Docker Networking

Why the default bridge network doesn't support name resolution between containers, creating user-defined networks with docker network create, container-to-container communication by name, and why localhost means something different inside a container.

Intermediate 30 min

Last updated: August 31, 2026
Read online: https://www.learnforgex.com/en/topics/docker-networking

---

This is the first lesson of "Docker in Practice," the second category in the Docker course. "Docker CLI Fundamentals" already used `-p 5432:5432` to reach a container from the host machine, and "Dockerizing a Spring Boot Application" reached from inside one container to a service on the host via `host.docker.internal` — both worked, but neither one actually explained *how* a container's networking works, or how two containers are meant to reach each other. This lesson fills that gap directly, and replaces that `host.docker.internal` workaround with the real mechanism: running this project's own app and its database as two separate containers that reach each other by name.

The Default Bridge Network

Every container Docker starts is attached to a network — unless told otherwise, that's Docker's own **default bridge network**, a private, isolated network shared by every container that doesn't specify one explicitly. Containers on it can reach the outside internet, and the host machine can reach them through published ports (`-p`, already covered in "Running a Container: `docker run`") — but there's one thing the default bridge network deliberately does *not* provide:

> Containers on Docker's default bridge network **cannot** reach each other by container name — only by IP address, which changes every time a container restarts. This is a genuine, documented Docker limitation, not an oversight to work around with `host.docker.internal` — the actual fix is a **user-defined** network, covered next, which is why every real multi-container setup (including "Docker Compose," later in this category) uses one instead of the default.

Publishing Ports: `-p` Revisited

`-p 8080:8080` (from "Running a Container: `docker run`") maps a port on the host machine to a port inside a container's own, isolated network namespace — without it, a container's ports exist only inside Docker's internal networking and are completely unreachable from the host or the outside world, no matter what `EXPOSE` in its Dockerfile documents. `-p <host-port>:<container-port>` is specifically a **host-to-container** bridge — it has nothing to do with how two containers reach each other, which is exactly the gap the rest of this lesson covers.

Creating a User-Defined Network: `docker network create`

A **user-defined bridge network** is a named network a developer creates explicitly, and — unlike the default one — it comes with automatic DNS-based name resolution between the containers attached to it.

```
docker network create learning-platform-net
```

```
4a3b2c1d0e9f8a7b6c5d4e3f2a1b0c9d8e7f6a5b4c3d2e1f0a9b8c7d6e5f4a3b
```

A container joins it with `--network` on `docker run`:

```
docker run --name learning-platform-db --network learning-platform-net -e
POSTGRES_PASSWORD=secret -d postgres:16
```

Container-to-Container Communication by Name

This is the actual payoff: on a user-defined network, one container can reach another using the **other container's `--name`** as a hostname — Docker runs an embedded DNS server for exactly this purpose.

```
docker run --name learning-platform-app \
  --network learning-platform-net \
  -e SPRING_DATASOURCE_URL=jdbc:postgresql://learning-platform-db:5432/postgres \
  -d learning-platform:0.1.0
```

`learning-platform-db` in that JDBC URL isn't a real DNS hostname anywhere on the internet — it resolves *only* inside `learning-platform-net`, to whatever container is currently running with that name on that network. This is a strictly better replacement for the `host.docker.internal` workaround "Dockerizing a Spring Boot Application" used: that trick reaches a

service on the *host machine*, a fundamentally different (and less portable) situation than two containers that are both meant to run together, which is what a real deployment of this project actually looks like.

Inspecting Networks: `docker network ls` / `docker network inspect`

```
docker network ls
```

```
NETWORK ID     NAME                    DRIVER    SCOPE
a1b2c3d4e5f6   bridge                  bridge    local
b2c3d4e5f6a7   learning-platform-net   bridge    local
```

`docker network inspect <name>` shows exactly which containers are currently attached to a given network — the fastest way to confirm two containers that are supposed to reach each other are actually on the same one, which is the single most common reason "Container-to-Container Communication by Name" doesn't work as expected (see "Common Mistakes").

Why `localhost` Means Something Different Inside a Container

Every container gets its own, isolated network namespace — which means `localhost` (or `127.0.0.1`) *inside* a container refers to that container's own loopback interface, not the host machine's, and not any other container's. A Spring Boot application running directly on a developer's own machine, connecting to a locally installed PostgreSQL via `jdbc:postgresql://localhost:5432/...`, cannot simply reuse that same URL once both the application and PostgreSQL move into separate containers — `localhost` from inside the app's container would mean the app container itself, which isn't running PostgreSQL at all. This is exactly why "Container-to-Container Communication by Name" uses the *other container's name*, `learning-platform-db`, instead of `localhost`.

Putting It All Together

The complete flow — create a network, run this project's own database and application containers on it, and confirm the application actually reached the database by name:

```
#!/bin/sh
# Running this project's own app and its PostgreSQL database as two separate
# containers that reach each other by name, instead of the host.docker.internal
# workaround "Dockerizing a Spring Boot Application" used as a stand-in.

docker network create learning-platform-net
# Output:
# 4a3b2c1d0e9f8a7b6c5d4e3f2a1b0c9d8e7f6a5b4c3d2e1f0a9b8c7d6e5f4a3b

docker run --name learning-platform-db \
  --network learning-platform-net \
  -e POSTGRES_PASSWORD=secret \
  -d postgres:16
# Output:
# 7f8e9a0b1c2d3e4f5a6b7c8d9e0f1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f

docker run --name learning-platform-app \
  --network learning-platform-net \
  -p 8080:8080 \
  -e SPRING_DATASOURCE_URL=jdbc:postgresql://learning-platform-db:5432/postgres \
  -e SPRING_DATASOURCE_PASSWORD=secret \
  -d learning-platform:0.1.0
# Output:
# 9b8c7d6e5f4a3b2c1d0e9f8a7b6c5d4e3f2a1b0c9d8e7f6a5b4c3d2e1f0a9b8c

# Note the datasource URL: "learning-platform-db" -- the OTHER container's
# --name -- resolves automatically over this user-defined network, no IP
# address and no host.docker.internal needed. Confirm it actually worked:

docker logs learning-platform-app
# Output (trimmed):
# Started LearningPlatformApplication in 3.4 seconds

curl http://localhost:8080/en
# Output (trimmed):
# <!doctype html> ... the homepage renders, meaning the app reached the database ...

docker network inspect learning-platform-net --format '{{range .Containers}}{{.Name}}
{{end}}'
# Output:
# learning-platform-db learning-platform-app

# Clean up.
docker stop learning-platform-app learning-platform-db
docker rm learning-platform-app learning-platform-db
docker network rm learning-platform-net
```

Everything here is a direct application of what this lesson covered: `docker network create` makes the network, `--network` attaches both containers to it, and the JDBC URL reaches the database container by its `--name` rather than by IP or `localhost` — exactly the mechanism "Docker Compose," the next lesson in this category, automates so that this network-creation step never has to be typed by hand.

Common Mistakes

- Expecting two containers to reach each other by name on Docker's **default** bridge network — they can't; that only works on a user-defined network (see "The Default Bridge Network").
- Using `localhost` in a container's own configuration to reach a *different* container — inside a container, `localhost` always means that container itself, never another one (see "Why `localhost` Means Something Different Inside a Container").
- Forgetting `--network` on one of two containers meant to talk to each other, and being confused when name resolution fails — `docker network inspect` (above) is the fastest way to confirm both are actually attached to the same network.
- Confusing `host.docker.internal` (reaching the *host machine* from inside a container) with container-to-container name resolution (reaching *another container*) — they solve different problems and aren't interchangeable.

Best Practices

- Always create and use a user-defined network for any setup with more than one container that needs to talk to each other — never rely on the default bridge network for that.
- Reach for the other container's `--name` as a hostname in connection strings and configuration, instead of a hardcoded IP address that changes on every restart.
- Use `docker network inspect` as a first troubleshooting step whenever containers that are supposed to communicate can't reach each other.
- Keep `-p` (host-to-container) and `--network` (container-to-container) mentally separate — they solve different problems and neither one substitutes for the other.

Summary, Cheat Sheet, and Glossary

Summary

- Every container is attached to a network — by default, Docker's own default bridge network, which does **not** support reaching other containers by name.
- `-p <host>:<container>` bridges the host machine to a container's port; it has nothing to do with how two containers reach each other.
- A user-defined network (`docker network create`), unlike the default one, gives every attached container automatic DNS resolution of every other attached container's `--name`.
- Inside a container, `localhost` always refers to that container's own loopback interface — never the host machine's, and never another container's.

- `host.docker.internal` (reaching the host from inside a container) and a container's `--name` on a user-defined network (reaching another container) solve two different problems.

Cheat Sheet

```
docker network create <name>                # create a user-defined bridge network
docker run --network <name> ...              # attach a container to it
docker network ls                            # list all networks
docker network inspect <name>                # show which containers are attached
```

```
jdbc:postgresql://<other-container-name>:5432/...   # reach another container by name
                                                       # (only works on a shared user-
defined network)
```

Glossary

- **Default bridge network**: the network every container joins unless told otherwise; does not support reaching other containers by name.
- **User-defined network**: a named network created with `docker network create` that provides automatic DNS-based name resolution between attached containers.
- **Port publishing (`-p`)**: mapping a host machine port to a container's port — a host-to-container bridge, unrelated to container-to-container communication.
- **`host.docker.internal`**: a special hostname a container can use to reach a service running on the host machine — not for reaching another container.