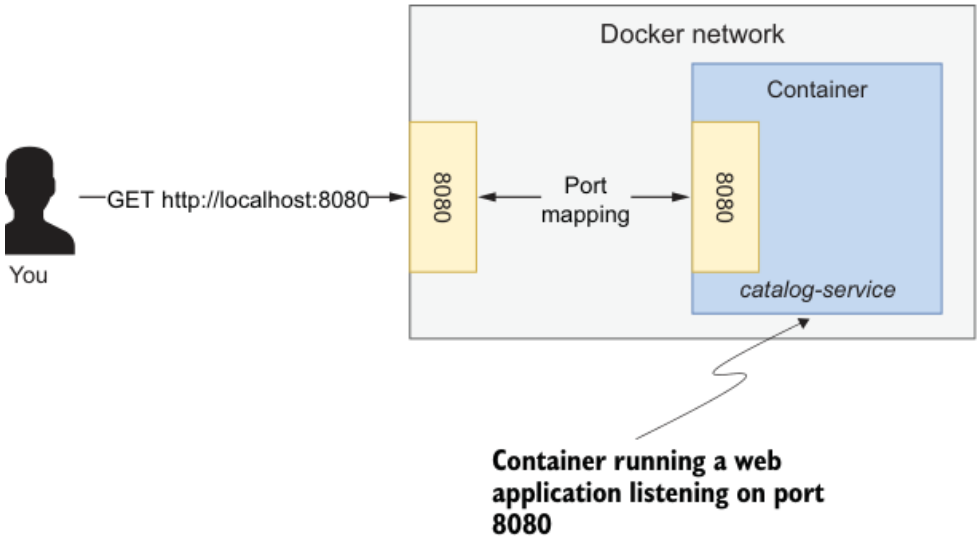
DOCKER-V2
// last updated 30.07.2025
// code
https://github.com/hong1234/springOnDockerV1

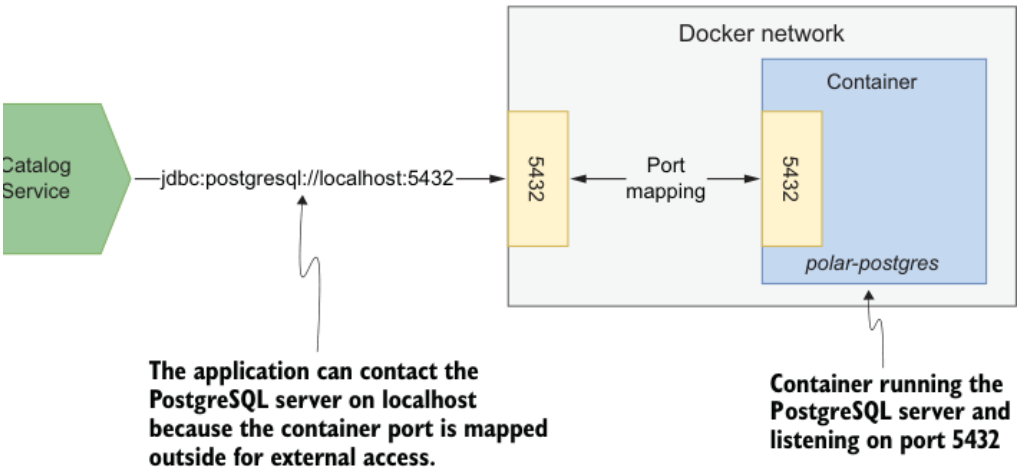
EXPOSING APPLICATION SERVICES THROUGH PORT FORWARDING (189)

By default, containers join an isolated network inside the Docker host. If you want to access any container from your local network, you must explicitly configure the port mapping.

For example, when you ran the Catalog Service application, you specified the mapping as an argument to the docker run command: -p 8080:8080 (where the first is the external port and the second is the container port). Figure 6.8 illustrates how this works.



Thanks to port forwarding, the Catalog Service application could access the PostgreSQL database server through the URL jdbc:postgresql://*localhost:5432*, even if it was running inside a container. The interaction is shown in Figure 6.9.

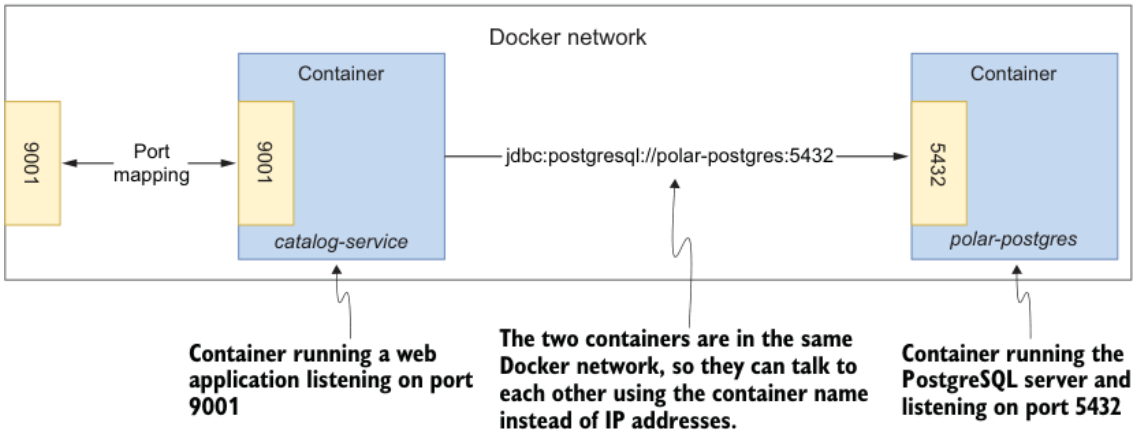


Note:
When running Catalog Service as a container, however, you will not be able to do that anymore, since localhost would represent the inside of your container and not your local machine. How can you solve this problem?

USING DOCKER'S BUILT-IN DNS SERVER FOR SERVICE DISCOVERY (189)

Docker has a built-in DNS server that can enable *containers in the same network to find each other using the container name* rather than a hostname or an IP address.

For example, Catalog Service will be able to call the PostgreSQL server through the URL jdbc:postgresql://*polar-postgres:5432*, where polar-postgres is the container name. Figure 6.10 shows how it works.



So before moving on, let's create a network inside which Catalog Service and PostgreSQL can talk to each other using the container name instead of an IP address or a hostname. You can run this command from any Terminal window:

```
$ docker network create catalog-network
```

Next, verify that the network has been successfully created:

```
$ docker network ls
```

You can then start a PostgreSQL container, specifying that it should be part of the *catalog-network* you just created. Using the --net argument ensures the container will join the specified network and rely on the Docker built-in DNS server:

```
$ docker run -d \
--name polar-postgres \
--net catalog-network \
-e POSTGRES_USER=user \
-e POSTGRES_PASSWORD=password \
-e POSTGRES_DB=polardb_catalog \
-p 5432:5432 \
postgres:14.4
```

You can then start the web service container

```
$ docker run -d \
--name catalog-service \
--net catalog-network \
-p 9001:9001 \
-e SPRING_DATASOURCE_URL=jdbc:postgresql://polar-postgres:5432/polardb_catalog \
-e SPRING_PROFILES_ACTIVE=docker \  // note: dont work ? by sb3
catalog-service
```

Remember the two aspects: port forwarding and using the Docker built-in DNS server. You can handle them by adding two arguments to the docker run command:

-p 9001:9001 will map port 9001 inside the container (where the Catalog Service is exposing its services) to port 9001 on your localhost.
--net catalog-network will connect the Catalog Service container to the catalog-network you previously created so that it can contact the PostgreSQL container.

We need *to overwrite* the spring.datasource.url property. In the previous section, we set the spring.datasource.url property for Catalog Service to jdbc:postgresql://localhost:5432/polardb_catalog. Since it points to localhost, it will not work from within a container. We need replacing localhost with the PostgreSQL container name: polar-postgres.

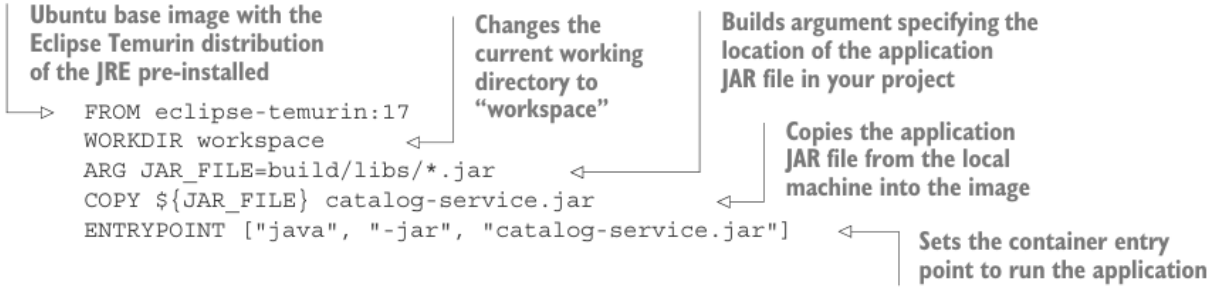
You already know how to configure a Spring Boot application from the outside without having to recompile it, right? *An environment variable will do*. Using another environmentvariable, we can also enable the testdata Spring profile to trigger the creation of test data in the catalog:

Open browser, call the application, and verify that it works correctly
http:9001/books

Containerizing your application

Open your Catalog Service project (catalog-service), and create an empty file called Dockerfile (with no extension) in the root folder. That file will contain the recipe for containerizing your application.

Listing 6.2 Dockerfile for describing the Catalog Service image



Before moving on, you need to build the JAR artifact for the Catalog Service application. Open a Terminal window and navigate to the Catalog Service project's root folder. First, build the JAR artifact:
$ ./gradlew clean bootJar

By default, the Dockerfile script will copy the application's JAR file from the location path used by Gradle: build/libs/. So if you're using Gradle, you can build the container image by running this command:
$ docker build -t catalog-service .

Note:
// project-root\src\main\resources*application-docker.properties*

DataSource Configuration

spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=${DATASOURCE_URL}
spring.datasource.username=${POSTGRES_USER}
spring.datasource.password=${POSTGRES_PASSWORD}
spring.datasource.initialize=true

Hibernate Configuration
…

Managing Spring Boot containers with Docker Compose

When it comes to running multiple containers, the Docker CLI can be a bit cumbersome. Writing commands in a Terminal window can be error-prone, hard to read, and challenging when it comes to applying version control.

Instead of a command line, you work with YAML files that describe which containers you want to run and their characteristics. With Docker Compose, you can define all the applications and services composing your system in one place, and you can manage their life cycles together.

```
// project-root\docker-compose.yml

version: '3.8'
services:

  postgres-db:
    image: postgres:latest
    container_name: postgres-container
    volumes:
      - product-data:/var/lib/postgresql/data
    ports:
      - 5432:5432
    environment:
      - DB_NAME=demodb
      - DB_USER=postgres
      - DB_PASSWORD=postgres
    # networks:
      # - demo-network

  springboot-web:
    image: hongle/springboot-demo
    container_name: web-container
    depends_on:
      - "postgres-db"
    environment:
      - DB_SERVER=postgres-container:5432
      # - SPRING_PROFILES_ACTIVE=docker
    ports:
      - 80:8080
    # networks:
      # - demo-network

volumes:
  product-data:

# networks:
  # demo-network:
```