# Docker Compose

## Overview

We've implemented a Spring Boot application that currently talks to a MongoDB database on the host computer. You will containerize the system as follows:

* You'll run MongoDB in a container, rather than running it directly on the host computer.
* You'll run the Spring Boot application in another container. You'll tweak the application so that it talks to the containerized MongoDB database.

## Getting started

Start 'Play with Docker' and run the following command to download the lab code:

wget http://olsensoft.com/DockerOpenShift/DoshDev.zip -O temp.zip; \

unzip temp.zip; \

rm temp.zip

## Source folders for this lab

* DoshDev/Student/04-DockerCompose
* DoshDev/Solutions/04-DockerCompose

## Roadmap

There are 6 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Familiarization
2. Defining a Dockerfile for a custom MongoDB image
3. Building the custom MongoDB image and running a container
4. Defining a Dockerfile for the Spring Boot application
5. Building the Spring Boot app image and running a container
6. (If time permits): Using Docker Compose

## Exercise 1: Familiarization

In the 'Play with Docker' environment, go to the following directory:

DoshDev/Student/04-DockerCompose

Now go to the employee-spring-boot-mongodb subdirectory, where we've implemented a complete Spring Boot console application that reads/writes data in an "employee" collection in a MongoDB database. Note the following points:

* The POM file specifies the spring-boot-starter-data-mongodb Maven dependency, so the application can talk to a MongoDB database. No surprise there.
* Go to the src/main/java/com/example/employeespringbootmongodb directory and take a quick look at the 4 Java files:
  + The Employee class represents a document in the "employee" collection in the MongoDB database.

The employeeId field is annotated with @Id, so Spring Boot knows it corresponds to the \_id field in a document in the database. The field is a BigInteger, which is the preferred datatype for MongoDB IDs (remember a MongoDB ID is a big number, generated by MongoDB upon insertion).

* + The EmployeeRepository interface is a simple Spring Boot Data Repository with a few query methods to get data out of the database. For a reminder about how this works, see the Spring Boot course (specifically the Spring Data chapter, in the Annex section at the end of the chapter).
  + The EmployeeService class uses the EmployeeRepository to perform various data-manipulation operations. This is similar to code you saw in the Spring Boot course.
  + The Application class is a regular Spring Boot application class. It simply invokes functionality provided by the EmployeeService bean.
* Go to the src/main/resources directory and review application.properties. We've set the spring.data.mongodb.uri property to an environment variable named EMPLOYEE\_MONGODB\_URI, which gives us the flexibility to point to MongoDB at a specified URL. We've provided a default of mongodb://localhost:27017/test, just in case the environment variable isn't set.
* Go to the target directory, which contains the application packaged as a JAR named employee-spring-boot-mongodb-0.0.1-SNAPSHOT.jar. We generated this JAR by running mvn package. You'll containerize this JAR in Docker image shortly.

**Exercise 2:** **Defining a Dockerfile for a custom MongoDB image**

Recall the purpose of this lab is to run MongoDB in one container, to run the Spring Boot app in another container, and to link them together so the Spring Boot app can talk to the containerized MongoDB database.

In this exercise you'll focus on how to run MongoDB in a container. In the previous lab you ran an instance of the off-the-shelf MongoDB image as follows (**don't do this now!**):

docker run -d -p 27017:27017 --name mongodb mongo:4.2.9

This is fine in simple cases, but in most real-world scenarios you need to build a custom Docker image with additional bells and whistles (e.g. set the username/password, run an initialization script to populate the database, etc.). In other words, you need to write a Dockerfile.

We've provided a skeletal Dockerfile named Dockerfile-mongodb to get you started. Open this file now, using the Editor in the 'Play with Docker' environment. Complete the Dockerfile as follows:

* At the start of the file, add a FROM instruction to specify the base image for your custom image. The base image you want is mongo:4.2.9.
* Add an EXPOSE instruction to "expose" port 27017. MongoDB will run on this port inside the container, and the EXPOSE instruction reminds anyone using your image that they need to map this port to an actual port on the host computer.
* Add a COPY instruction to copy an initialization file named mongodbsetup.js into the /docker-entrypoint-initdb.d/ directory inside the container. This is a special directory in the container - when you run the container, Docker will automatically run any scripts it sees in this directory to initialize the database. You can have lots of files in this directory if you like - Docker will run all of them, in alphabetic order.
* Add a RUN instruction to run the Linux command chmod 777, to enable all permissions on the file /docker-entrypoint-initdb.d/mongodbsetup.js file. This is necessary to allow the JS file to be executed.

Save the file and return to the 'Play with Docker' shell. Then open mongodbsetup.js in the editor, and edit the JS code to insert several documents into the employee collection. Docker will run this initialization script when a container starts, to populate the MongoDB database.

**Exercise 3: Building the custom MongoDB image and running a container**

Run the following command to build the Docker image based on your Dockerfile:

docker build -f Dockerfile-mongodb -t mymongodb .

Then run the following command to run a container from this image (this creates a container named mymongodb1):

docker run -d -p 27017:27017 --name mymongodb1 mymongodb

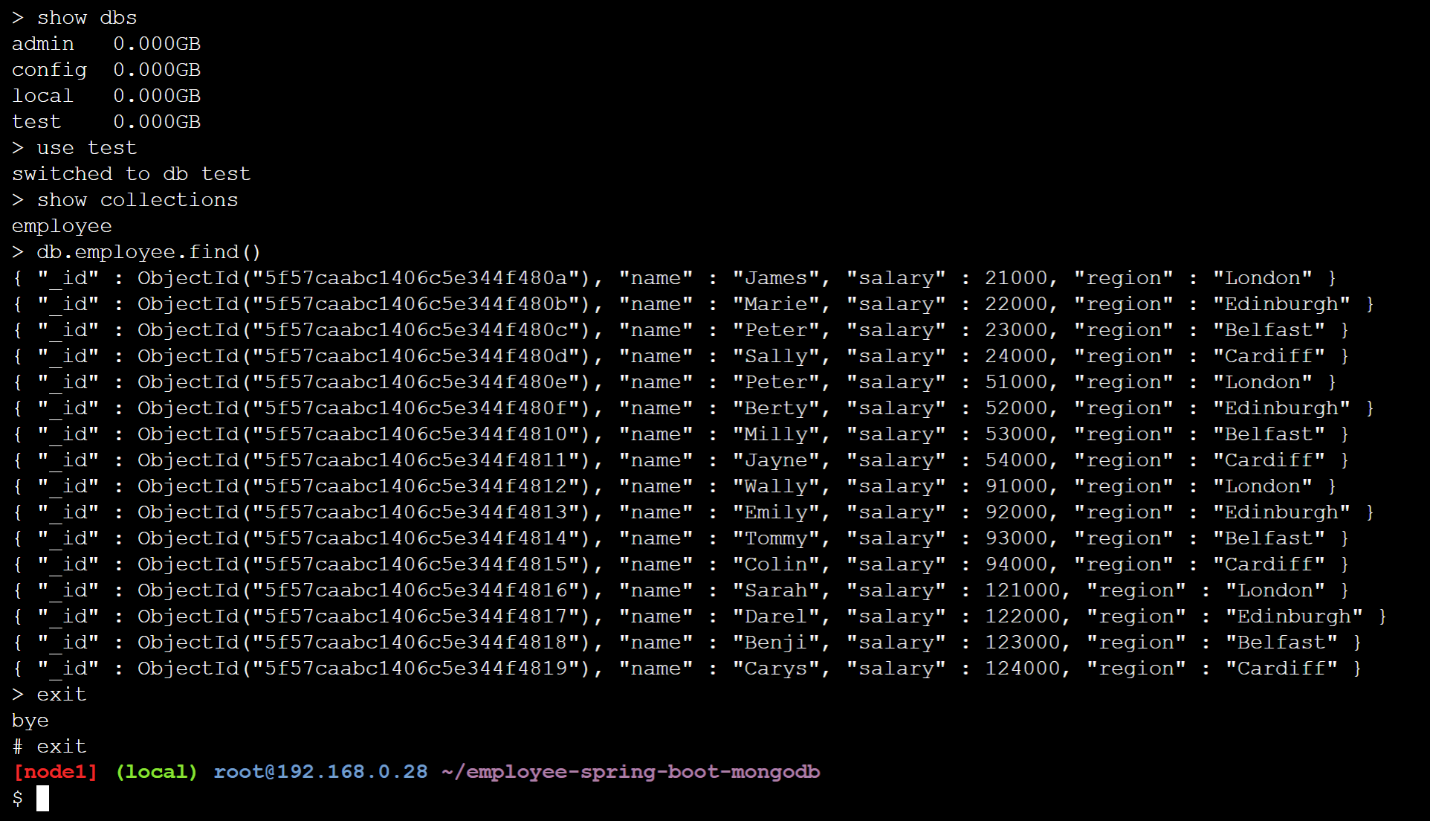
Double-check the container is running successfully:

docker container ls -a

Now exec into the container to run a Bash shell inside it:

docker container exec -it mymongodb1 /bin/sh

When you’re inside the container, run the mongo command to open a MongoDB shell. Inside the MongoDB shell, run some MongoDB commands to view the database structure and data. For example:



**Exercise 4: Defining a Dockerfile for the Spring Boot application**

You will now turn your attention to defining a Dockerfile to containerize the Spring Boot application. We've provided a skeletal Dockerfile named Dockerfile-app to get you started. Open this file now using the Editor in the 'Play with Docker' environment, and complete the Dockerfile as follows:

* At the start of the file, add a FROM instruction to specify the base image for your custom image. The base image you want is openjdk:11.0.
* Add a COPY (or ADD) instruction to copy the Spring Boot application JAR file into the image.
* Add an ENV instruction to set the EMPLOYEE\_MONGODB\_URI environment variable to point to mongodb://mymongodb1:27017/test (i.e. the MongoDB database running in the mymongodb1 container). The Spring Boot application uses this environment variable to set the spring.data.mongodb.uri application property, so it knows how to connect to the MongoDB database.
* Add an ENTRYPOINT instruction to run the Java application in the image.

**Exercise 5: Building the Spring Boot app image and running a container**

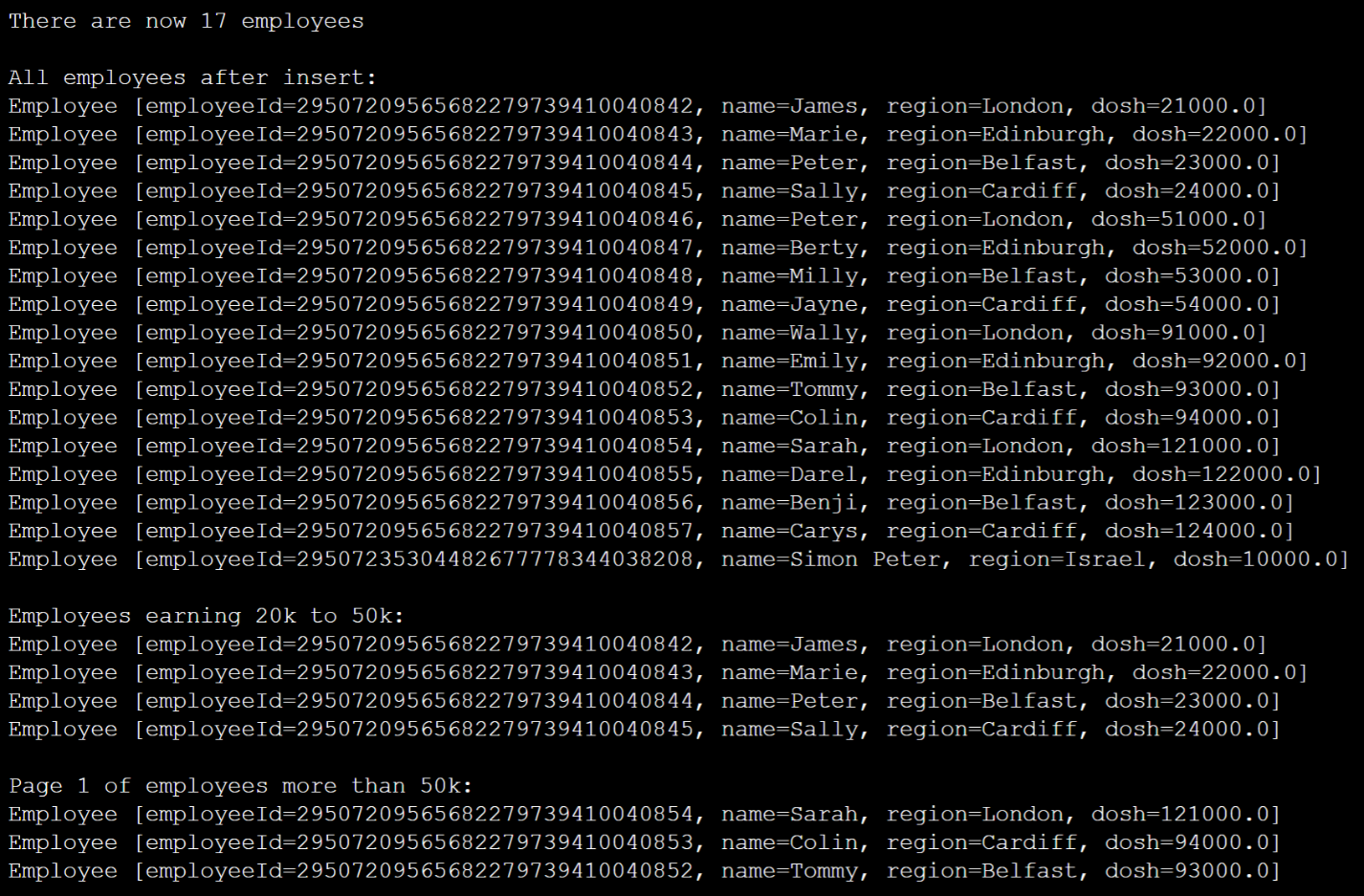
Run the following command to build the Docker image based on your Dockerfile:

docker build -f Dockerfile-app -t myapp .

Then run the following command to run a container from this image (this creates a container named myapp1, and links it to the mymongodb1 container so it can access the containerized MongoDB database):

docker run --name myapp1 --link mymongodb1:mymongodb1 myapp

The containerized Spring Boot app should successfully execute queries and updates against the MongoDB database



**Exercise 6 (If time permits): Using Docker Compose**

In Exercises 3 and 5 above, you manually executed Docker commands to build images and run containers. Having to tip-toe through these manual commands is not an easy task.

To simplify matters, use Docker Compose to automate the building of images and the running of containers. We've provided a skeletal docker-compose.yaml file to get you started. Refer back to the chapter notes if you need a reminder of how to write and run a Docker Compose file.