Spring Boot PostgreSQL tutorial

Spring Boot PostgreSQL tutorial shows how to use PostgreSQL database in a Spring Boot application.

Tweet

Spring is a popular Java application framework for creating enterprise applications. *Spring Boot* is an evolution of Spring framework which helps create stand-alone, production-grade Spring based applications with minimal effort.

PostgreSQL

PostgreSQL is a powerful, open source object-relational database system. It is a multi-user database management system. It runs on multiple platforms including Linux, FreeBSD, Solaris, Microsoft Windows, and Mac OS X. PostgreSQL is developed by the PostgreSQL Global Development Group.

PostgreSQL setup

We are going to show how to install PostgreSQL database on a Debian Linux system.

```
$ sudo apt-get install postgresql
```

This command installs PostgreSQL server and related packages.

```
$ /etc/init.d/postgresql status
```

We check the status of the database with `postgresql status` command.

```
$ sudo -u postgres psql postgres
psql (9.5.10)
Type "help" for help.

postgres=# \password postgres
Enter new password:
Enter it again:
```

After the installation, a `postgres` user with administration priviliges was created with empty

default password. As the first step, we need to set a password for `postgres`.

```
$ sudo -u postgres createuser --interactive --password user12
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles? (y/n) n
Password:
```

We create a new database user.

```
$ sudo -u postgres createdb testdb -O user12
```

We create a new `testdb` database with `createdb` command, which is going to be owned by `user12`.

```
$ sudo vi /etc/postgresql/9.5/main/pg_hba.conf
```

We edit the `pg_hba.conf` file.

```
# "local" is for Unix domain socket connections only
local   all             all                                     trust
# IPv4 local connections:
host    all             all             127.0.0.1/32            trust
```

In order to be able to run a Spring Boot application with a local PostgreSQL installation, we change the authentication method for the Unix domain socket and local connections to `trust`.

```
$ sudo service postgresql restart
```

We restart PostgreSQL to enable the changes.

```
$ psql -U user12 -d testdb -W
Password for user user12:
psql (9.5.10)
Type "help" for help.

testdb=>
```

Now we can use the `psql` tool to connect to the database.

Spring Boot PostgreSQL example

The following application is a simple Spring Boot web application, which uses PostgreSQL database. We have a home page with a link to display data from a database table. We use Thymeleaf templating system to join data with HTML.

```
pom.xml
src
├───main
│   ├───java
│   │   └───com
│   │       └───zetcode
│   │           │   Application.java
│   │           ├───controller
```

```
│   │           │       MyController.java
│   │           ├───model
│   │           │       City.java
│   │           ├───repository
│   │           │       CityRepository.java
│   │           └───service
│   │                   CityService.java
│   │                   ICityService.java
│   └───resources
│       │   application.properties
│       │   data-postgres.sql
│       │   schema-postgres.sql
│       ├───static
│       │       index.html
│       └───templates
│               showCities.html
└───test
    └───java
```

This is the project structure.

pom.xml

```
<?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.zetcode</groupId>
    <artifactId>springbootpostgreex</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.7.RELEASE</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <scope>runtime</scope>
        </dependency>
```

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Spring Boot starters are a set of convenient dependency descriptors which greatly simplify Maven configuration. The `spring-boot-starter-parent` has some common configurations for a Spring Boot application. The `spring-boot-starter-web` is a starter for building web, including RESTful, applications using Spring MVC. It uses Tomcat as the default embedded container. The `spring-boot-starter-thymeleaf` is a starter for building MVC web applications using Thymeleaf views. The `spring-boot-starter-data-jpa` is a starter for using Spring Data JPA with Hibernate.

The `postgresql` dependency is for the PostgreSQL database driver.

The `spring-boot-maven-plugin` provides Spring Boot support in Maven, allowing us to package executable JAR or WAR archives. Its `spring-boot:run` goal runs the Spring Boot application.

resources/application.properties

```
spring.main.banner-mode=off
logging.level.org.springframework=ERROR

spring.jpa.hibernate.ddl-auto=none

spring.datasource.initialization-mode=always
spring.datasource.platform=postgres
spring.datasource.url=jdbc:postgresql://localhost:5432/testdb
spring.datasource.username=postgres
spring.datasource.password=s$cret

spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
```

In the `application.properties` file we write various configuration settings of a Spring Boot application.

With the `spring.main.banner-mode` property we turn off the Spring banner. To load a database that is not embedded, in Spring Boot 2 we need to add `spring.datasource.initialization-mode=always`. To avoid conflicts, we turn off automatic schema creation with `spring.jpa.hibernate.ddl-auto=none`.

In the spring datasource properties we set up the PostgreSQL datasource.

The `spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation` option is set avoid a recent issue. Without this option, we get the following error:

```
java.sql.SQLFeatureNotSupportedException: Method org.postgresql.jdbc.PgConnection.createClob()
is not yet implemented.
```

com/zetcode/model/City.java

```
package com.zetcode.model;

import java.util.Objects;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "cities")
public class City {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;
    private int population;

    public City() {
    }

    public City(Long id, String name, int population) {

        this.id = id;
        this.name = name;
        this.population = population;
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public int getPopulation() {
        return population;
    }

    public void setPopulation(int population) {
        this.population = population;
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 79 * hash + Objects.hashCode(this.id);
        hash = 79 * hash + Objects.hashCode(this.name);
        hash = 79 * hash + this.population;
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final City other = (City) obj;
        if (this.population != other.population) {
            return false;
        }
        if (!Objects.equals(this.name, other.name)) {
            return false;
        }
        return Objects.equals(this.id, other.id);
    }

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("City{");
        sb.append("id=").append(id);
        sb.append(", name='").append(name).append('\'');
        sb.append(", population=").append(population);
        sb.append('}');
        return sb.toString();
    }
}
```

This is the `City` entity. Each entity must have at least two annotations defined: `@Entity` and `@Id`.

```
@Entity
@Table(name = "cities")
public class City {
```

The `@Entity` annotation specifies that the class is an entity and is mapped to a database table. The `@Table` annotation specifies the name of the database table to be used for mapping.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

The `@Id` annotation specifies the primary key of an entity and the `@GeneratedValue` provides for the specification of generation strategies for the values of primary keys.

resources/schema-postgres.sql

```
DROP TABLE IF EXISTS cities;
CREATE TABLE cities(id serial PRIMARY KEY, name VARCHAR(255), population integer);
```

When the application is started, the `schema-postgres.sql` script is executed provided that the automatic schema creation is turned off. The script creates a new database table.

resources/data-postgres.sql

```
INSERT INTO cities(name, population) VALUES('Bratislava', 432000);
INSERT INTO cities(name, population) VALUES('Budapest', 1759000);
INSERT INTO cities(name, population) VALUES('Prague', 1280000);
INSERT INTO cities(name, population) VALUES('Warsaw', 1748000);
INSERT INTO cities(name, population) VALUES('Los Angeles', 3971000);
INSERT INTO cities(name, population) VALUES('New York', 8550000);
INSERT INTO cities(name, population) VALUES('Edinburgh', 464000);
INSERT INTO cities(name, population) VALUES('Berlin', 3671000);
```

Later, the `data-postgres.sql` file is executed to fill the table with data.

com/zetcode/repository/CityRepository.java

```
package com.zetcode.repository;

import com.zetcode.model.City;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CityRepository extends CrudRepository<City, Long> {

}
```

By extending from the Spring `CrudRepository`, we will have some methods for our data repository implemented, including `findAll()`. This way we save a lot of boilerplate code.

com/zetcode/service/ICityService.java

```
package com.zetcode.service;
```

```
import com.zetcode.model.City;

import java.util.List;

public interface ICityService {

    List<City> findAll();
}
```

`ICityService` provides a contract method to get all cities from the data source.

com/zetcode/service/CityService.java

```
package com.zetcode.service;

import com.zetcode.model.City;
import com.zetcode.repository.CityRepository;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class CityService implements ICityService {

    @Autowired
    private CityRepository repository;

    @Override
    public List<City> findAll() {

        var cities = (List<City>) repository.findAll();

        return cities;
    }
}
```

`CityService` contains the implementation of the `findAll()` method. We use the repository to retrieve data from the database.

```
@Autowired
private CityRepository repository;
```

`CityRepository` is injected.

```
var cities = (List<City>) repository.findAll();
```

The `findAll()` method of the repository returns the list of cities.

com/zetcode/MyController.java

```
package com.zetcode.controller;
```

```
import com.zetcode.model.City;
import com.zetcode.service.ICityService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

import java.util.List;

@Controller
public class MyController {

    @Autowired
    private ICityService cityService;

    @GetMapping("/showCities")
    public String findCities(Model model) {

        var cities = (List<City>) cityService.findAll();

        model.addAttribute("cities", cities);

        return "showCities";
    }
}
```

`MyController` contains one mapping.

```
@Autowired
private ICityService cityService;
```

We inject a `ICityService` into the `countryService` field.

```
@GetMapping("/showCities")
public String findCities(Model model) {

    var cities = (List<City>) cityService.findAll();

    model.addAttribute("cities", cities);

    return "showCities";
}
```

We map a request with the `showCities` path to the controller's `findCities()` method. The default request is a GET request. The model gains a list of cities and the processing is sent to the `showCities.html` Thymeleaf template file.

resources/templates/showCities.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Cities</title>
```

```
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>

    <body>
        <h2>List of cities</h2>

        <table>
            <tr>
                <th>Id</th>
                <th>Name</th>
                <th>Population</th>
            </tr>

            <tr th:each="city : ${cities}">
                <td th:text="${city.id}">Id</td>
                <td th:text="${city.name}">Name</td>
                <td th:text="${city.population}">Population</td>
            </tr>
        </table>

    </body>
</html>
```

In the `showCities.html` template file, we display the data in an HTML table.

resources/static/index.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>Home page</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <a href="showCities">Show cities</a>
    </body>
</html>
```

In the `index.html` there is a link to show all cities.

com/zetcode/Application.java

```
package com.zetcode;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
```

```
}
```

The `Application` sets up the Spring Boot application. The `@SpringBootApplication` enables auto-configuration and component scanning.

```
$ mvn spring-boot:run
```

After the application is run, we can navigate to `localhost:8080`.

In this tutorial, we have showed how to use PostgreSQL database in a Spring Boot application. You might also be interested in the related tutorials:

- Spring Boot MySQL tutorial
- Spring Boot H2 REST tutorial
- Spring Boot Thymeleaf tutorial
- Introduction to Spring web applications
- Spring Boot RESTFul application

Find out all Spring Boot tutorials.