Spring JPA – Multiple Databases

Last modified: October 23, 2022

by Eugen Paraschiv (https://www.baeldung.com/author/eugen)

Spring Data (https://www.baeldung.com/category/persistence/spring-persistence/spring-data)

JPA (https://www.baeldung.com/tag/jpa)

Spring Data JPA (https://www.baeldung.com/tag/spring-data-jpa)

JPA is huge! It covers nearly every aspect of communication between relational databases and the Java application and is deeply integrated into all major frameworks.

1. Overview

In this tutorial, we'll implement a simple Spring configuration for a **Spring Data JPA system with multiple databases.**

Further reading:

Spring Data JPA – Derived Delete Methods (/spring-data-jpa-deleteby)

Learn how to define Spring Data deleteBy and removeBy methods

Read more (/spring-data-jpa-deleteby) →

Configuring a DataSource Programmatically in Spring Boot (/spring-boot-configure-data-source-programmatic)

Learn how to configure a Spring Boot DataSource programmatically, thereby side-stepping Spring Boot's automatic DataSource configuration algorithm.

Read more (/spring-boot-configure-data-source-programmatic) →

2. The Entities

First, let's create two simple entities, with each living in a separate database.

Here is the first *User* entity:

```
package com.baeldung.multipledb.model.user;

@Entity
@Table(schema = "users")
public class User {

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

    private String name;

    @Column(unique = true, nullable = false)
    private String email;

    private int age;
}
```

And here's the second entity, *Product*:

```
package com.baeldung.multipledb.model.product;

@Entity
@Table(schema = "products")
public class Product {

    @Id
    private int id;

    private String name;

    private double price;
}
```

We can see that **the two entities are also placed in independent packages.** This will be important as we move into the configuration.

3. The JPA Repositories

Next, let's take a look at our two JPA repositories, *UserRepository*:

```
package com.baeldung.multipledb.dao.user;

public interface UserRepository
  extends JpaRepository<User, Integer> { }
```

and *ProductRepository*:

```
package com.baeldung.multipledb.dao.product;

public interface ProductRepository
  extends JpaRepository<Product, Integer> { }
```

Note again how we created these two repositories in different packages.

4. Configure JPA With Java

Now we'll get to the actual Spring configuration. **We'll first set up two configuration classes — one for the *User* and the other for the *Product*.**

In each configuration class, we'll need to define the following interfaces for *User*:

- *DataSource*
- *EntityManagerFactory* (*userEntityManager*)
- *TransactionManager* (*userTransactionManager*)

Let's start by looking at the User configuration:

```
@Configuration
@PropertySource({ "classpath:persistence-multiple-db.properties" })
@EnableJpaRepositories(
    basePackages = "com.baeldung.multipledb.dao.user",
```

```
    entityManagerFactoryRef = "userEntityManager",
    transactionManagerRef = "userTransactionManager"
)
public class PersistenceUserConfiguration {
    @Autowired
    private Environment env;

    @Bean
    @Primary
    public LocalContainerEntityManagerFactoryBean
userEntityManager() {
        LocalContainerEntityManagerFactoryBean em
          = new LocalContainerEntityManagerFactoryBean();
        em.setDataSource(userDataSource());
        em.setPackagesToScan(
          new String[] { "com.baeldung.multipledb.model.user" });

        HibernateJpaVendorAdapter vendorAdapter
          = new HibernateJpaVendorAdapter();
        em.setJpaVendorAdapter(vendorAdapter);
        HashMap<String, Object> properties = new HashMap<>();
        properties.put("hibernate.hbm2ddl.auto",
          env.getProperty("hibernate.hbm2ddl.auto"));
        properties.put("hibernate.dialect",
          env.getProperty("hibernate.dialect"));
        em.setJpaPropertyMap(properties);

        return em;
    }

    @Primary
    @Bean
    public DataSource userDataSource() {

        DriverManagerDataSource dataSource
          = new DriverManagerDataSource();
        dataSource.setDriverClassName(
          env.getProperty("jdbc.driverClassName"));
        dataSource.setUrl(env.getProperty("user.jdbc.url"));
        dataSource.setUsername(env.getProperty("jdbc.user"));
        dataSource.setPassword(env.getProperty("jdbc.pass"));

        return dataSource;
    }

    @Primary
    @Bean
    public PlatformTransactionManager userTransactionManager() {

        JpaTransactionManager transactionManager
          = new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(
          userEntityManager().getObject());
        return transactionManager;
```

```
    }
}
```

Notice how we use the *userTransactionManager* as our **Primary *TransactionManager*** by annotating the bean definition with *@Primary*. That's helpful whenever we're going to implicitly or explicitly inject the transaction manager without specifying which one by name.

Next, let's discuss *PersistenceProductConfiguration*, where we define similar beans:

```
@Configuration
@PropertySource({ "classpath:persistence-multiple-db.properties" })
@EnableJpaRepositories(
    basePackages = "com.baeldung.multipledb.dao.product",
    entityManagerFactoryRef = "productEntityManager",
    transactionManagerRef = "productTransactionManager"
)
public class PersistenceProductConfiguration {
    @Autowired
    private Environment env;

    @Bean
    public LocalContainerEntityManagerFactoryBean
productEntityManager() {
        LocalContainerEntityManagerFactoryBean em
          = new LocalContainerEntityManagerFactoryBean();
        em.setDataSource(productDataSource());
        em.setPackagesToScan(
          new String[] { "com.baeldung.multipledb.model.product" });

        HibernateJpaVendorAdapter vendorAdapter = new
HibernateJpaVendorAdapter();
        em.setJpaVendorAdapter(vendorAdapter);
        HashMap<String, Object> properties = new HashMap<>();
        properties.put("hibernate.hbm2ddl.auto",
          env.getProperty("hibernate.hbm2ddl.auto"));
        properties.put("hibernate.dialect",
          env.getProperty("hibernate.dialect"));
        em.setJpaPropertyMap(properties);

        return em;
    }

    @Bean
    public DataSource productDataSource() {

        DriverManagerDataSource dataSource
          = new DriverManagerDataSource();
        dataSource.setDriverClassName(
          env.getProperty("jdbc.driverClassName"));
        dataSource.setUrl(env.getProperty("product.jdbc.url"));
```

```
        dataSource.setUsername(env.getProperty("jdbc.user"));
        dataSource.setPassword(env.getProperty("jdbc.pass"));

        return dataSource;
    }

    @Bean
    public PlatformTransactionManager productTransactionManager() {

        JpaTransactionManager transactionManager
          = new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(
          productEntityManager().getObject());
        return transactionManager;
    }
}
```

5. Simple Test

Finally, let's test our configurations.

To do that, we will create an instance of each entity and make sure it is created:


```
@RunWith(SpringRunner.class)
@SpringBootTest
@EnableTransactionManagement
public class JpaMultipleDBIntegrationTest {

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private ProductRepository productRepository;

    @Test
    @Transactional("userTransactionManager")
    public void whenCreatingUser_thenCreated() {
        User user = new User();
        user.setName("John");
        user.setEmail("john@test.com");
        user.setAge(20);
        user = userRepository.save(user);

        assertNotNull(userRepository.findOne(user.getId()));
    }
```

```
    @Test
    @Transactional("userTransactionManager")
    public void whenCreatingUsersWithSameEmail_thenRollback() {
        User user1 = new User();
        user1.setName("John");
        user1.setEmail("john@test.com");
        user1.setAge(20);
        user1 = userRepository.save(user1);
        assertNotNull(userRepository.findOne(user1.getId()));

        User user2 = new User();
        user2.setName("Tom");
        user2.setEmail("john@test.com");
        user2.setAge(10);
        try {
            user2 = userRepository.save(user2);
        } catch (DataIntegrityViolationException e) {
        }

        assertNull(userRepository.findOne(user2.getId()));
    }

    @Test
    @Transactional("productTransactionManager")
    public void whenCreatingProduct_thenCreated() {
        Product product = new Product();
        product.setName("Book");
        product.setId(2);
        product.setPrice(20);
        product = productRepository.save(product);

        assertNotNull(productRepository.findOne(product.getId()));
    }
}
```

6. Multiple Databases in Spring Boot

Spring Boot can simplify the configuration above.

By default, **Spring Boot will instantiate its default *DataSource* with the configuration properties prefixed by *spring.datasource.****:

```
spring.datasource.jdbcUrl = [url]
spring.datasource.username = [username]
spring.datasource.password = [password]
```

We now want to keep using the same way to **configure the second *DataSource*, but with a different property namespace**:

```
spring.second-datasource.jdbcUrl = [url]
spring.second-datasource.username = [username]
spring.second-datasource.password = [password]
```

Because we want the Spring Boot autoconfiguration to pick up those different properties (and instantiate two different *DataSources*), we'll define two configuration classes similar to the previous sections:

```
@Configuration
@PropertySource({"classpath:persistence-multiple-db-boot.properties"})
@EnableJpaRepositories(
  basePackages = "com.baeldung.multipledb.dao.user",
  entityManagerFactoryRef = "userEntityManager",
  transactionManagerRef = "userTransactionManager")
public class PersistenceUserAutoConfiguration {

    @Primary
    @Bean
    @ConfigurationProperties(prefix="spring.datasource")
    public DataSource userDataSource() {
        return DataSourceBuilder.create().build();
    }
    // userEntityManager bean

    // userTransactionManager bean
}
```

```
@Configuration
@PropertySource({"classpath:persistence-multiple-db-boot.properties"})
@EnableJpaRepositories(
  basePackages = "com.baeldung.multipledb.dao.product",
  entityManagerFactoryRef = "productEntityManager",
  transactionManagerRef = "productTransactionManager")
public class PersistenceProductAutoConfiguration {

    @Bean
    @ConfigurationProperties(prefix="spring.second-datasource")
    public DataSource productDataSource() {
        return DataSourceBuilder.create().build();
    }

    // productEntityManager bean

    // productTransactionManager bean
}
```

Now we have defined the data source properties inside *persistence-multiple-db-boot.properties* according to the Boot autoconfiguration convention.

The interesting part is **annotating the data source bean creation method with *@ConfigurationProperties*.** We just need to specify the corresponding config prefix. Inside this method, we're using a *DataSourceBuilder*, and Spring Boot will automatically take care of the rest.

But how do the configured properties get injected into the *DataSource* configuration?

When calling the *build()* method on the *DataSourceBuilder*, it'll call its private *bind()* method:


```
public T build() {
    Class<? extends DataSource> type = getType();
    DataSource result = BeanUtils.instantiateClass(type);
    maybeGetDriverClassName();
    bind(result);
    return (T) result;
}
```

This private method performs much of the autoconfiguration magic, binding the resolved configuration to the actual *DataSource* instance:

```
private void bind(DataSource result) {
    ConfigurationPropertySource source = new
MapConfigurationPropertySource(this.properties);
    ConfigurationPropertyNameAliases aliases = new
ConfigurationPropertyNameAliases();
    aliases.addAliases("url", "jdbc-url");
    aliases.addAliases("username", "user");
    Binder binder = new Binder(source.withAliases(aliases));
    binder.bind(ConfigurationPropertyName.EMPTY,
Bindable.ofInstance(result));
}
```

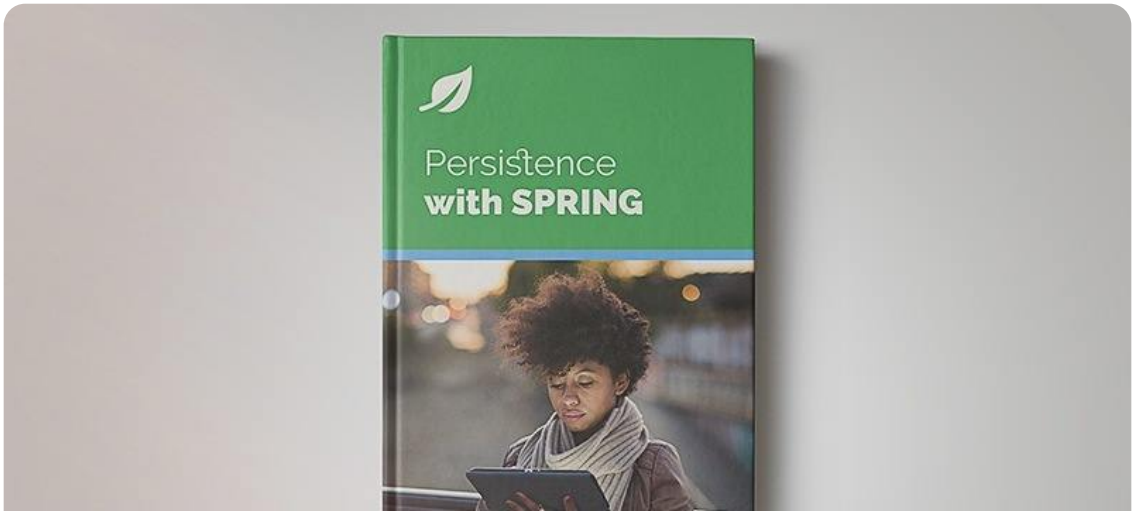
Although we don't have to touch any of this code ourselves, it's still useful to know what's happening under the hood of the Spring Boot autoconfiguration.

Besides this, the Transaction Manager and Entity Manager beans configuration is the same as the standard Spring application.

7. Conclusion

This article was a practical overview of how to configure our Spring Data JPA project to use multiple databases.

The **full implementation** can be found in the GitHub project (https://github.com/eugenp/tutorials/tree/master/persistence-modules/spring-data-jpa-enterprise-2). This is a Maven-based project, so it should be easy to import and run as it is.



Comments are closed on this article!

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)
OUR PARTNERS (/PARTNERS)
PARTNER WITH BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)