



Geeks México

BLOG DE PROGRAMACIÓN EN ESPAÑOL SOBRE JAVA,
FRAMEWORKS, BASES DE DATOS, CÓMPUTO EN LA NUBE, ETC.
EN ESPAÑOL Y EN INGLÉS.

[HOME](#)[ABOUT](#)[CONTACT](#)

Anuncios

[Report this ad](#)

Utiliza multiples bases de datos

con Spring boot y Spring JDBC

📅 [HACE 1 DÍA](#) 💬 [DEJA UN COMENTARIO](#)

Crear micro servicios que obtengan información de una base de datos es muy común, pero muchas veces la información que queremos obtener se encuentra en más de una base de datos, en este post explicaremos como escribir una aplicación REST spring boot que obtenga información de más de una base de datos.

Paso 1 Preparando las bases de datos

El primer paso será crear las dos bases de datos a las que conectaremos nuestra aplicación, en este ejemplo utilizaremos el motor de bases de datos MySQL, a continuación se presentan los scripts a utilizar para cada una:

- Base de datos 1

```
1 create database database1;
2 use database1;
3 CREATE TABLE USER(
4 USER_ID INTEGER PRIMARY KEY AUTO_INCREMENT
5 USERNAME VARCHAR(100) NOT NULL,
6 PASSWORD VARCHAR(100) NOT NULL
7 );
8 INSERT INTO USER (USERNAME,PASSWORD)VALUES
9 INSERT INTO USER (USERNAME,PASSWORD)VALUES
10 INSERT INTO USER (USERNAME,PASSWORD)VALUES
```

- Base de datos 2

```
1 create database database2;
2 use database2;
```

```
3 CREATE TABLE USER_LEGACY(  
4 USER_ID INTEGER PRIMARY KEY AUTO_INCREMENT  
5 USERNAME VARCHAR(100) NOT NULL,  
6 PASSWORD VARCHAR(100) NOT NULL  
7 );  
8 INSERT INTO USER_LEGACY (USERNAME, PASSWORD)  
9 INSERT INTO USER_LEGACY (USERNAME, PASSWORD)  
10 INSERT INTO USER_LEGACY (USERNAME, PASSWORD)
```

Con esto tendremos dos bases de datos, las cuales contienen tablas diferentes y datos diferentes.

Paso 2 Crear aplicación Spring boot

En este ejemplo tomaremos como base el proyecto [Spring Boot + REST Jersey Parte 1 \(https://geeks-mexico.com/2016/09/02/spring-boot-rest-jersey-parte-1/\)](https://geeks-mexico.com/2016/09/02/spring-boot-rest-jersey-parte-1/) que explica como configurar un proyecto básico de spring boot y le agregaremos las siguientes dependencias:

```
1 <dependency>  
2   <groupId>org.springframework.boot</groupId>  
3   <artifactId>spring-boot-starter-jdbc</artifactId>  
4 </dependency>  
5 <dependency>  
6   <groupId>mysql</groupId>  
7   <artifactId>mysql-connector-java</artifactId>  
8 </dependency>  
9 <dependency>  
10   <groupId>org.springframework.boot</groupId>  
11   <artifactId>spring-boot-configuration-  
12   <optional>true</optional>  
13 </dependency>
```

Como se puede ver haremos la conexión utilizando Spring JDBC e incluiremos el driver de mysql para conectarnos a las bases de datos.

Paso 3 Crear clase para representar a los usuarios

Como se puede ver, aunque son dos bases de datos diferentes y tablas diferentes ambas comparten las mismas columnas y tipos de datos, por esto solo crearemos una clase User para representar la tabla

USER que se encuentra en la base de datos database1 y la tabla USER_LEGACY que se encuentra en la base de datos database2.

```
1  /**
2   *
3   * @author raidentrance
4   *
5   */
6  public class User {
7      private Integer id;
8      private String user;
9      private String password;
10
11     public User() {
12     }
13
14     public User(Integer id, String user, String password) {
15         super();
16         this.id = id;
17         this.user = user;
18         this.password = password;
19     }
20
21     public Integer getId() {
22         return id;
23     }
24
25     public void setId(Integer id) {
26         this.id = id;
27     }
28
29     public String getUser() {
30         return user;
31     }
32
33     public void setUser(String user) {
34         this.user = user;
35     }
36
37     public String getPassword() {
38         return password;
39     }
40
41     public void setPassword(String password) {
42         this.password = password;
43     }
44
45 }
```

Como se puede ver la clase cuenta con los atributos id, user y password, esto es suficiente para hacer el mapping en las dos tablas.

Paso 4 Agregar la configuración de las bases de datos

El siguiente paso será agregar al archivo **application.properties** los datos necesarios para conectarse a ambas bases de datos, para esto agregaremos las siguientes líneas:

```
1 #Settings for database connection to data
2 spring.datasource.url=jdbc:mysql://localhost:3306/
3 spring.datasource.username=root
4 spring.datasource.password=root
5 spring.datasource.driver-class-name=com.mysql.jdbc.Driver
6
7 #Settings for database connection to data
8 legacy.datasource.url=jdbc:mysql://localhost:3306/
9 legacy.datasource.username=root
10 legacy.datasource.password=root
11 legacy.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Como se puede ver se incluye la información de ambos datasources.

Paso 5 Agregar configuración para ambos datasources

Agregar las líneas al archivo properties no es suficiente, ahora crearemos una clase de configuración que contendrá los dos **datasources** y **jdbcTemplate** a utilizar.

```
1 import javax.sql.DataSource;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.beans.factory.annotation.Value;
5 import org.springframework.boot.autoconfigure.jdbc.DataSourceProperties;
6 import org.springframework.boot.context.properties.ConfigurationProperties;
7 import org.springframework.context.annotation.Configuration;
8 import org.springframework.context.annotation.Primary;
9 import org.springframework.context.annotation.Bean;
10 import org.springframework.jdbc.core.JdbcTemplate;
11
12 /**
13  * @author raidentrance
14  *
15  */
16 @Configuration
17 public class DatabaseConfig {
18     @Bean(name = "dsSlave")
```

```
19 @ConfigurationProperties(prefix = "leg
20 public DataSource slaveDataSource() {
21     return DataSourceBuilder.create().
22 }
23
24 @Bean(name = "dsMaster")
25 @Primary
26 @ConfigurationProperties(prefix = "spr
27 public DataSource masterDataSource() {
28     return DataSourceBuilder.create().
29 }
30
31 @Bean(name = "jdbcSlave")
32 @Autowired
33 public JdbcTemplate slaveJdbcTemplate(
34     return new JdbcTemplate(dsSlave);
35 }
36
37 @Bean(name = "jdbcMaster")
38 @Autowired
39 public JdbcTemplate masterJdbcTemplate(
40     return new JdbcTemplate(dsMaster);
41 }
42 }
```

En el código anterior se crean 4 objetos 2 de tipo DataSource y 2 de tipo JdbcTemplate :

- DataSource
 - **slaveDataSource** : En la anotación **@ConfigurationProperties(prefix = "legacy.datasource")** se define la configuración de la base de datos legacy, que en este caso es la database2, como se puede ver es posible asignarle un nombre al bean que se está generando, en este caso es dsSlave.
 - **masterDataSource**: Este método devolverá el datasource para la base de datos database1 con el nombre dsMaster.
 - **slaveJdbcTemplate**: Para utilizar Spring JDBC es necesario utilizar un objeto de este tipo, como se puede ver el método recibe un objeto de tipo datasource el cuál es inyectado gracias a la anotación **@Autowired**, en este caso existen 2 beans de este tipo, por esto es

necesario incluir la anotación **@Qualifier("dsSlave")** para indicarle a Spring cuál de los dos datasources va a inyectar. Por último del mismo modo que en el anterior es posible definir un nombre al bean que se generará en este caso es jdbcSlave.

- **masterJdbcTemplate**: Este método devolverá el objeto de tipo **JdbcTemplate** con referencia a el datasource dsMaster con el nombre de jdbcMaster.

Paso 6 Creación de los daos

Como tendremos dos tablas diferentes en dos bases de datos diferentes crearemos dos daos uno apuntando a la tabla user en la base de datos database1 y el otro apuntando a la tabla user_legacy en la base de datos database2.

UserDao.java

```

1  import java.sql.ResultSet;
2  import java.sql.SQLException;
3  import java.util.List;
4
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.beans.factory.annotation.Qualifier;
7  import org.springframework.jdbc.core.JdbcTemplate;
8  import org.springframework.jdbc.core.RowMapper;
9  import org.springframework.stereotype.Component;
10
11 import com.raidentrance.model.api.User;
12
13 /**
14  * @author raidentrance
15  *
16  */
17 @Component
18 public class UserDao {
19
20     @Autowired
21     @Qualifier("jdbcMaster")
22     private JdbcTemplate jdbcTemplate;
23
24     public List<User> findAll() {
25         return jdbcTemplate.query("select
26             @Override

```

```

27         public User mapRow(ResultSet r
28             return new User(rs.getInt(
29         }
30     });
31 }
32 }

```

En este DAO se inyecta la referencia al JDBC template utilizando @Autowired y un @Qualifier("jdbcMaster") esto es para determinar cuál de los dos jdbctemplates tiene que inyectar en esta referencia, en este caso es el que contiene el datasource que apunta a la base de datos database1.

UserLegacyDao.java

```

1  import java.sql.ResultSet;
2  import java.sql.SQLException;
3  import java.util.List;
4
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.beans.factory.annotation.Qualifier;
7  import org.springframework.jdbc.core.JdbcTemplate;
8  import org.springframework.jdbc.core.RowMapper;
9  import org.springframework.stereotype.Component;
10
11  import com.raidentrance.model.api.User;
12
13  /**
14   * @author raidentrance
15   *
16   */
17  @Component
18  public class UserLegacyDao {
19      @Autowired
20      @Qualifier("jdbcSlave")
21      private JdbcTemplate jdbcTemplate;
22
23      public List<User> findAll() {
24          return jdbcTemplate.query("select
25              @Override
26              public User mapRow(ResultSet r
27                  return new User(rs.getInt(
28              }
29          });
30      }
31  }

```

En este DAO se inyecta la referencia al JDBC template utilizando @Autowired y un @Qualifier("jdbcSlave") esto es para determinar cuál de los dos jdbctemplates tiene que inyectar en esta referencia, en este caso es el que

contiene el datasource que apunta a la base de datos database2.

Paso 7 Creando un servicio común

Una vez que tenemos los dos DAOs en nuestra aplicación, el siguiente paso será crear un servicio común, en este se inyectarán ambos daos:

```
1  import java.util.List;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.stereotype.Service;
5
6  import com.raidentrance.dao.UserDao;
7  import com.raidentrance.dao.UserLegacyDao;
8  import com.raidentrance.model.api.User;
9
10 /**
11  * @author raidentrance
12  *
13  */
14 @Service
15 public class UserService {
16     @Autowired
17     private UserDao userDao;
18
19     @Autowired
20     private UserLegacyDao userLegacyDao;
21
22     public List<User> getUsers() {
23         return userDao.findAll();
24     }
25
26     public List<User> getLegacyUsers() {
27         return userLegacyDao.findAll();
28     }
29
30 }
```

Como se puede ver en esta clase se inyectan ambos daos, ya no es necesario utilizar algún qualifier porque son beans diferentes.

Paso 8 Exponiendo la información vía REST

El último paso será exponer esta información en servicios REST, para esto crearemos la siguiente clase:

```

1  import javax.ws.rs.Consumes;
2  import javax.ws.rs.GET;
3  import javax.ws.rs.Path;
4  import javax.ws.rs.Produces;
5  import javax.ws.rs.core.MediaType;
6  import javax.ws.rs.core.Response;
7
8  import org.slf4j.Logger;
9  import org.slf4j.LoggerFactory;
10 import org.springframework.beans.factory.annotation.Autowired;
11 import org.springframework.stereotype.Component;
12
13 import com.raidentrance.service.UserService;
14
15 /**
16  * @author raidentrance
17  *
18  */
19
20 @Component
21 @Path("/users")
22 @Consumes(MediaType.APPLICATION_JSON)
23 @Produces(MediaType.APPLICATION_JSON)
24 public class UserResource {
25
26     @Autowired
27     private UserService userService;
28
29     private static final Logger log = LoggerFactory.getLogger(UserResource.class);
30
31     @GET
32     public Response getUsers() {
33         log.info("Getting user");
34         return Response.ok(userService.getAllUsers()).build();
35     }
36
37     @GET
38     @Path("/legacy")
39     public Response getLegacyUsers() {
40         log.info("Getting user");
41         return Response.ok(userService.getLegacyUsers()).build();
42     }
43 }

```

En el código anterior se puede ver que se exponen 2 endpoints uno es /users que devolverá a todos los usuarios que se encuentran en la base de datos database1, y el otro es /users/legacy que devuelve a todos los usuarios que se encuentran en la base de datos database2.

Paso 9 Ejecutando los endpoints

Para ejecutar la aplicación solo ejecutaremos la clase `SprinBootApplication` que es la que contiene el main de nuestra aplicación e invocaremos los siguientes endpoints para ver sus salidas:

- GET /users

Salida:

```
1  [
2    {
3      "id": 1,
4      "user": "raidentrance",
5      "password": "superSecret"
6    },
7    {
8      "id": 2,
9      "user": "john",
10     "password": "smith"
11   },
12   {
13     "id": 3,
14     "user": "juan",
15     "password": "hola123"
16   }
17 ]
```

- GET /users/legacy

Salida

```
1  [
2    {
3      "id": 1,
4      "user": "rocky",
5      "password": "Adrianna"
6    },
7    {
8      "id": 2,
9      "user": "ivanDrago",
10     "password": "golpesFuertes"
11   },
12   {
13     "id": 3,
14     "user": "apolloCreed",
15     "password": "theBest"
16   }
17 ]
```

Puedes encontrar el código completo en el siguiente enlace <https://github.com/raidentrance/spring-boot-example/tree/part11-multipledb>

(<https://github.com/raidentrance/spring-boot-example/tree/part11-multipledb>).

Si te gusta el contenido y quieres enterarte cuando realicemos un post nuevo síguenos en nuestras redes sociales

https://twitter.com/geeks_mx (https://twitter.com/geeks_mx) y <https://www.facebook.com/geeksJavaMexico/> (<https://www.facebook.com/geeksJavaMexico/>).

Autor: Alejandro Agapito Bautista

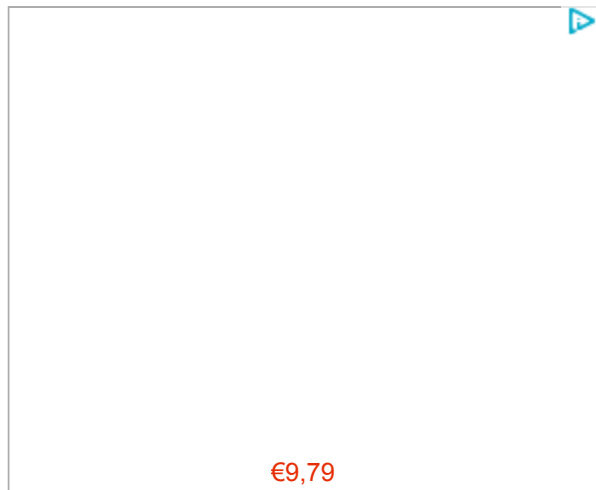
Twitter: @raidentrance

Contacto:raidentrance@gmail.com

Anuncios



[Report this ad](#)



[Report this ad](#)

ADVERTISEMENT



