Multiple Data Sources with Spring Boot

August 15, 2019 by Java Development Journal

In this post, we will explore the configurations to set up ***multiple data sources with Spring Boot.*** We will use Spring Data JPA system with **multiple databases.**

Introduction

Spring Boot provides first-class support to the *Spring JPA* that makes it easy to access the database with little boilerplate code by using *Spring Repositories* feature. ***Spring Boot*** does not provide an out of the box solution in case our application needs multiple *DataSources* (e.g. multi-tenant system).In this article, we will explore the steps for setting up multiple data sources with Spring Boot using Spring JPA.

1. Maven Setup

To set up our *Spring Boot project*, we need to add `spring-boot-starter-data-jpa` dependency to the `<em>pom.xml</em>` file.

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

```
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

The spring boot starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported a set of managed transitive dependencies. Read our article Spring Boot Starters on for more detail on the starters.

2. DataSource Configurations

Spring Boot provides a way to configure our project properties using `<em>application.properties</em>` or yml file. In this post, we will use `<em>application.properties</em>` file. To use *multiple DataSources*, let's add the following details in the property file.

```
spring.jpa.generate-ddl=true

spring.datasource.url = jdbc:mysql://localhost:3306/db1
spring.datasource.username = [username]
spring.datasource.password = [password]
spring.datasource.driverClassName = com.mysql.jdbc.Driver

#second db2 ...
db2.datasource.url = jdbc:mysql://localhost:3306/db2
```

Let's keep in mind the following important points

- Please create 2 different databases in your *MySql Database*.

[pullquote align="normal"]The dialect to use is also automatically detected based on the current DataSource, but you can set `<em>spring.jpa.database</em>` yourself if you want to be explicit and bypass that check on startup. [/pullquote]

3. JPA Entities

Let's define the following 2 JPA entities for our post.

1. Product
2. Customer

3.1 Product Entity

```
@Entity
public class ProductModel {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private int id;

 @Column(nullable = false)
 private String code;
 private String name;
 private double price;


 protected ProductModel() {}
```

```
 }

 @Override
 public String toString() {
  return String.format(
   "Product[id=%d, code='%s', name='%s', price='%s']",
   id, code, name, price);
 }

 public int getId() {
  return id;
 }

 public String getCode() {
  return code;
 }

 public String getName() {
  return name;
 }

 public double getPrice() {
  return price;
 }
}
```

3.2 Customer Entity

```
package com.javadevjournal.customer.data;

import javax.persistence.*;

@Entity
public class CustomerModel {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Integer id;
 @Column(unique = true, nullable = false)
 private String email;
```

```
 public CustomerModel(String email, String firstName, String lastNa
  this.email = email;
  this.firstName = firstName;
  this.lastName = lastName;
 }

 @Override
 public String toString() {
  return String.format(
    "Customer[id=%d, firstName='%s', lastName='%s',email='%s']",
    id, firstName, lastName, email);
 }

 public Integer getId() {
  return id;
 }

 public String getEmail() {
  return email;
 }

 public String getFirstName() {
  return firstName;
 }

 public String getLastName() {
  return lastName;
 }
}
```

[pullquote align="normal"]We added both **entities in different packages**. This is important and we will explain this in the next section. [/pullquote]

4. Package Structure

We will have a different package structure for both product and customer in this example. This is important while creating configuration classes. While creating config classes, we need to specify the base packages which will be used by

```
src/main/java
- com.javadevjournal
      - product
        - data
        - repo
        - config
     - customer
       - data
       - repo
       - config
```

5. JPA Repositories

Let's create the *JPA repositories* for our Customer and Product entities. ***Spring Data JPA*** focuses on using JPA to store data in a relational database. Its most compelling feature is the ability to create repository implementations automatically, at runtime, from a repository interface.

5.1 Product Repository

```
package com.javadevjournal.product.repo;

import com.javadevjournal.product.data.ProductModel;
import org.springframework.data.jpa.repository.JpaRepository;

@Repository
public interface ProductRepository extends JpaRepository<ProductMod
}
```

```
package com.javadevjournal.customer.repo;

import com.javadevjournal.customer.data.CustomerModel;
import org.springframework.data.jpa.repository.JpaRepository;

@Repository
public interface CustomerRepository extends JpaRepository < Custome
}
```

6. Spring Configuration Classes

In the final step, we will create two *Spring configuration classes* whose responsibilities are to read the configurations and make sure all necessary setup/classes are available to our project on the application startup. We are creating configuration classes for the Product and Customer with the following details:

1. *DataSource* details
2. *EntityManagerFactory*
3. *TransactionManager*

To put it in simple words, we will have these separate configuration for both Customer and Product class.

6.1 Customer Configuration

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
 entityManagerFactoryRef = "customerEntityManagerFactory",
 transactionManagerRef = "customerTransactionManager",
```

```
public class CustomerConfig {

 @Primary
 @Bean(name = "customerDataSource")
 @ConfigurationProperties(prefix = "spring.datasource")
 public DataSource customerDataSource() {
  return DataSourceBuilder.create().build();
 }

 @Primary
 @Bean(name = "customerEntityManagerFactory")
 public LocalContainerEntityManagerFactoryBean
 entityManagerFactory(
  EntityManagerFactoryBuilder builder,
  @Qualifier("customerDataSource") DataSource dataSource
 ) {
  return builder
    .dataSource(dataSource)
    .packages("com.javadevjournal.customer.data")
    .persistenceUnit("db1")
    .build();
 }

 @Primary
 @Bean(name = "customerTransactionManager")
 public PlatformTransactionManager customerTransactionManager(
  @Qualifier("customerEntityManagerFactory") EntityManagerFactory c
 ) {
  return new JpaTransactionManager(customerEntityManagerFactory);
 }
}
```

[pullquote align="normal"]We are setting customerTranscationManager as the primary manager using `<em>@Primary</em>` annotation. This is important in case we are injecting transaction manager without specifying it. Spring will pick the primary bean in case multiple instances found during injection. [/pullquote]

6.2 Product Configuration

```
 entityManagerFactoryRef = "productEntityManagerFactory",
 transactionManagerRef = "productTransactionManager",
 basePackages = {
  "com.javadevjournal.product.repo"
 }
)
public class ProductConfig {

 @Bean(name = "productDataSource")
 @ConfigurationProperties(prefix = "db2.datasource")
 public DataSource dataSource() {
  return DataSourceBuilder.create().build();
 }

 @Bean(name = "productEntityManagerFactory")
 public LocalContainerEntityManagerFactoryBean
 barEntityManagerFactory(
  EntityManagerFactoryBuilder builder,
  @Qualifier("productDataSource") DataSource dataSource
 ) {
  return
  builder
   .dataSource(dataSource)
   .packages("com.javadevjournal.product.data")
   .persistenceUnit("db2")
   .build();
 }

 @Bean(name = "productTransactionManager")
 public PlatformTransactionManager productTransactionManager(
  @Qualifier("productEntityManagerFactory") EntityManagerFactory pr
 ) {
  return new JpaTransactionManager(productEntityManagerFactory);
 }
}
```

7. Testing

Let's create a simple test case to see the above code in action.

```
    @Autowired
    private ProductRepository productRepository;

    @Test
    @Transactional("productTransactionManager")
    public void create_check_product() {
        ProductModel product = new ProductModel("228781","Running S
        product = productRepository.save(product);

        assertNotNull(productRepository.findById(product.getId()));
    }

}
//Customer test
@RunWith(SpringRunner.class)
@SpringBootTest
public class MultipleDataSourcesCustomerTests {


    @Autowired
    private CustomerRepository customerRepository;

    @Test
    @Transactional("customerTransactionManager")
    public void create_check_customer() {

        CustomerModel customer = new CustomerModel("user@www.javade
        customer = customerRepository.save(customer);

        assertNotNull(customerRepository.findById(customer.getId())
        assertEquals(customerRepository.findById(customer.getId()).
    }
}
```

If you are using Spring Boot 2.0, please keep in mind that Spring Boot switched to Hikari and you may see some exception related to Jdbc URL configurations. Please Configure a Custom DataSource for more detail

Summary

application.The source code for this post is available on the GitHub

17 COMMENTS Newest

Navigation

Remote debug spring boot application with maven and IntelliJ

@ConfigurationProperties in Spring Boot

FailureAnalyzer in Spring Boot

Creating a Web Application with Spring Boot

Spring Boot Web Application Configuration