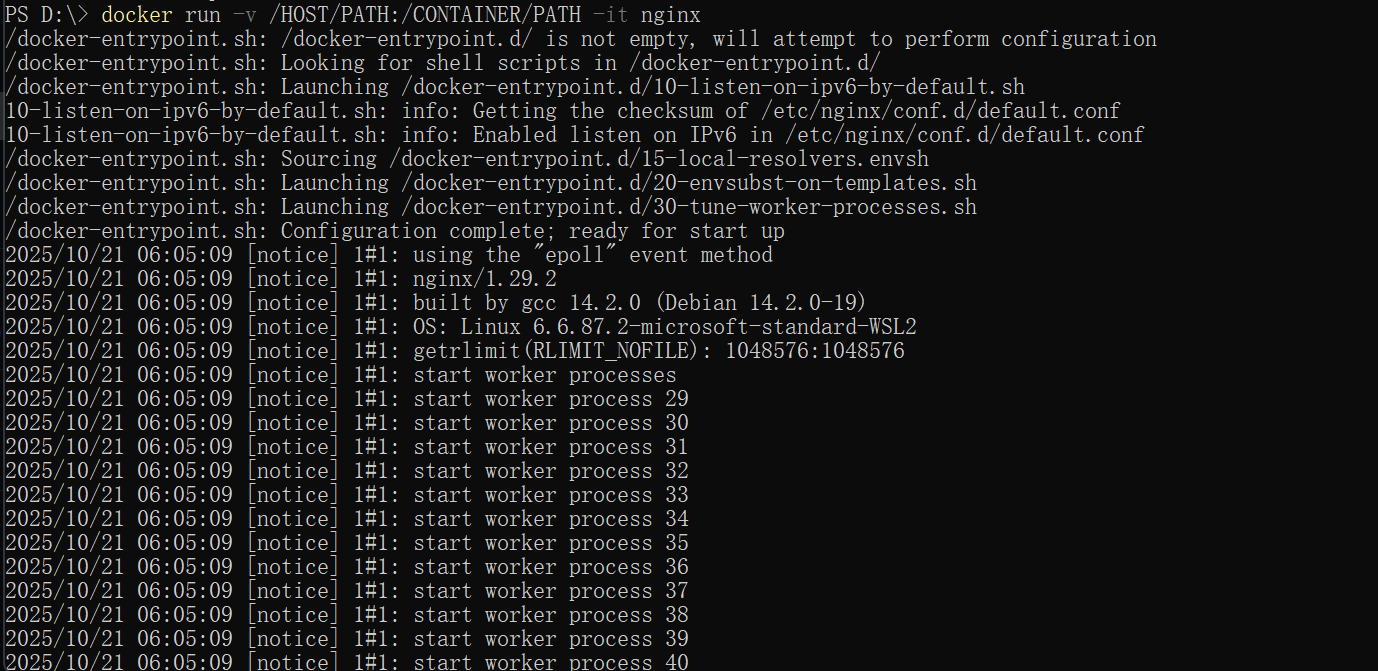


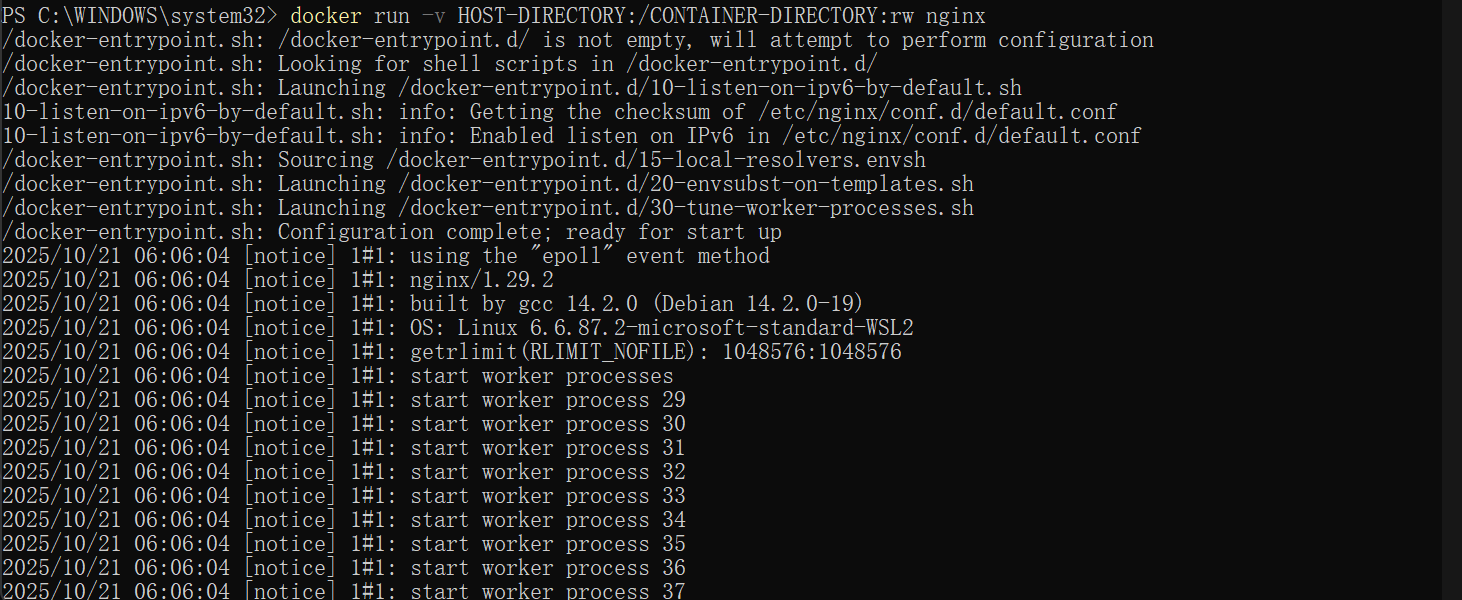


7：

Each container has everything it needs to function with no reliance on any pre-installed dependencies on the host machine. Since containers run in isolation, they have minimal influence on the host and other containers. This isolation has a major benefit: containers minimize conflicts with the host system and other containers. However, this isolation also means containers can't directly access data on the host machine by default.

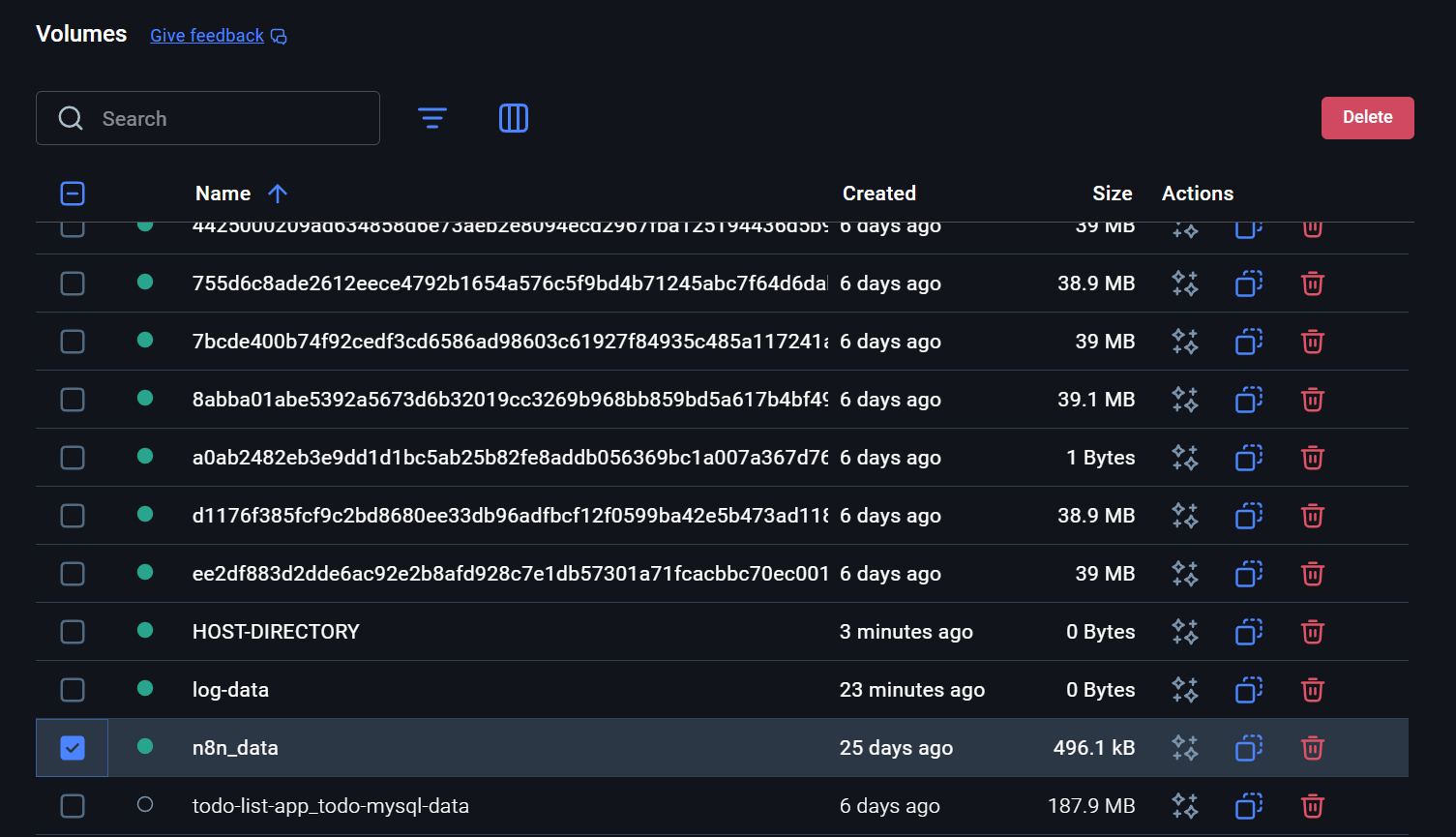
Consider a scenario where you have a web application container that requires access to configuration settings stored in a file on your host system. This file may contain sensitive data such as database credentials or API keys. Storing such sensitive information directly within the container image poses security risks, especially during image sharing. To address this challenge, Docker offers storage options that bridge the gap between container isolation and your host machine's data.



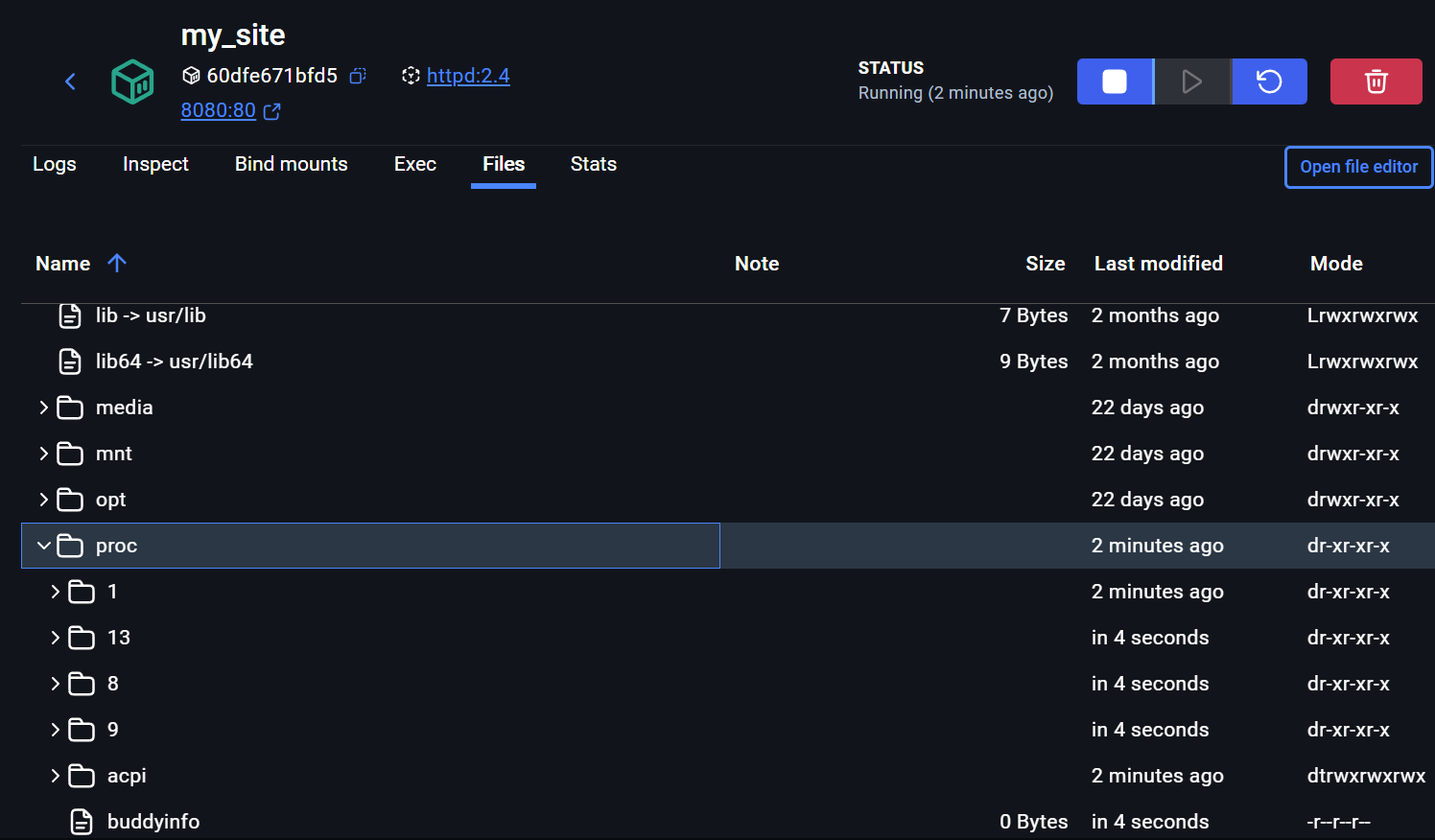


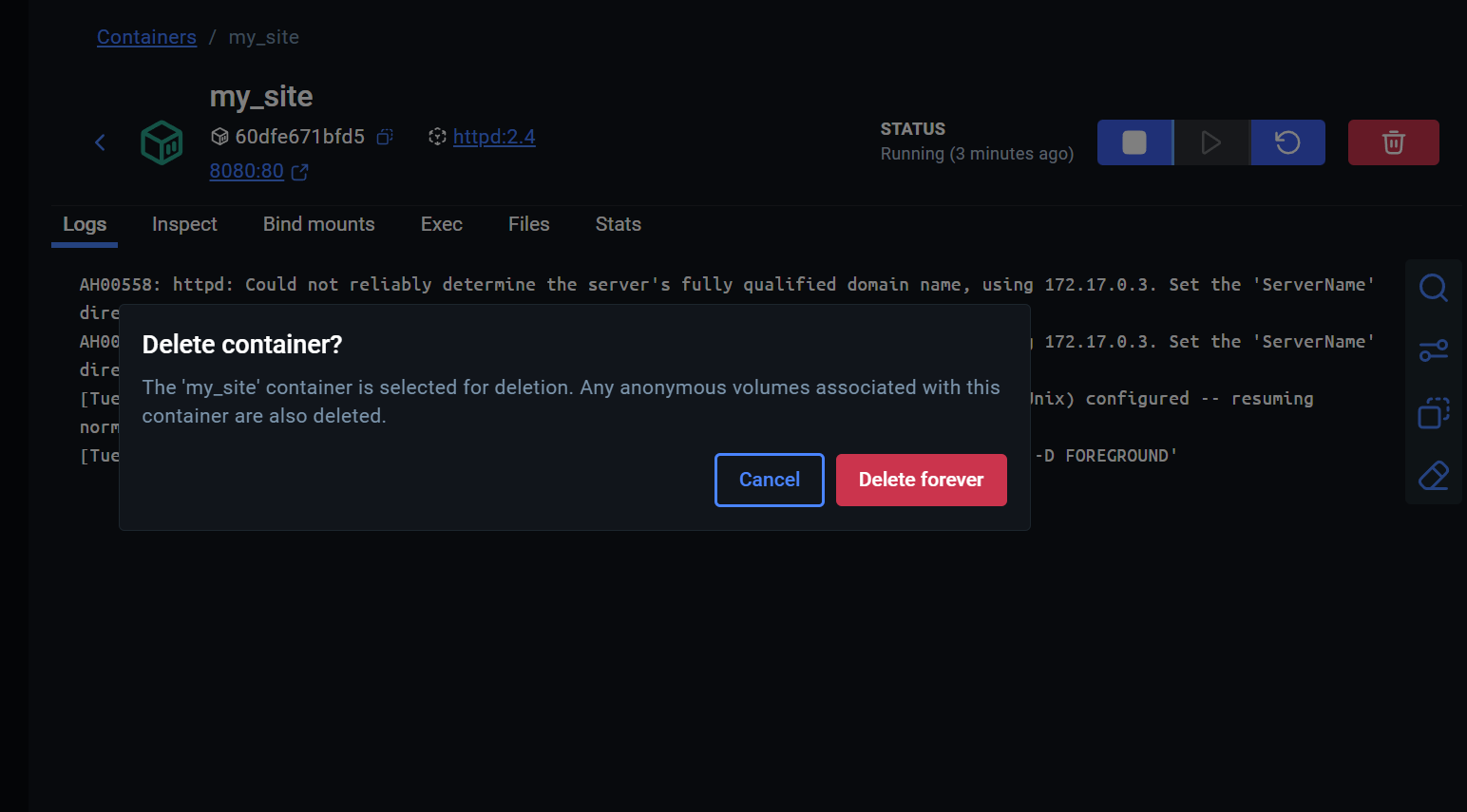












8：

Starting up a single-container application is easy. For example, a Python script that performs a specific data processing task runs within a container with all its dependencies. Similarly, a Node.js application serving a static website with a small API endpoint can be effectively containerized with all its necessary libraries and dependencies. However, as applications grow in size, managing them as individual containers becomes more difficult.

Imagine the data processing Python script needs to connect to a database. Suddenly, you're now managing not just the script but also a database server within the same container. If the script requires user logins, you'll need an authentication mechanism, further bloating the container size.

