DOCKERIZING SPRING BOOT App
// Book: Cloud Native Spring in Action – 2023; last Updated 07.2025

2.3.2 Running a Spring application as a container (55)

Let's go back to Catalog Service and see how you can run it as a container.
Open a Terminal window, navigate to the root folder of your Catalog Service project (catalog-service), and run the bootBuildImage Gradle task.
That's all you need to do to package your application as a container image, using Cloud Native Buildpacks under the hood.

```
$ ./gradlew bootBuildImage
```

The resulting image will be named catalog-service:0.0.1-SNAPSHOT by default (<project_name>:<version>).

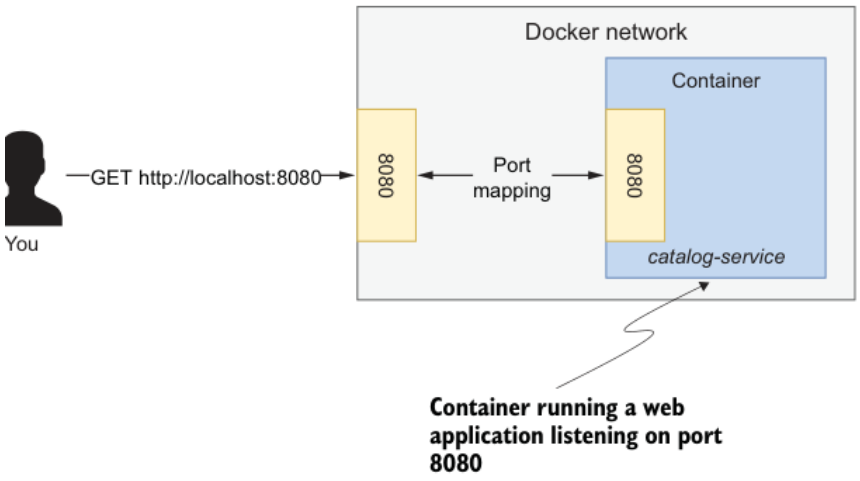
You can run the following command to get the details of the newly created image:

```
$ docker images catalog-service:0.0.1-SNAPSHOT
```

The last thing to do is run the image and verify that the containerized application is working correctly.
Open a Terminal window and run the following command:

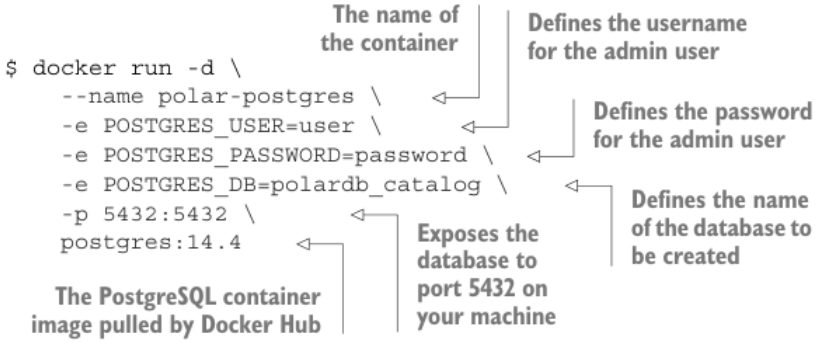
```
$ docker run --rm --name catalog-service -p 8080:8080 \
catalog-service:0.0.1-SNAPSHOT
```

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.



5.1.2 Running PostgreSQL as a container

Docker makes running databases locally easier, so I'll show you how to run PostgreSQL as a container on your local machine. Make sure your Docker Engine is up and running, open a Terminal window, and execute the following command:



Compared to how you ran the Catalog Service container, you'll notice a few new elements. First, the Docker image from which you run a container (postgres:14.4) is not created by you—it's pulled from the Docker Hub container registry (configured by default when you install Docker).
The second new thing is passing environment variables as arguments to the container. PostgreSQL accepts a few environment variables that are used during the container's creation to configure a database.

5.2.1 Connecting to a database with JDBC

These are the main dependencies:

```
dependencies {
   ...
   implementation 'org.springframework.boot:spring-boot-starter-data-jdbc'
   runtimeOnly 'org.postgresql:postgresql'
}
```

The PostgreSQL database is a backing service to the Catalog Service application. As such, it should be handled as an attached resource according to the 15-factor methodology. The attachment is done through resource binding, which in the case of PostgreSQL, consists of the following:
 A URL to define which driver to use, where to find the database server, and which database to connect the application to
 Username and password to establish a connection with the specified database

Thanks to Spring Boot, you can provide those values as configuration properties. This means you can easily replace the attached database by changing the values for the resource binding.

Open the application.yml file for the Catalog Service project, and add the properties for configuring the connection with PostgreSQL. Those values are the ones you defined earlier as environment variables when creating the PostgreSQL container.

```
spring:
   datasource:
      username: user
      password: password
      url: jdbc:postgresql:/ /localhost:5432/polardb_catalog
```

Thanks to port forwarding, the Catalog Service application in the previous chapter 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.

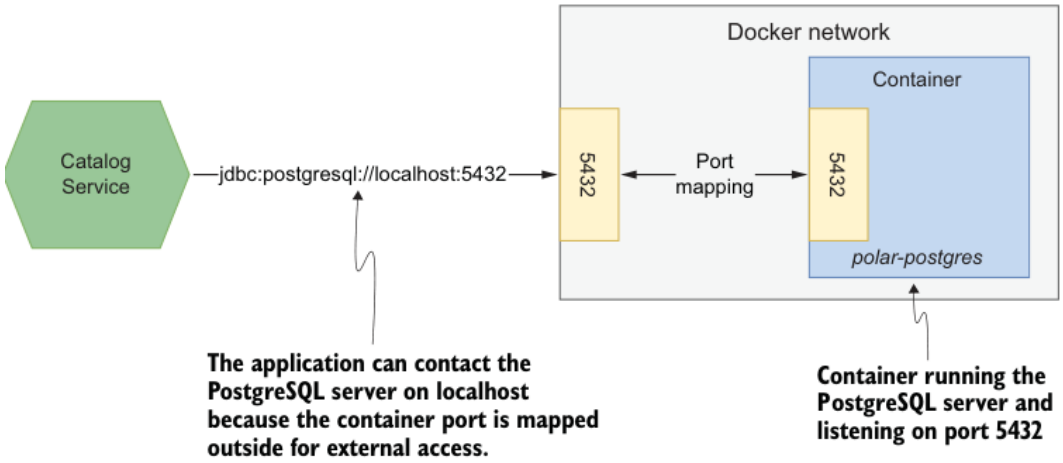


Figure 6.9 The Catalog Service application can interact with the PostgreSQL container thanks to the port mapping, making the database accessible from the outside world.

6.2 Packaging Spring Boot applications as container images

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?

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.

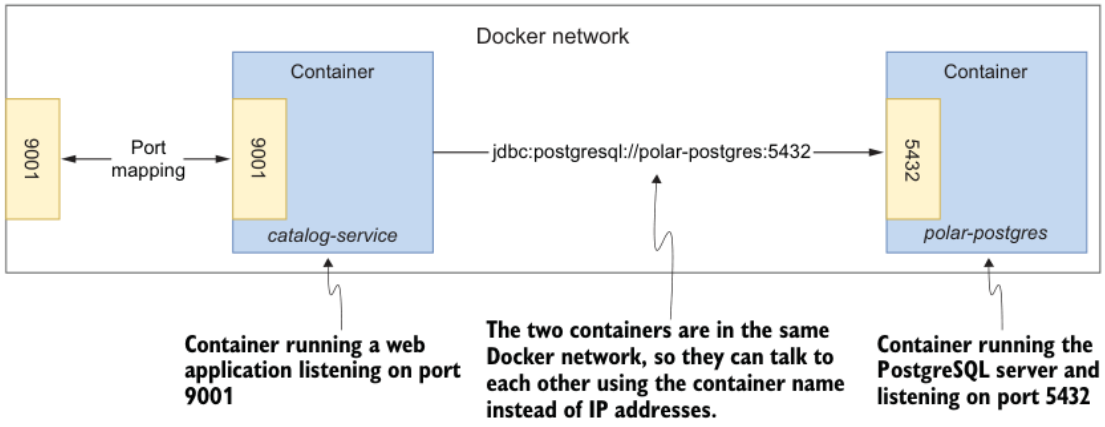


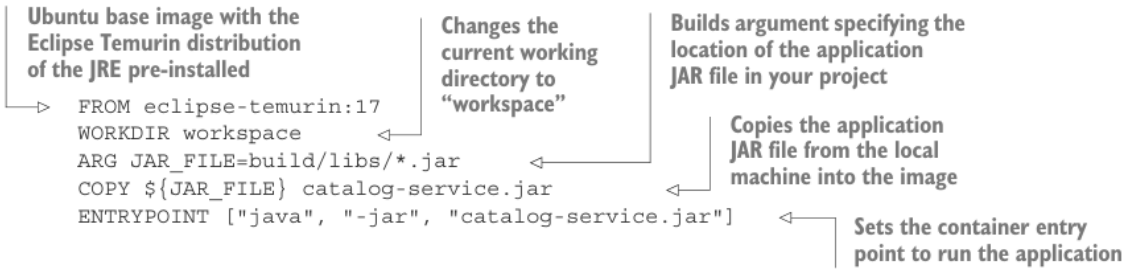
Figure 6.10 The Catalog Service container can directly interact with the PostgreSQL container because they are both on the same Docker network.

// Image

Cloud native applications are self-contained. Spring Boot lets you package your applications as standalone JARs, including everything they need to run except the runtime environment.

// Dockerfile for describing the Catalog Service image

Open your Catalog Service project (catalog-service), and create an Dockerfile in the root folder. That file will contain the recipe for containerizing your application.



This Dockerfile declares a JAR_FILE argument that can be specified when creating the image with the docker build command.

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 .
```

If you're using Maven, you can specify the location used by Maven as a build argument with the following command (don't forget the final dot):

```
$ docker build --build-arg JAR_FILE=target/*.jar -t catalog-service .
```

// run containers
Since we haven't specified any version, the image will be tagged as latest automatically.
Let's verify that it works **(*)**.

Remember the two aspects I covered in the previous section: 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.

That is still not enough. In the previous chapter, 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.

You already know how to configure a Spring Boot application from the outside without having to recompile it, right? An environment variable will do. We need to overwrite the spring.datasource.url property and specify the same URL, replacing localhost with the PostgreSQL container name: polar-postgres. Using another environment variable, we can also enable the testdata Spring profile to trigger the creation of test data in the catalog:

```
$ 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=testdata \
catalog-service
```

That's quite a long command, isn't it? You won't use the Docker CLI for long, though, I promise. Later in the chapter I'll introduce Docker Compose.

Open a Terminal window, call the application, and verify that it works correctly, as it did in chapter 5:

```
$ http :9001/books
```

When you're done, remember to delete both containers:

```
$ docker rm -f catalog-service polar-postgres
```

See => Building container images for production

(*)
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
```

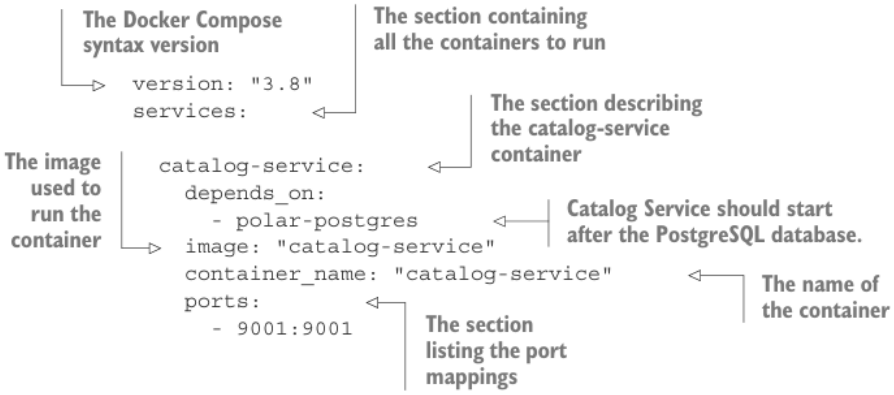
6.3 Managing Spring Boot containers with Docker Compose

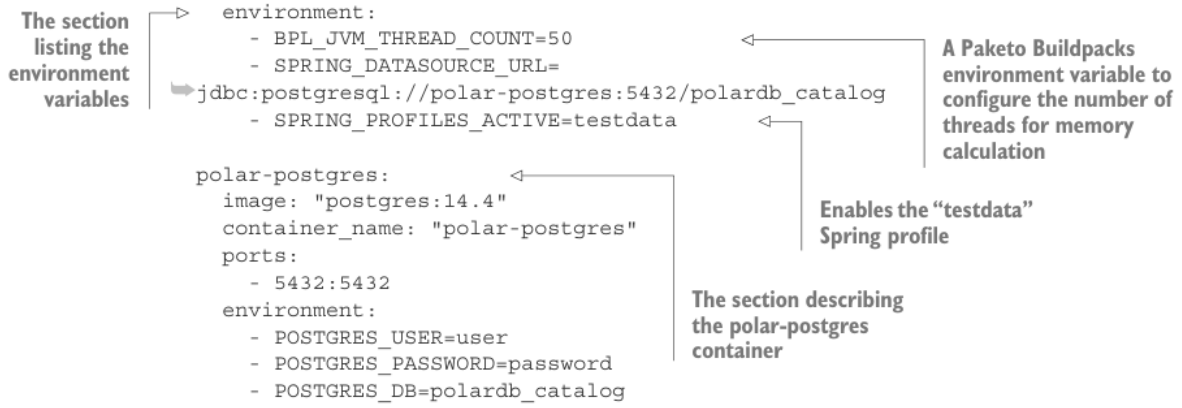
But when it comes **to running multiple containers**, the Docker CLI can be a bit cumbersome. Writing commands in a Terminal window can be error-prone, hard to read, and challenging when it comes to applying version control.

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.

You'll configure the execution of the Catalog Service and PostgreSQL containers using Docker Compose.
If you have installed **Docker Desktop** for Mac or Docker Desktop for Windows, *you already have Docker Compose installed*.
In the polar-deployment/docker folder, create a docker-compose.yml file, and define the services to run as follows.

Listing 6.7 Docker Compose file describing the catalog services





```
version: "3.8"
services:

 # Applications

 catalog-service:
  depends_on:
   - polar-postgres
  image: "catalog-service"
  container_name: "catalog-service"
  ports:
   - 9001:9001
   - 8001:8001
  environment:
   # Buildpacks environment variable to configure the number of threads in memory calculation
   - BPL_JVM_THREAD_COUNT=50
   # Buildpacks environment variable to enable debug through a socket on port 8001
   - BPL_DEBUG_ENABLED=true
   - BPL_DEBUG_PORT=8001
   - SPRING_CLOUD_CONFIG_URI=http://config-service:8888
   - SPRING_DATASOURCE_URL=jdbc:postgresql://polar-postgres:5432/polardb_catalog
   - SPRING_PROFILES_ACTIVE=testdata

 config-service:
  image: "config-service"
  container_name: "config-service"
  ports:
   - 8888:8888
   - 9888:9888
  environment:
   # Buildpacks environment variable to configure the number of threads in memory calculation
   - BPL_JVM_THREAD_COUNT=50
   # Buildpacks environment variable to enable debug through a socket on port 9888
   - BPL_DEBUG_ENABLED=true
   - BPL_DEBUG_PORT=9888

 # Backing Services

 polar-postgres:
  image: "postgres:14.12"
  container_name: "polar-postgres"
  ports:
   - 5432:5432
  environment:
   - POSTGRES_USER=user
   - POSTGRES_PASSWORD=password
   - POSTGRES_DB=polardb_catalog
```

You might have noticed the presence of an additional environment variable for the Catalog Service container. Docker Compose configures both containers on the same network by default, so you don't need to specify one explicitly, as you did previously.

Let's see now how to spin them up. Open a Terminal window, navigate to the folder containing the file, and run the following command to start the containers in detached mode:

```
$ docker-compose up -d
```

When the command is done, try calling the Catalog Service application at http://localhost:9001/books and verify that it works correctly.