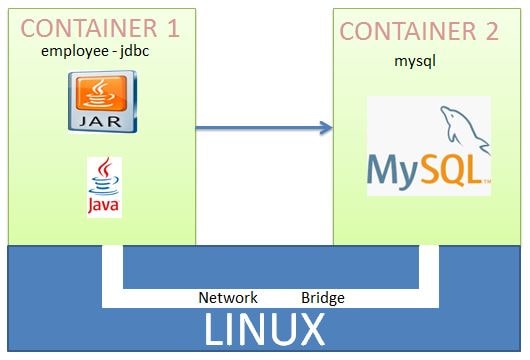
**Deploy Spring Boot + MYSQL Application to Docker**

In this tutorial we will be deploying Spring Boot + MYSQL application to docker. In a [previous docker tutorial](https://www.javainuse.com/devOps/docker/docker-networking) we saw how to deploy multiple Spring Boot Microservices to Docker Container using docker networking. Previously we had seen how to [deploy applications using docker compose.](https://www.javainuse.com/devOps/docker/docker-compose-tutorial) We will first be deploying this application to docker without compose. Later we will also see how to achieve the same using Docker Compose.  


For deploying Spring Boot + MYSQL application to docker, we will need to consider Spring Boot Application and MYSQL as two different services. As we had seen in previous tutorial we run a single service on a single Docker Container. So for Spring Boot Application and MYSQL to be deployed we will need two docker containers. And also these will then need to be running on same network so that they can communicate with each other. In this tutorial we will first be creating a Spring Boot + MYSQL application and running it locally. We will then see how this application will be deployed on docker.

Docker Tutorial - Table Of Contents

[Docker](https://www.javainuse.com/devOps/docker)

[Deploying Spring Based WAR Application to Docker](https://www.javainuse.com/devOps/docker/docker-war)

[Deploying Spring Based JAR Application to Docker](https://www.javainuse.com/devOps/docker/docker-jar)

[Deploying Multiple Spring Boot Microservices using Docker Networking](https://www.javainuse.com/devOps/docker/docker-networking)

[Deploying Multiple Spring Boot Microservices using Docker Compose](https://www.javainuse.com/devOps/docker/docker-compose-tutorial)

[**Deploying Spring Boot + MYSQL Application to Docker**](https://www.javainuse.com/devOps/docker/docker-mysql)

[Publishing Docker Image to DockerHub](https://www.javainuse.com/devOps/docker/docker-hub)

[Deploy Docker Swarm services using Play With Docker](https://www.javainuse.com/devOps/docker/docker-swarm1)

[Deploy Docker Swarm services using Docker Stack](https://www.javainuse.com/devOps/docker/docker-swarm2)

[Deploy Docker Swarm services to multiple AWS EC2 instances](https://www.javainuse.com/devOps/docker/docker-aws)

[Docker Cheat Sheet](https://www.javainuse.com/devOps/docker/docker-commands-cheat-sheet)

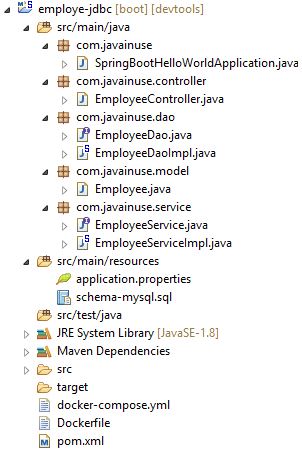
Video

This tutorial is explained in the below Youtube Video.

Lets Begin-

In a previous tutorial we had seen the basics of [Spring Boot + JDBC](https://www.javainuse.com/spring/bootjdbc) . In this tutorial we will be developing an application named employee-jdbc. We will be exposing two REST API's. One for inserting employee details and other for getting list of employees.

Create Spring Boot + MYSQL Application

The Spring Boot + MYSQL application we will be developing is as follows-  
  
The pom.xml will be as follows-

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.javainuse</groupId>

<artifactId>employee-jdbc</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>employee-jdbc</name>

<description>Demo project for Spring Boot</description>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.1.RELEASE</version>

<relativePath /> <!-- lookup parent from repository -->

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-jdbc</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>mysql</groupId>**

**<artifactId>mysql-connector-java</artifactId>**

**<scope>runtime</scope>**

**</dependency>**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

In the application.properties file specify the datasource properties

spring.datasource.url=jdbc:mysql://localhost/bootdb?createDatabaseIfNotExist=true&autoReconnect=true&useSSL=false

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.platform=mysql

spring.datasource.initialization-mode=always

Create the schema-mysql.sql file and specify the initialization scripts-

CREATE TABLE IF NOT EXISTS employee (

empId VARCHAR(10) NOT NULL,

empName VARCHAR(100) NOT NULL

);

Create the Employee Domain class

package com.javainuse.model;

public class Employee {

private String empId;

private String empName;

public String getEmpId() {

return empId;

}

public void setEmpId(String empId) {

this.empId = empId;

}

public String getEmpName() {

return empName;

}

public void setEmpName(String empName) {

this.empName = empName;

}

@Override

public String toString() {

return "Employee [empId=" + empId + ", empName=" + empName + "]";

}

}

Create the DAO interface where we specify the database operation methods to be performed.

package com.javainuse.dao;

import java.util.List;

import com.javainuse.model.Employee;

public interface EmployeeDao {

List<Employee> getAllEmployees();

void insertEmployee(Employee employee);

}

The DAO implementation class. We have autowired the DataSource bean here. The DataSource bean uses the properties defined in the application.properties.

package com.javainuse.dao;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

import javax.annotation.PostConstruct;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.support.JdbcDaoSupport;

import org.springframework.stereotype.Repository;

import com.javainuse.dao.EmployeeDao;

import com.javainuse.model.Employee;

@Repository

public class EmployeeDaoImpl extends JdbcDaoSupport implements EmployeeDao {

@Autowired

DataSource dataSource;

@PostConstruct

private void initialize() {

setDataSource(dataSource);

}

@Override

public List<Employee> getAllEmployees() {

String sql = "SELECT \* FROM employee";

List<Map<String, Object>> rows = getJdbcTemplate().queryForList(sql);

List<Employee> result = new ArrayList<Employee>();

for (Map<String, Object> row : rows) {

Employee emp = new Employee();

emp.setEmpId((String) row.get("empId"));

emp.setEmpName((String) row.get("empName"));

result.add(emp);

}

return result;

}

@Override

public void insertEmployee(Employee employee) {

String sql = "INSERT INTO employee " + "(empId, empName) VALUES (?, ?)";

getJdbcTemplate().update(sql, new Object[] { employee.getEmpId(), employee.getEmpName() });

}

}

Create Service interface to specify employee operations to be performed.

package com.javainuse.service;

import java.util.List;

import com.javainuse.model.Employee;

public interface EmployeeService {

List<Employee> getAllEmployees();

void insertEmployee(Employee employee);

}

The Service class implementation. Here we autowire the EmployeeDao bean.

package com.javainuse.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.javainuse.dao.EmployeeDao;

import com.javainuse.model.Employee;

import com.javainuse.service.EmployeeService;

@Service

public class EmployeeServiceImpl implements EmployeeService {

@Autowired

EmployeeDao employeeDao;

public List<Employee> getAllEmployees() {

List<Employee> employees = employeeDao.getAllEmployees();

return employees;

}

@Override

public void insertEmployee(Employee employee) {

employeeDao.insertEmployee(employee);

}

}

Create the controller class to expose two REST API's. One is a GET API to return the list of employees from the database. The other is a POST API which takes the employee object and persists it in the database.

package com.javainuse.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RestController;

import com.javainuse.model.Employee;

import com.javainuse.service.EmployeeService;

@RestController

public class EmployeeController {

@Autowired

EmployeeService empService;

@RequestMapping(value = "/employees", method = RequestMethod.GET)

public List<Employee> getEmployees() {

return empService.getAllEmployees();

}

@RequestMapping(value = "/insertemployee", method = RequestMethod.POST)

public void insertEmployee(@RequestBody Employee employee) {

empService.insertEmployee(employee);

}

}

Finally create the class with @SpringBootApplication annotation.

package com.javainuse;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.RestController;

@RestController

@EnableAutoConfiguration

@SpringBootApplication

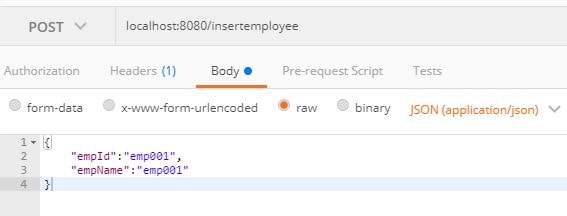
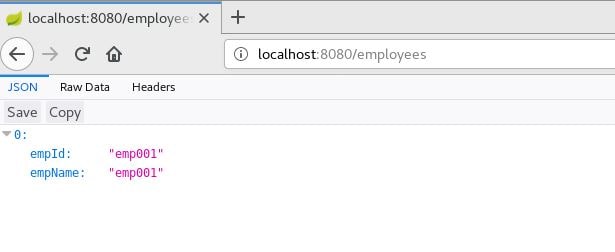
public class SpringBootHelloWorldApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootHelloWorldApplication.class, args);

}

}

Start the Spring Boot Application.  
Open POSTMAN and create a POST request to url - localhost:8080/insertemployee with employee object to be persisted in DB.  
  
In the browser use localhost:8080/employees to fetch the list of employees.  


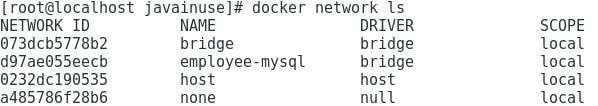
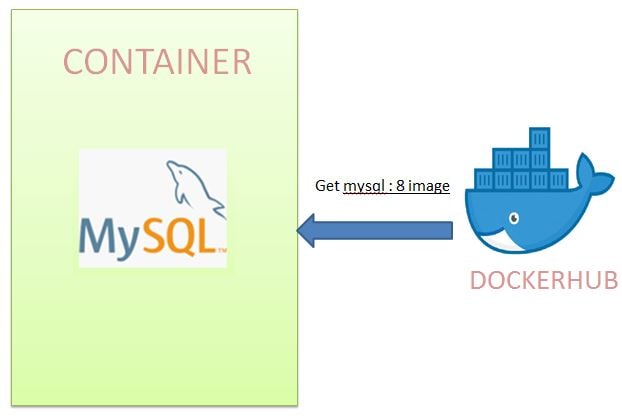
Deploying Spring Boot + MYSQL to Docker-

Since are going to create two docker containers which should communicate with each other, we will need to start them on same network. We had seen the Docker Networking details in a previous tutorial. Open the terminal and start the docker

systemctl start docker

First lets create a network named employee-mysql

docker network create employee-mysql

spring boot mysql docker tutorial  
  
  
MYSQL provides an image in dockerhub which we can run as container.  


We will use the image provided by dockerhub to run as container. Also we will specify following when running the container

* a. name of the mysql container
* a. What should be the password for MYSQL
* b. We want to create the Database named bootdb.
* c. specify the network employee-mysql on which this container should be created.
* d. start the container in detached mode.

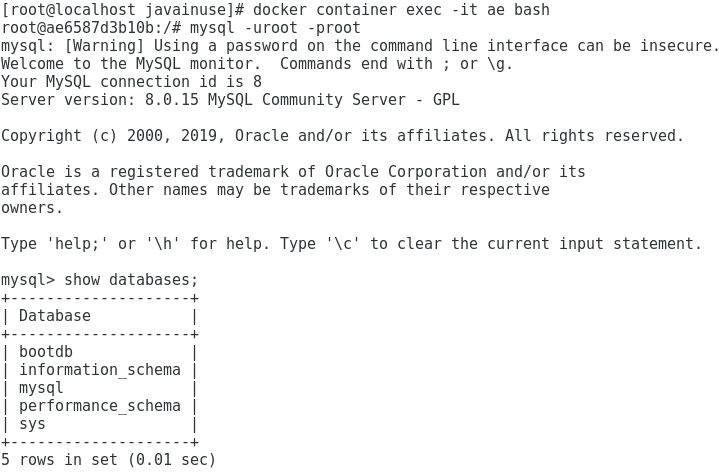
docker container run --name mysqldb --network employee-mysql -e MYSQL\_ROOT\_PASSWORD=root -e MYSQL\_DATABASE=bootdb -d mysql:8

spring boot mysql docker start container  
Next let us check if container has started correctly using logs command.

docker container logs -f ae

  
Using the exec command we can also inspect if the database named bootdb is created successfully.

docker container exec -it ae bash



Next we will modify the application.properties in the Spring Boot application to make use of the mysql container name i.e.mysqldb instead of localhost

spring.datasource.url=jdbc:mysql://**mysqldb**/bootdb

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.platform=mysql

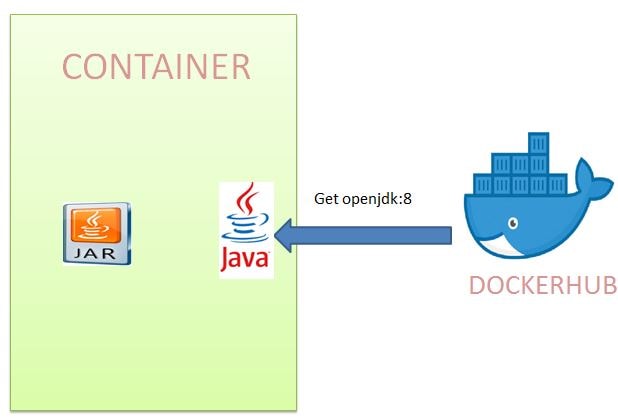
spring.datasource.initialization-mode=always

The docker file for spring boot project will be as follows-

From openjdk:8

copy ./target/employee-jdbc-0.0.1-SNAPSHOT.jar employee-jdbc-0.0.1-SNAPSHOT.jar

CMD ["java","-jar","employee-jdbc-0.0.1-SNAPSHOT.jar"]

  
Build the docker image for the spring boot project

docker image build -t employee-jdbc .



Next run this as a container. Also we are running the container on the employee-mysql network.

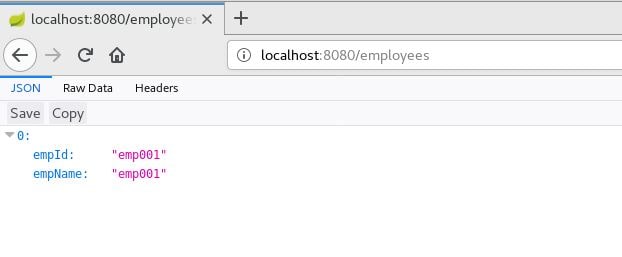
docker container run --network employee-mysql --name employee-jdbc-container -p 8080:8080 -d employee-jdbc

spring boot mysql docker container image run

docker container logs -f 34

  
Both our containers have started successfully. Let us insert data with POST request using curl-

curl --header "Content-Type: application/json" --request POST --data '{"empId":"emp001","empName":"emp001"}' http://localhost:8080/insertemployee

spring boot mysql docker container image run curl  
Finally if we go to localhost:8080/employees we get the inserted records.  


In previous tutorial we had seen [what is docker compose and its advantages.](https://www.javainuse.com/devOps/docker/docker-compose-tutorial) To achieve the above using docker-compose create the docker-compose.yml as follows-

version: "3"

services:

employee-jdbc:

image: employee-jdbc

ports:

- "8080:8080"

networks:

- employee-mysql

depends\_on:

- mysqldb

mysqldb:

image: mysql:8

networks:

- employee-mysql

environment:

- MYSQL\_ROOT\_PASSWORD=root

- MYSQL\_DATABASE=bootdb

networks:

employee-mysql:

Use docker-compose up to start docker compose.  
