Baeldung

Persistence with Spring

Table of Contents

1: A Guide to JPA with Spring

2: Bootstrapping Hibernate 5 with Spring

Table of Contents

3: The DAO with Spring and Hibernate

4: Simplify the DAO with Spring and Java Generics

5: Transactions with Spring and JPA

6: Introduction to Spring Data JPA

7: Spring Data JPA @Query

Table of Contents

8: Spring JDBC

1: A Guide to JPA with Spring

1. Overview

This chapter **shows how to set up Spring with JPA**, using Hibernate as a persistence provider.

For a step by step introduction about setting up the Spring context using Java based configuration and the basic Maven pom for the project, see this article.

We'll start by setting up JPA in a Spring Boot project, then we'll look into the full configuration we need if we have a standard Spring project.

The Spring Boot project is intended to make creating Spring applications much faster and easier. This is done with the use of starters and auto-configuration for various Spring functionalities, JPA among them.

2.1. Maven Dependencies

To enable JPA in a Spring Boot application, we need the spring-boot-starter and spring-boot-starter-data-jpa dependencies:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.1.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
    <version>2.1.4.RELEASE</version>
</dependency>
```

The spring-boot-starter contains the necessary auto-configuration for Spring JPA. Also, the *spring-boot-starter-jpa project* references all the necessary dependencies such as hibernate-entitymanager.

2.2. Configuration

Spring Boot configures Hibernate as the default JPA provider, so it's no longer necessary to define the entityManagerFactory bean unless we want to customize it.

Spring Boot can also auto-configure the dataSource bean, depending on the database we're using. In the case of an in-memory database of type H2, HSQLDB and Apache Derby, Boot automatically configures the DataSource if the corresponding database dependency is present on the classpath.
For example, if we want to use an in-memory H2 database in a Spring Boot JPA application, we only need to add the h2 dependency to the pom.xml file:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.197</version>
</dependency>
```

This way, we don't need to define the *dataSource* bean, but we can do so if we want to customize it.

If we want to use JPA with *MySQL database*, then we need the *mysql-connector-java* dependency, as well as to define the *DataSource* configuration.

We can do this in a *@Configuration class*, or by using standard Spring Boot properties.

The Java configuration looks the same as it does in a standard Spring project:

```
@Bean
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();

    dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
    dataSource.setUsername("mysqluser");
    dataSource.setPassword("mysqlpass");
    dataSource.setUrl(
      "jdbc:mysql://localhost:3306/myDb?createDatabaseIfNotExist=true");

    return dataSource;
}
```

To configure the data source using a properties file, we have to set properties prefixed with *spring.datasource*:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=mysqluser
spring.datasource.password=mysqlpass
spring.datasource.url=
  jdbc:mysql://localhost:3306/myDb?createDatabaseIfNotExist=true
```

Spring Boot will automatically configure a data source based on these properties.

Also in Spring Boot 1, the default connection pool was *Tomcat*, but with Spring Boot 2 it has been changed to *HikariCP*.

You can find more examples of configuring JPA in Spring Boot in the *GitHub project.*

As we can see, the basic JPA configuration is fairly simple if we're using Spring Boot.

However, **if we have a standard Spring project, then we need more explicit configuration, using either Java or XML**. That's what we'll focus on in the next sections.

To use JPA in a Spring project, **we need to set up the *EntityManager*.**

This is the main part of the configuration and we can do it via a Spring factory bean. This can be either the simpler L*ocalEntityManagerFactoryBean* or **the more flexible *LocalContainerEntityManagerFactoryBean*.**

Let's see how we can use the latter option:

```
1.  @Configuration
2.  @EnableTransactionManagement
3.  public class PersistenceJPAConfig{
4.
5.     @Bean
6.     public LocalContainerEntityManagerFactoryBean entityManagerFactory()
7.  {
8.        LocalContainerEntityManagerFactoryBean em
9.          = new LocalContainerEntityManagerFactoryBean();
10.       em.setDataSource(dataSource());
11.       em.setPackagesToScan(new String[] { “com.baeldung.persistence.
12. model” });
13.
14.       JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
15.       em.setJpaVendorAdapter(vendorAdapter);
16.       em.setJpaProperties(additionalProperties());
17.
18.       return em;
19.    }
20.
21.    // ...
22.
23. }
```

We also need to explicitly define the DataSource bean we've used above:

```
@Bean
public DataSource dataSource(){
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/spring_jpa");
    dataSource.setUsername( "tutorialuser" );
    dataSource.setPassword( "tutorialmy5ql" );
    return dataSource;
}
```

The final part of the configuration are the additional Hibernate properties and the *TransactionManager* and *exceptionTranslation* beans:

```
@Bean
public PlatformTransactionManager
transactionManager(EntityManagerFactory emf) {
    JpaTransactionManager transactionManager = new
JpaTransactionManager();
    transactionManager.setEntityManagerFactory(emf);

    return transactionManager;
}

@Bean
public PersistenceExceptionTranslationPostProcessor
exceptionTranslation(){
    return new PersistenceExceptionTranslationPostProcessor();
}

Properties additionalProperties() {
    Properties properties = new Properties();
    properties.setProperty("hibernate.hbm2ddl.auto", "create-drop");
    properties.setProperty("hibernate.dialect", "org.hibernate.dialect.
MySQL5Dialect");

    return properties;
}
```

Next, let's see the same Spring Configuration with XML:

```
<bean id="myEmf"
  class="org.springframework.orm.jpa.
LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="com.baeldung.persistence.
model" />
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.
HibernateJpaVendorAdapter" />
    </property>
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.hbm2ddl.auto">create-drop</prop>
            <prop key="hibernate.dialect">org.hibernate.dialect.
MySQL5Dialect</prop>
        </props>
    </property>
</bean>

<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/spring_jpa"
/>
    <property name="username" value="tutorialuser" />
    <property name="password" value="tutorialmy5ql" />
</bean>

<bean id="transactionManager" class="org.springframework.orm.jpa.
JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf" />
</bean>
<tx:annotation-driven />

<bean id="persistenceExceptionTranslationPostProcessor" class=
  "org.springframework.dao.annotation.
PersistenceExceptionTranslationPostProcessor" />
```

There is a relatively small difference between the XML and the new Java-based configuration. Namely, in XML, a reference to another bean can point to either the bean or a bean factory for that bean.

In Java, however, since the types are different, the compiler doesn't allow it, and so the *EntityManagerFactory* is first retrieved from its bean factory and then passed to the transaction manager:

```
txManager.setEntityManagerFactory(this.entityManagerFactoryBean().
getObject());
```

Usually, JPA defines a persistence unit through the *META-INF/persistence.xml* file. ***Starting with Spring 3.1, the persistence.xml is no longer necessary***. The *LocalContainerEntityManagerFactoryBean* now supports a '*packagesToScan*' property where the packages to scan for @*Entity* classes can be specified.

This file was the last piece of XML we need to remove. **We can now set up JPA fully with no XML.**

We would usually specify JPA properties in the *persistence.xml* file. Alternatively, we can add the properties directly to the entity manager factory bean:

factoryBean.setJpaProperties(this.additionalProperties());

As a side-note, if Hibernate would be the persistence provider, then this would be the way to specify Hibernate specific properties.

In addition to the Spring Core and persistence dependencies – show in detail in the Spring with Maven tutorial – we also need to define JPA and Hibernate in the project, as well as a MySQL connector:

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>5.4.2.Final</version>
   <scope>runtime</scope>
</dependency>

<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>6.0.6</version>
   <scope>runtime</scope>
</dependency>
```

Note that the MySQL dependency is included as an example. We need a driver to configure the datasource, but any Hibernate supported database will do.

7. Conclusion

This chapter illustrated how to configure **JPA with Hibernate in Spring** in both a Spring Boot, and a standard Spring application.

As always, the code presented in this chapter is available over on Github

2: Bootstrapping Hibernate 5 with Spring

1. Overview

In this chapter, we'll discuss how to **bootstrap Hibernate 5 with Spring**, using both Java and XML configuration.

Bootstrapping a *SessionFactory* with the native Hibernate API is a bit complicated and would take us quite a few lines of code (have a look at the official documentation in case you really need to do that).

Fortunately, **Spring supports bootstrapping the *SessionFactory*** – so that we only need a few lines of Java code or XML configuration.

Also, before we jump in, if you're working with older versions of Hibernate, you can have a look at the articles about Hibernate 3 as well as Hibernate 4 with Spring.

Let's get started by first adding the necessary dependencies to our *pom.xml:*

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.2.Final</version>
</dependency>
```

The spring-orm module provides the Spring integration with Hibernate:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>5.1.6.RELEASE</version>
</dependency>
```

For the sake of simplicity, we'll use H2 as our database:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.197</version>
</dependency>
```

Finally, we are going to use Tomcat JDBC Connection Pooling, which fits better for production purposes than the DriverManagerDataSource provided by Spring:

```
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-dbcp</artifactId>
    <version>9.0.1</version>
</dependency>
```

As mentioned before, Spring supports us with bootstrapping the Hibernate SessionFactory.

All we have to do is to **define some beans as well as a few parameters**.

With Spring, we have **two options for these configurations**, a Java-based and an XML-based way.

4.1. Using Java Configuration

For using Hibernate 5 with Spring, little has changed since Hibernate 4: we have to use *LocalSessionFactoryBean* from the package org.*springframework.orm.hibernate5* instead of *org.springframework.orm.hibernate4.*

Like with Hibernate 4 before, we have to define beans for *LocalSessionFactoryBean*, *DataSource*, and *PlatformTransactionManager*, as well as some Hibernate-specific properties.

Let's create our HibernateConfig class to **configure Hibernate 5 with Spring:**

```
@Configuration
@EnableTransactionManagement
public class HibernateConf {

    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactory = new
LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource());
        sessionFactory.setPackagesToScan(
          {"com.baeldung.hibernate.bootstrap.model" });
        sessionFactory.setHibernateProperties(hibernateProperties());

        return sessionFactory;
    }

    @Bean
    public DataSource dataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:db;DB_CLOSE_DELAY=-1");
        dataSource.setUsername("sa");
        dataSource.setPassword("sa");

        return dataSource;
    }

    @Bean
    public PlatformTransactionManager hibernateTransactionManager() {
        HibernateTransactionManager transactionManager
          = new HibernateTransactionManager();
        transactionManager.setSessionFactory(sessionFactory().
getObject());
        return transactionManager;
    }

    private final Properties hibernateProperties() {
        Properties hibernateProperties = new Properties();
        hibernateProperties.setProperty(
          "hibernate.hbm2ddl.auto", "create-drop");
        hibernateProperties.setProperty(
          "hibernate.dialect", "org.hibernate.dialect.H2Dialect");

        return hibernateProperties;
    }
}
```

4.2. Using XML Configuration

As a secondary option, we can also **configure Hibernate 5 with an XML-based configuration:**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="...">

    <bean id="sessionFactory"
      class="org.springframework.orm.hibernate5.
LocalSessionFactoryBean">
        <property name="dataSource"
          ref="dataSource"/>
        <property name="packagesToScan"
          value="com.baeldung.hibernate.bootstrap.model"/>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.hbm2ddl.auto">
                    create-drop
                </prop>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.H2Dialect
                </prop>
            </props>
        </property>
    </bean>

    <bean id="dataSource"
      class="org.apache.tomcat.dbcp.dbcp2.BasicDataSource">
        <property name="driverClassName" value="org.h2.Driver"/>
        <property name="url" value="jdbc:h2:mem:db;DB_CLOSE_DELAY=-1"/>
        <property name="username" value="sa"/>
        <property name="password" value="sa"/>
    </bean>

    <bean id="txManager"
      class="org.springframework.orm.hibernate5.
HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
</beans>
```

As we can easily see, we're defining exactly the same beans and parameters as in the Java-based configuration earlier.

To bootstrap the XML into the Spring context, we can use a simple Java configuration file if the application is configured with Java configuration:

```
1.  @Configuration
2.  @EnableTransactionManagement
3.  @ImportResource({"classpath:hibernate5Configuration.xml"})
4.  public class HibernateXMLConf {
5.      //
6.  }
```

Alternatively, we can simply provide the XML file to the Spring Context, if the overall configuration is purely XML.

At this point, Hibernate 5 is fully configured with Spring, and we can **inject the raw Hibernate** *SessionFactory* directly whenever we need to:

```
1. public abstract class BarHibernateDAO {
2.
3.     @Autowired
4.     private SessionFactory sessionFactory;
5.
6.     // ...
7. }
```

6. Supported Databases

Unfortunately, the Hibernate project doesn't exactly provide an official list of supported databases.

That being said, **it's easy to see if a particular database type might be supported**, we can have a look at the list of supported dialects.

7. Conclusion

In this quick chapter, **we configured Spring with Hibernate 5** – with both Java and XML configuration.

As always, the full source code of the examples is available over on GitHub

3: The DAO with Spring and Hibernate

1. Overview

This chapter will show how to implement the DAO with Spring and Hibernate.
For the core Hibernate configuration, check out the previous chapter.

Starting Spring 3.0 and Hibernate 3.0.1, **the Spring *HibernateTemplate* is no longer necessary** to manage the Hibernate Session. It's now possible to make use of contextual sessions – **sessions managed directly by Hibernate** and active throughout the scope of a transaction.

As a consequence, it's now best practice to use the Hibernate API directly instead of the *HibernateTemplate*. This will effectively decouple the DAO layer implementation from Spring entirely.

2.1. Exception Translation without the *HibernateTemplate*

Exception Translation was one of the responsibilities of *HibernateTemplate* – translating the low-level Hibernate exceptions to higher level, generic Spring exceptions.

Without the template, **this mechanism is still enabled and active for all the DAOs annotated with the @*Repository* annotation**. Under the hood, this uses a Spring bean postprocessor that will advise all @*Repository* beans with all the PersistenceExceptionTranslator found in the Spring context.

One thing to remember is that exception translation uses proxies. For Spring to be able to create proxies around the DAO classes, these must not be declared as *final*.

2.2. Hibernate Session management without the Template

When Hibernate support for contextual sessions came out, the *HibernateTemplate* essentially became obsolete. In fact, the Javadoc of the class now highlights this aspect (bold from the original):

NOTE: As of Hibernate 3.0.1, transactional Hibernate access code can also be coded in plain Hibernate style. Hence, for newly started projects, consider adopting the standard Hibernate3 style of coding data access objects instead, based on {@link org.hibernate. SessionFactory#getCurrentSession()}.

We'll start with the **base DAO – an abstract, parametrized DAO** which supports the common generic operations and that we can extend for each entity:

```
1.  public abstract class AbstractHibernateDAO< T extends Serializable >{
2.     private Class< T > clazz;
3.
4.     @Autowired
5.     private SessionFactory sessionFactory;
6.
7.     public void setClazz(Class< T > clazzToSet) {
8.        clazz = clazzToSet;
9.     }
10.
11.    public T findOne(long id) {
12.       return (T) getCurrentSession().get( clazz, id );
13.    }
14.    public List< T > findAll() {
15.       return getCurrentSession()
16.        .createQuery( "from " + clazz.getName() ).list();
17.    }
18.
19.    public void save(T entity) {
20.       getCurrentSession().persist( entity );
21.    }
22.
23.    public T update(T entity) {
24.       return (T) getCurrentSession().merge( entity );
25.    }
26.
27.    public void delete(T entity) {
28.       getCurrentSession().delete( entity );
29.    }
30.    public void deleteById(long id) {
31.       final T entity = findOne( id);
32.       delete( entity );
33.    }
34.
35.    protected final Session getCurrentSession(){
36.       return sessionFactory.getCurrentSession();
37.    }
38. }
```

A few aspects are interesting here – as discussed, the abstract DAO doesn't extend any Spring template (such as *HibernateTemplate*). Instead, the Hibernate *SessionFactory* is injected directly in the DAO, and will have the role of the main Hibernate API, through the contextual Session it exposes:

this.sessionFactory.getCurrentSession();

Also, note that the constructor receives the *Class* of the entity as a parameter to be used in the generic operations.

Now, let's look at **an example implementation of this DAO**, for a Foo entity:

```
1.  @Repository
2.  public class FooDAO extends AbstractHibernateDAO< Foo > implements
3.  IFooDAO{
4.
5.     public FooDAO(){
6.        setClazz(Foo.class );
7.     }
8.  }
```

4. Conclusion

This chapter covered the configuration and implementation of the persistence layer with Hibernate and Spring.

The reasons to stop relying on templates for the DAO layer was discussed, as well as possible pitfalls of configuring Spring to manage transactions and the Hibernate Session. The final result is a lightweight, clean DAO implementation, with almost no compile-time reliance on Spring.

The implementation of this simple project can be found in the github project.

4: Simplify the DAO with Spring and Java Generics

1. Overview

This chapter will focus on **simplifying the DAO** layer by using a single, generified Data Access Object for all entities in the system, which will result in elegant data access, with no unnecessary clutter or verbosity.

We'll build on the Abstract DAO class we saw in our previous chapter on Spring and Hibernate, and add generics support.

Most production codebases have some kind of DAO layer. Usually, the implementation ranges from multiple classes with no abstract base class to some kind of generified class. However, one thing is consistent – there is always more than one. Most likely, there is a one to one relation between the DAOs and the entities in the system.

Also, depending on the level of generics involved, the actual implementations can vary from heavily duplicated code to almost empty, with the bulk of the logic grouped in a base abstract class.

These multiple implementations can usually be replaced by a single parametrized DAO. We can implement this such that no functionality is lost by taking full advantage of the type safety provided by Java Generics.

We'll show two implementations of this concept next, one for a Hibernate centric persistence layer and the other focusing on JPA.

These implementations are by no means complete, but we can easily add more additional data access methods are included.

2.1. The Abstract Hibernate DAO

Let's take a quick look at the *AbstractHibernateDao* class:

```
1.  public abstract class AbstractHibernateDao<T extends Serializable> {
2.
3.      private Class<T> clazz;
4.
5.      @Autowired
6.      SessionFactory sessionFactory;
7.
8.      public void setClazz(Class< T > clazzToSet) {
9.         this.clazz = clazzToSet;
10.     }
11.
12.     public List findAll() {
13.         return getCurrentSession().createQuery("from " +
14.           clazz.getName()).list();
15.     }
16.
17.     public T create(T entity) {
18.         getCurrentSession().saveOrUpdate(entity);
19.         return entity;
20.     }
21.
22.     public T update(T entity) {
23.         return (T) getCurrentSession().merge(entity);
24.     }
25.
26.     public void delete(T entity) {
27.         getCurrentSession().delete(entity);
28.     }
29.
30.     public void deleteById(long entityId) {
31.         T entity = findOne(entityId);
32.         delete(entity);
33.     }
34.
35.     protected Session getCurrentSession() {
36.         return sessionFactory.getCurrentSession();
37.     }
38. }
```

This is an abstract class with several data access methods, that uses the *SessionFactory* for manipulating entities.

2.2. The Generic Hibernate DAO

Now that we have the abstract DAO class, we can extend it just once. The generic DAO implementation will become the only implementation we need:

```
1.  @Repository
2.  @Scope(BeanDefinition.SCOPE_PROTOTYPE)
3.  public class GenericHibernateDao<T extends Serializable>
4.    extends AbstractHibernateDao<T> implements IGenericDao<T>{
5.     //
6.  }
```

First, **note that the generic implementation is itself parameterized**, allowing the client to choose the correct parameter on a case by case basis. This will mean that the clients get all the benefits of type safety without needing to create multiple artifacts for each entity.

Secondly, notice **the prototype scope of this generic DAO implementation**. Using this scope means that the Spring container will create a new instance of the DAO each time it's requested (including on autowiring). That will allow a service to use multiple DAOs with different parameters for different entities, as needed.

The reason this scope is so important is due to the way Spring initializes beans in the container. Leaving the generic DAO without a scope would mean using **the default singleton scope, which would lead to a single instance of the DAO living in the container**. That would obviously be majorly restrictive for any kind of more complex scenario.

The *IGenericDao* is simply an interface for all the DAO methods so that we can inject the implementation we need:

```
1.  public interface IGenericDao<T extends Serializable> {
2.     T findOne(final long id);
3.     List<T> findAll();
4.     void create(final T entity);
5.     T update(final T entity);
6.     void delete(final T entity);
7.     void deleteById(final long entityId);
8.  }
```

2.3. The Abstract JPA DAO

The AbstractJpaDao is very similar to the AbstractHibernateDao:

```
1.  public abstract class AbstractJpaDao< T extends Serializable > {
2.
3.     private Class< T > clazz;
4.
5.     @PersistenceContext
6.     EntityManager entityManager;
7.
8.     public void setClazz( Class< T > clazzToSet ) {
9.        this.clazz = clazzToSet;
10.    }
11.
12.    public T findOne( Long id ){
13.       return entityManager.find( clazz, id );
14.    }
15.    public List< T > findAll(){
16.       return entityManager.createQuery( "from " + clazz.getName() )
17.        .getResultList();
18.    }
19.
20.    public void save( T entity ){
21.       entityManager.persist( entity );
22.    }
23.
24.    public void update( T entity ){
25.       entityManager.merge( entity );
26.    }
27.
28.    public void delete( T entity ){
29.       entityManager.remove( entity );
30.    }
31.    public void deleteById( Long entityId ){
32.       T entity = getById( entityId );
33.       delete( entity );
34.    }
35. }
```

Similar to the Hibernate DAO implementation, we're using the Java Persistence API directly, without relying on the now deprecated Spring *JpaTemplate*.

2.4. The Generic JPA DAO

Similar to the Hibernate implementation, the JPA Data Access Object is straightforward as well:

```
1. @Repository
2. @Scope( BeanDefinition.SCOPE_PROTOTYPE )
3. public class GenericJpaDao< T extends Serializable >
4.  extends AbstractJpaDao< T > implements IGenericDao< T >{
5.    //
6. }
```

We now have a single DAO interface we can inject. We also need to specify the Class

```
1.  @Service
2.  class FooService implements IFooService{
3.
4.     IGenericDao<Foo> dao;
5.
6.     @Autowired
7.     public void setDao(IGenericDao<Foo> daoToSet) {
8.        dao = daoToSet;
9.        dao.setClazz(Foo.class);
10.    }
11.
12.    // ...
13. }
```

Spring **autowires the new DAO instance using setter injection** so that the implementation can be customized with the *Class* object. After this point, the DAO is fully parametrized and ready to be used by the service.

There are of course other ways that the class can be specified for the DAO – via reflection, or even in XML. My preference is towards this simpler solution because of the improved readability and transparency compared to using reflection.

4. Conclusion

This chapter discussed the **simplification of the Data Access Layer** by providing a single, reusable implementation of a generic DAO. We showed the implementation in both a Hibernate and a JPA based environment. The result is a streamlined persistence layer, with no unnecessary clutter.

For a step by step introduction about setting up the Spring context using Java based configuration and the basic Maven pom for the project, see this article.

Finally, the code for this chapter can be found in the GitHub project.

5: Transactions with Spring and JPA

1. Overview

This chapter will discuss the **right way to configure Spring Transactions**, how to use the @*Transactional* annotation and common pitfalls.

Basically, there are two distinct ways to configure Transactions – annotations and AOP – each with their own advantages. We're going to discuss the more common annotation-config here.

Spring 3.1 introduces the @*EnableTransactionManagement* annotation that we can use in a @*Configuration* class and enable transactional support:

```
1.  @Configuration
2.  @EnableTransactionManagement
3.  public class PersistenceJPAConfig{
4.
5.     @Bean
6.     public LocalContainerEntityManagerFactoryBean
7.       entityManagerFactoryBean(){
8.        //...
9.     }
10.
11.    @Bean
12.    public PlatformTransactionManager transactionManager(){
13.       JpaTransactionManager transactionManager
14.         = new JpaTransactionManager();
15.       transactionManager.setEntityManagerFactory(
16.         entityManagerFactoryBean().getObject() );
17.       return transactionManager;
18.    }
19. }
```

However, if we're using a Spring Boot project, and have a spring-data-* or spring-tx dependencies on the classpath, then transaction management will be enabled by default.

Before 3.1 or if Java is not an option, here is the XML configuration, using *annotation-driven* and the namespace support:

```
<bean id="txManager" class="org.springframework.orm.jpa.
JpaTransactionManager">
   <property name="entityManagerFactory" ref="myEmf" />
</bean>
<tx:annotation-driven transaction-manager="txManager" />
```

With transactions configured, we can now annotate a bean with @*Transactional* either at the class or method level:

```
@Service
@Transactional
public class FooService {
    //...
}
```

The annotation supports **further configuration** as well:

- the *Propagation Type* of the transaction
- the *Isolation Level* of the transaction
- a *Timeout* for the operation wrapped by the transaction
- a *readOnly flag* – a hint for the persistence provider that the transaction should be read only
- the Rollback rules for the transaction

Note that – by default, rollback happens for runtime, unchecked exceptions only. The checked exception does not trigger a rollback of the transaction. We can, of course, configure this behavior with the *rollbackFor* and *noRollbackFor* annotation parameters.

5.1. Transactions and Proxies

At a high level, **Spring creates proxies for all the classes annotated with @*Transactional*** – either on the class or on any of the methods. The proxy allows the framework to inject transactional logic before and after the running method – mainly for starting and committing the transaction.

What's important to keep in mind is that, if the transactional bean is implementing an interface, by default the proxy will be a Java Dynamic Proxy. This means that only external method calls that come in through the proxy will be intercepted. **Any self-invocation calls will not start any transaction**, even if the method has the **@*Transactional* annotation**.

Another caveat of using proxies is that **only public methods should be annotated with @*Transactional*.** Methods of any other visibilities will simply ignore the annotation silently as these are not proxied.

This article discusses further proxying pitfalls in great detail here.

5.2. Changing the Isolation level

We can also change the transaction isolation level:

```
1.  @Transactional(isolation = Isolation.SERIALIZABLE)
```

Note that this has actually been introduced in Spring 4.1; if we run the above example before Spring 4.1, it will result in:

"*org.springframework.transaction.InvalidIsolationLevelException*: Standard JPA does not support custom isolation levels – use a special JpaDialect for your JPA implementation"

5.3. Read-Only Transactions

The *readOnly* flag usually generates confusion, especially when working with JPA; from the Javadoc:

"This just serves as a hint for the actual transaction subsystem; it will *not necessarily* cause failure of write access attempts. A transaction manager which cannot interpret the read-only hint will *not* throw an exception when asked for a read-only transaction."

The fact is that **we can't be sure that an insert or update will not occur when the *readOnly* flag is set**. This behavior is vendor dependent, whereas JPA is vendor agnostic.

It's also important to understand that **the *readOnly* flag is only relevant inside a transaction**. If an operation occurs outside of a transactional context, the flag is simply ignored. A simple example of that would call a method annotated with:

```
1.  @Transactional(propagation = Propagation.SUPPORTS,readOnly = true )
```

from a non-transactional context – a transaction will not be created and the *readOnly* flag will be ignored.

5.4. Transaction Logging

A helpful method to understand transactional related issues is fine-tuning logging in the transactional packages. The relevant package in Spring is "*org.springframework.transaction*", *which should be configured with a logging level of TRACE*.

6. Conclusion

We covered the basic configuration of transactional semantics using both Java and XML, how to use @*Transactional* and best practices of a Transactional Strategy.

As always, the code presented in this chapter is available over on Github.

6: Introduction to Spring Data JPA

1. Overview

This chapter will focus on **introducing Spring Data JPA into a Spring project** and fully configuring the persistence layer.

As we discussed in an earlier chapter, the DAO layer usually consists of a lot of boilerplate code that can and should be simplified. The advantages of such a simplification are many: a decrease in the number of artifacts that we need to define and maintain, consistency of data access patterns and consistency of configuration.

Spring Data takes this simplification one step forward and **makes it possible to remove the DAO implementations entirely**. The interface of the DAO is now the only artifact that we need to explicitly define.

In order to start leveraging the Spring Data programming model with JPA, a DAO interface needs to extend the JPA specific *Repository* interface – *JpaRepository*. This will enable Spring Data to find this interface and automatically create an implementation for it.

By extending the interface we get the most relevant CRUD methods for standard data access available in a standard DAO.

As discussed, **by implementing one of the Repository interfaces, the DAO will already have some basic CRUD methods (and queries) defined and implemented**.

To define more specific access methods, Spring JPA supports quite a few options:

- simply **define a new method** in the interface
- provide the actual **JPQ query** by using the @*Query* annotation
- use the more advanced **Specification and Querydsl support** in Spring Data
- define **custom queries** via JPA Named Queries

The third option – the Specifications and Querydsl support – is similar to JPA Criteria but using a more flexible and convenient API. This makes the whole operation much more readable and reusable. The advantages of this API will become more pronounced when dealing with a large number of fixed queries, as we could potentially express these more concisely through a smaller number of reusable blocks.

This last option has the disadvantage that it either involves XML or burdening the domain class with the queries.

3.1. Automatic Custom Queries

When Spring Data creates a new *Repository* implementation, it analyses all the methods defined by the interfaces and tries to **automatically generate queries from the method names**. While this has some limitations, it's a very powerful and elegant way of defining new custom access methods with very little effort.

Let's look at an example: if the entity has a name field (and the Java Bean standard *getName* and *setName* methods), **we'll define the findByName method in the DAO interface**; this will automatically generate the correct query:

```
1.  public interface IFooDAO extends JpaRepository<Foo, Long> {
2.
3.     Foo findByName(String name);
4.
5.  }
```

This is a relatively simple example. The query creation mechanism supports a much larger set of keywords.

In case that the parser cannot match the property with the domain object field, we'll see the following exception:

```
1.  java.lang.IllegalArgumentException: No property nam found for type class
2.  org.rest.model.Foo
```

3.2. Manual Custom Queries

Let's now look at a custom query that we'll define via the @*Query* annotation:

```
1. @Query("SELECT f FROM Foo f WHERE LOWER(f.name) = LOWER(:name)")
2. Foo retrieveByName(@Param("name") String name);
```

For even more fine-grained control over the creation of queries, such as using named parameters or modifying existing queries, the reference is a good place to start.

The actual implementation of the Spring Data managed DAO is indeed hidden since we don't work with it directly. However, this is a simple enough implementation – the ***SimpleJpaRepository*** **– which defines transaction semantics using annotations**.

More explicitly, this uses a read-only @*Transactional* annotation at the class level, which is then overridden for the non-read-only methods. The rest of the transaction semantics are default, but these can be easily overridden manually per method.

4.1. Exception Translation is Alive and Well

The question is now – since we're not using the default Spring ORM templates (*JpaTemplate, HibernateTemplate*) – are we losing exception translation by using Spring Data JPA? Are we not going to get our JPA exceptions translated to Spring's *DataAccessException* hierarchy?

Of course not – **exception translation is still enabled by the use of the @Repository annotation on the DAO**. This annotation enables a Spring bean postprocessor to advise all @*Repository* beans with all the *PersistenceExceptionTranslator* instances found in the Container, and provide exception translation just as before.

Let's verify exception translation with an integration test:

```
1.  @Test(expected = DataIntegrityViolationException.class)
2.  public void givenFooHasNoName_whenInvalidEntityIsCreated_
3.  thenDataException() {
4.      service.create(new Foo());
5.  }
```

Keep in mind that **exception translation is done through proxies**. In order for Spring to be able to create proxies around the DAO classes, these must not be declared *final*.

To activate the Spring JPA repository support we can use the *@EnableJpaRepositories* annotation and specify the package that contains the DAO interfaces:

```
1. @EnableJpaRepositories(basePackages = “com.baeldung.jpa.dao”)
2. public class PersistenceConfig { ... }
```

We can do the same with an XML configuration:

```
1. <jpa:repositories base-package=”org.rest.dao.spring” />
```

We already discussed in great detail how to configure JPA in Spring in a previous chapter. Spring Data also takes advantage of the Spring support for the JPA @*PersistenceContext* annotation. It uses this to wire the *EntityManagerinto* the Spring factory bean responsible for creating the actual DAO implementations – *JpaRepositoryFactoryBean*.

In addition to the already discussed configuration, we also need to include the Spring Data XML Config – if we are using XML:

```
1.  @Configuration
2.  @EnableTransactionManagement
3.  @ImportResource( “classpath*:*springDataConfig.xml” )
4.  public class PersistenceJPAConfig{
5.      ...
6.  }
```

In addition to the Maven configuration for JPA-defined in a previous chapter, the spring-data-jpa dependency is added:

```
<dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-jpa</artifactId>
   <version>2.1.6.RELEASE</version>
</dependency>
```

We can also use the Spring Boot Starter Data JPA dependency that will automatically configure the DataSource for us.

We also need to make sure that the database we want to use is present in the classpath. In our example, we've added the H2 in-memory database:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
   <version>2.1.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.197</version>
</dependency>
```

That's it, just by doing these dependencies, our application is up and running and we can use it for other database operations.

The explicit configuration for a standard Spring application is now included as part of Spring Boot auto-configuration.

We can, of course, modify the auto-configuration by adding our own explicit configuration.

Spring Boot provides an easy way to do this using properties in the *application.properties file*:

```
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1
spring.datasource.username=sa
spring.datasource.password=sa
```

In this example, we've changed the connection URL and credentials.

9. Conclusion

This chapter covered the configuration and implementation of the persistence layer with Spring 4, JPA 2 and Spring Data JPA (part of the Spring Data umbrella project), using both XML and Java based configuration.

We discussed ways to define more **advanced custom queries**, as well as a **configuration with the new *jpa* name space** and transactional semantics. The final result is a new and elegant take on data access with Spring, with almost no actual implementation work.

The implementation of this chapter can be found in the GitHub project.

7: Spring Data JPA @Query

1. Overview

Spring Data provides many ways to define a query that we can execute. One of these is the @Query annotation.

In this chapter, we'll demonstrate **how to use the @*Query* annotation in Spring Data JPA to execute both JPQL and native SQL queries.**

Also, we'll show how to build a dynamic query when the @*Query* annotation is not enough.

In order to define SQL to execute for a Spring Data repository method, we can **annotate the method with the @*Query* annotation — its value attribute contains the JPQL or SQL to execute.**

The @*Query* annotation takes precedence over named queries, which are annotated with @*NamedQuery* or defined in an orm.xml file.

It's a good approach to place a query definition just above the method inside the repository rather than inside our domain model as named queries. The repository is responsible for persistence, so it's a better place to store these definitions.

2.1. JPQL

By default the query definition uses JPQL.

Let's look at a simple repository method that returns active *User* entities from the database:

```
1. @Query("SELECT u FROM User u WHERE u.status = 1")
2. Collection<User> findAllActiveUsers();
```

2.2. Native

We can use also native SQL to define our query. All we have to do is to set the value of the *nativeQuery* attribute to true and define the native SQL query in the *value* attribute of the annotation

```
1. @Query(
2.   value = "SELECT * FROM USERS u WHERE u.status = 1",
3.   nativeQuery = true)
4. Collection<User> findAllActiveUsersNative();
```

We can pass an additional parameter of type *Sort* to a Spring Data method declaration that has the @*Query* annotation. It'll be translated into the *ORDER BY* clause that gets passed to the database.

3.1. Sorting for JPA Provided and Derived Method

For the methods we get out-of-the-box like *findAll(Sort)* or the ones that are generated by parsing method signatures, **we can only use object properties to define our sort:**

```
1.  userRepository.findAll(new Sort(Sort.Direction.ASC, "name"));
```

Now imagine that we want to sort by the length of a name property:

```
1.  userRepository.findAll(new Sort("LENGTH(name)"));
```

When we execute the above code we'll receive an exception:

org.springframework.data.mapping
PropertyReferenceException: No Property LENGTH(name)
found for type User!

3.2. JPQL

When we use JPQL for a query definition, then Spring Data can handle sorting without any problem — all we have to do is to add a method parameter of type *Sort*:

```
1. @Query(value = "SELECT u FROM User u")
2. List<User> findAllUsers(Sort sort);
```

We can call this method and pass a *Sort* parameter, which will order the result by the *name* property of the *Userobject*:

```
1. userRepository.findAllUsers(new Sort("name"));
```

And because we used @*Query* annotation, **we can use the same method to get the sorted list of *Users* by the length of their names**:

```
1. userRepository.findAllUsers(JpaSort.unsafe("LENGTH(name)"));
```

It's crucial that we use *JpaSort.unsafe()* to create a *Sort* object instance. When we use:

```
1. new Sort("LENGTH(name)");
```

then we'll receive exactly the same exception as we saw above for the *findAll()* method.

When Spring Data discovers the unsafe *Sort* order for a method that uses the @*Query* annotation, then it just appends the sort clause to the query — it skips checking whether the property to sort by belongs to the domain model.

3.3. Native

When the @*Query* annotation uses native SQL, then it's not possible to define a *Sort*.

If we do, we'll receive an exception:

org.springframework.data.jpa.repository.query.
InvalidJpaQueryMethodException: Cannot use native queries with dynamic sorting and/or pagination

As the exception says, the sort isn't supported for native queries. The error message gives us a hint that pagination will cause an exception too. However, there is a workaround that enables pagination, and we'll cover in the next section.

Pagination allows us to return just a subset of a whole result in a *Page*. This is useful, for example, when navigating through several pages of data on a web page.

Another advantage of pagination is that the amount of data sent from server to client is minimized. By sending smaller pieces of data, we can generally see an improvement in performance.

4.1. JPQL

Using pagination in the JPQL query definition is straightforward:

```
1. @Query(value = "SELECT u FROM User u ORDER BY id")
2. Page<User> findAllUsersWithPagination(Pageable pageable);
```

We can pass a *PageRequest* parameter to get a page of data. Pagination is also supported for native queries but requires a little bit of additional work.

4.2. Native

We can **enable pagination for native queries by declaring an additional attribute *countQuery*** — this defines the SQL to execute to count the number of rows in the whole result:

```
1. @Query(
2.   value = "SELECT * FROM Users ORDER BY id",
3.   countQuery = "SELECT count(*) FROM Users",
4.   nativeQuery = true)
5. Page<User> findAllUsersWithPagination(Pageable pageable);
```

4.3. Spring Data JPA Versions Prior to 2.0.4

The above solution for native queries works fine for Spring Data JPA version 2.0.4 and later.

Prior to that version, when we try to execute such a query we'll receive an exception — the same one we described in the previous section on sorting.

We can overcome this by adding an additional parameter for pagination inside our query:

```
1. @Query(
2.   value = "SELECT * FROM Users ORDER BY id \n-- #pageable\n",
3.   countQuery = "SELECT count(*) FROM Users",
4.   nativeQuery = true)
5. Page<User> findAllUsersWithPagination(Pageable pageable);
```

In the above example, we add "\n– #pageable\n" as the placeholder for the pagination parameter. This tells Spring Data JPA how to parse the query and inject the pageable parameter. This solution works for the *H2* database.

We've covered how to create simple select queries via JPQL and native SQL. Next, we'll show how to define additional parameters.

There are two possible ways that we can pass method parameters to our query. In this section, we'll cover indexed parameters.

5.1. JPQL

For indexed parameters in JPQL, Spring Data will **pass method parameters to the query in the same order they appear in the method declaration:**

```
1.  @Query("SELECT u FROM User u WHERE u.status = ?1")
2.  User findUserByStatus(Integer status);
3.
4.  @Query("SELECT u FROM User u WHERE u.status = ?1 and u.name = ?2")
5.  User findUserByStatusAndName(Integer status, String name);
```

For the above queries, the *status* method parameter will be assigned to the query parameter with index 1, and the *name* method parameter will be assigned to the query parameter with index 2.

5.2. Native

Indexed parameters for the native queries work exactly in the same way as for JPQL:

```
1.  @Query(
2.    value = "SELECT * FROM Users u WHERE u.status = ?1",
3.    nativeQuery = true)
4.  User findUserByStatusNative(Integer status);
```

In the next section, we'll show a different approach — passing parameters via name.

We can also **pass method parameters to the query using named parameters**. We define these using the @*Param* annotation inside our repository method declaration.

Each parameter annotated with @*Param* must have a value string matching the corresponding JPQL or SQL query parameter name. A query with named parameters is easier to read and is less error-prone in case the query needs to be refactored.

6.1. JPQL

As mentioned above, we use the @*Param* annotation in the method declaration to match parameters defined by name in JPQL with parameters from the method declaration:

```
1.  @Query("SELECT u FROM User u WHERE u.status = :status and u.name =
2.  :name")
3.  User findUserByStatusAndNameNamedParams(
4.    @Param("status") Integer status,
5.    @Param("name") String name);
```

Note that in the above example, we defined our SQL query and method parameters to have the same names, but it's not required, as long as the value strings are the same:

```
1.  @Query("SELECT u FROM User u WHERE u.status = :status and u.name =
2.  :name")
3.  User findUserByUserStatusAndUserName(@Param("status") Integer
4.  userStatus,
5.    @Param("name") String userName);
```

6.2. Native

For the native query definition, there is no difference how we pass a parameter via the name to the query in comparison to JPQL — we use the @*Param* annotation:

```
1.  @Query(value = “SELECT * FROM Users u WHERE u.status = :status and
2.  u.name = :name”,
3.    nativeQuery = true)
4.  User findUserByStatusAndNameNamedParamsNative(
5.    @Param(“status”) Integer status, @Param(“name”) String name);
```

7. Collection Parameter

Let's consider the case when the where clause of our JPQL or SQL query contains the *IN (or NOT IN)* keyword

```
1.  SELECT u FROM User u WHERE u.name IN :names
```

In this case we can define a query method which takes *Collection* as a parameter:

```
1.  @Query(value = "SELECT u FROM User u WHERE u.name IN :names")
2.  List<User> findUserByNameList(@Param("names") Collection<String> names);
```

As the parameter is a *Collection* it can be used with *List, HashSet*, etc.
Next, we'll show how to modify data with the @*Modifying* annotation.

We can use the @***Query*** **annotation to modify the state of the database by also adding the @*Modifying* annotation** to the repository method.

8.1. JPQL

The repository method that modifies the data has two difference in comparison to the *select* query — it has the @*Modifying* annotation and, of course, the JPQL query uses *update* instead of select:

```
1.  @Modifying
2.  @Query("update User u set u.status = :status where u.name = :name")
3.  int updateUserSetStatusForName(@Param("status") Integer status,
4.    @Param("name") String name);
```

The return value defines how many rows the execution of the query updated. Both indexed and named parameters can be used inside update queries.

8.2. Native

We can modify the state of the database also with a native query — we just need to add the @*Modifying* annotation:

```
1.  @Modifying
2.  @Query(value = "update Users u set u.status = ? where u.name = ?",
3.    nativeQuery = true)
4.  int updateUserSetStatusForNameNative(Integer status, String name);
```

8.3. Inserts

To perform an insert operation, we have to both apply @Modifying and use a native query since INSERT is not a part of the JPA interface:

```
1.  @Modifying
2.  @Query(value = “insert into Users (name, age, email, status) values
3.  (:name, :age, :email, :status)”,
4.    nativeQuery = true)
5.  void insertUser(@Param(“name”) String name, @Param(“age”) Integer
6.  age,
7.    @Param(“status”) Integer status, @Param(“email”) String email);
```

Often times, we'll encounter the need for building SQL statements based on conditions or data sets whose values are only known at runtime. And, in those cases, we can't just use a static query.

9.1. Example of a Dynamic Query

For example, let's imagine a situation, where we need to select all the users whose email is *LIKE* one from a set defined at runtime — *email1, email2, …, emailn:*

```
1.  SELECT u FROM User u WHERE u.email LIKE '%email1%'
2.      or  u.email LIKE '%email2%'
3.      ...
4.      or  u.email LIKE '%emailn%'
```

Since the set is dynamically constructed, we can't know at compile-time how many *LIKE* clauses to add.

In this case, **we can't just use the @*Query* annotation since we can't provide a static SQL statement**.

Instead, by implementing a custom composite repository, we can extend the base *JpaRepository* functionality and provide our own logic for building a dynamic query. Let's take a look at how to do this.

9.2. Custom Repositories and the JPA Criteria API

Luckily for us, Spring provides a way for extending the base repository through the use of custom fragment interfaces. We can then link them together to create a composite repository.

We'll start by creating a custom fragment interface:

```
1. public interface UserRepositoryCustom {
2.     List<User> findUserByEmails(Set<String> emails);
3. }
```

And then, we'll implement it:

```
1. public class UserRepositoryCustomImpl implements UserRepositoryCustom {
2.
3.     @PersistenceContext
4.     private EntityManager entityManager;
5.
6.     @Override
7.     public List<User> findUserByEmails(Set<String> emails) {
8.         CriteriaBuilder cb = entityManager.getCriteriaBuilder();
9.         CriteriaQuery<User> query = cb.createQuery(User.class);
10.        Root<User> user = query.from(User.class);
11.
12.        Path<String> emailPath = user.get("email");
13.
14.        List<Predicate> predicates = new ArrayList<>();
15.        for (String email : emails) {
16.            predicates.add(cb.like(emailPath, email));
17.        }
18.        query.select(user)
19.            .where(cb.or(predicates.toArray(new Predicate[predicates.
20. size()])));
21.
22.        return entityManager.createQuery(query)
23.            .getResultList();
24.    }
25. }
```

As shown above, we leveraged the JPA Criteria API to build our dynamic query.

Also, we need to make sure to include the Impl postfix in the class name. Spring will search the *UserRepositoryCustom* implementation as *UserRepositoryCustomImpl*. Since fragments are not repositories by themselves, Spring relies on this mechanism to find the fragment implementation.

9.3. Extending the Existing Repository

Notice that all the query methods from section 2 – section 7 are in the *UserRepository*. So now, we'll integrate our fragment by extending the new interface in the *UserRepository*:

```
1.  public interface UserRepository extends JpaRepository<User,
2.  Integer>, UserRepositoryCustom {
3.      //  query methods from section 2 - section 7
4.  }
```

9.4. Using the Repository

And finally, we can call our dynamic query method:

```
1.  Set<String> emails = new HashSet<>();
2.  // filling the set with any number of items
3.
4.  userRepository.findUserByEmails(emails);
```

We've successfully created a composite repository and called our custom method.

In this chapter, we covered several ways of defining queries in Spring Data JPA repository methods using the @*Query*annotation.

Also, we learned how to implement a custom repository and create a dynamic query.

As always, the complete code examples used in this chapter are available over on Github.

8: Spring JDBC

1. Overview

In this chapter, we'll go through practical use cases of the Spring JDBC module.

All the classes in Spring JDBC are divided into four separate packages:

- ***core*** – the core functionality of JDBC. Some of the important classes under this package include *JdbcTemplate*,*SimpleJdbcInsert*, *SimpleJdbcCall* and *NamedParameterJdbcTemplate*.

- ***datasource*** – utility classes to access a datasource. It also has various datasource implementations for testing JDBC code outside the Java EE container.

- ***object*** – DB access in an object-oriented manner. It allows executing queries and returning the results as a business object. It also maps the query results between the columns and properties of business objects.

- ***support*** – support classes for classes under *core* and *object* packages. E.g. provides the *SQLExceptiontranslation* functionality.

To begin with, let's start with some simple configuration of the data source (we'll use a MySQL database for this example):

```
@Configuration
@ComponentScan("com.baeldung.jdbc")
public class SpringJdbcConfig {
    @Bean
    public DataSource mysqlDataSource() {
        DriverManagerDataSource dataSource = new
DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/springjdbc");
        dataSource.setUsername("guest_user");
        dataSource.setPassword("guest_password");

        return dataSource;
    }
}
```

Alternatively, we can also make good use of an embedded database for development or testing – here is a quick configuration that creates an instance of H2 embedded database and pre-populates it with simple SQL scripts:

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.H2)
        .addScript("classpath:jdbc/schema.sql")
        .addScript("classpath:jdbc/test-data.sql").build();
}
```

Finally – the same can, of course, be done using XML configuring for the *datasource*:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/
springjdbc"/>
    <property name="username" value="guest_user"/>
    <property name="password" value="guest_password"/>
</bean>
```

3.1. Basic Queries

The JDBC template is the main API through which we'll access most of the functionality that we're interested in:

- creation and closing of connections
- executing statements and stored procedure calls
- iterating over the *ResultSet* and returning results

Firstly, let's start with a simple example to see what the *JdbcTemplate* can do:

```
1. int result = jdbcTemplate.queryForObject(
2.     "SELECT COUNT(*) FROM EMPLOYEE", Integer.class);
```

and also here's a simple INSERT:

```
1. public int addEmplyee(int id) {
2.     return jdbcTemplate.update(
3.         "INSERT INTO EMPLOYEE VALUES (?, ?, ?, ?)", 5, "Bill",
4. "Gates", "USA");
5. }
```

Notice the standard syntax of providing parameters – using the `?` character. Next – let's look at an alternative to this syntax.

3.2. Queries with Named Parameters

To get **support for named parameters**, we'll use the other JDBC template provided by the framework – the *NamedParameterJdbcTemplate*.

Additionally, this wraps the *JbdcTemplate* and provides an alternative to the traditional syntax using "?" to specify parameters. Under the hood, it substitutes the named parameters to JDBC "?" placeholder and delegates to the wrapped *JDCTemplate* to execute the queries:

```
1.  SqlParameterSource namedParameters = new MapSqlParameterSource().
2.  addValue("id", 1);
3.  return namedParameterJdbcTemplate.queryForObject(
4.      "SELECT FIRST_NAME FROM EMPLOYEE WHERE ID = :id",
5.  namedParameters, String.class);
```

Notice how we are using the *MapSqlParameterSource* to provide the values for the named parameters.

For instance, let's look at below example that uses properties from a bean to determine the named parameters:

```
1.  Employee employee = new Employee();
2.  employee.setFirstName("James");
3.
4.  String SELECT_BY_ID = "SELECT COUNT(*) FROM EMPLOYEE WHERE FIRST_
5.  NAME = :firstName";
6.
7.  SqlParameterSource namedParameters = new
8.  BeanPropertySqlParameterSource(employee);
9.  return namedParameterJdbcTemplate.queryForObject(SELECT_BY_ID,
10. namedParameters, Integer.class);
```

Note how we're now making use of the *BeanPropertySqlParameterSource* implementations instead of specifying the named parameters manually like before.

3.3. Mapping Query Results to Java Object

Another very useful feature is the ability to map query results to Java objects – by implementing the *RowMapper* interface.

For example – for every row returned by the query, Spring uses the row mapper to populate the java bean:

```
1.  public class EmployeeRowMapper implements RowMapper<Employee> {
2.      @Override
3.      public Employee mapRow(ResultSet rs, int rowNum) throws
4.  SQLException {
5.          Employee employee = new Employee();
6.
7.          employee.setId(rs.getInt("ID"));
8.          employee.setFirstName(rs.getString("FIRST_NAME"));
9.          employee.setLastName(rs.getString("LAST_NAME"));
10.         employee.setAddress(rs.getString("ADDRESS"));
11.
12.         return employee;
13.     }
14. }
```

Subsequently, we can now pass the row mapper to the query API and get fully populated Java objects:

```
1.  String query = "SELECT * FROM EMPLOYEE WHERE ID = ?";
2.  List<Employee> employees = jdbcTemplate.queryForObject(
3.      query, new Object[] { id }, new EmployeeRowMapper());
```

Spring comes with its own data exception hierarchy out of the box – with *DataAccessException* as the root exception – and it translates all underlying raw exceptions to it.

And so we keep our sanity by not having to handle low-level persistence exceptions and benefit from the fact that Spring wraps the low-level exceptions in *DataAccessException* or one of its sub-classes.

Also, this keeps the exception handling mechanism independent of the underlying database we are using.

Besides, the default *SQLErrorCodeSQLExceptionTranslator*, we can also provide our own implementation of *SQLExceptionTranslator*.

Here's a quick example of a custom implementation, customizing the error message when there is a duplicate key violation, which results in error code 23505 when using H2:

```
1.  public class CustomSQLErrorCodeTranslator extends
2.  SQLErrorCodeSQLExceptionTranslator {
3.      @Override
4.      protected DataAccessException customTranslate
5.        (String task, String sql, SQLException sqlException) {
6.          if (sqlException.getErrorCode() == 23505) {
7.              return new DuplicateKeyException(
8.                  "Custom Exception translator - Integrity constraint
9.  violation.", sqlException);
10.         }
11.         return null;
12.     }
13. }
```

To use this custom exception translator, we need to pass it to the *JdbcTemplate* by calling *setExceptionTranslator()* method:

```
1.  CustomSQLErrorCodeTranslator customSQLErrorCodeTranslator = new
2.  CustomSQLErrorCodeTranslator();
3.  jdbcTemplate.setExceptionTranslator(customSQLErrorCodeTranslator);
```

SimpleJdbc classes provide an easy way to configure and execute SQL statements. These classes use database metadata to build basic queries. *SimpleJdbcInsert* and *SimpleJdbcCall* classes provide an easier way to execute insert and stored procedure calls.

5.1. *SimpleJdbcInsert*

Let's take a look at executing simple insert statements with minimal configuration.

The INSERT statement is generated based on the configuration of *SimpleJdbcInsert* and all we need is to provide the Table name, Column names and values.

First, let's create a *SimpleJdbcInsert*:

```
1. SimpleJdbcInsert simpleJdbcInsert = new SimpleJdbcInsert(dataSource).
2. withTableName("EMPLOYEE");
```

Next, let's provide the Column names and values, and execute the operation

```
1. public int addEmplyee(Employee emp) {
2.     Map<String, Object> parameters = new HashMap<String, Object>();
3.     parameters.put("ID", emp.getId());
4.     parameters.put("FIRST_NAME", emp.getFirstName());
5.     parameters.put("LAST_NAME", emp.getLastName());
6.     parameters.put("ADDRESS", emp.getAddress());
7.
8.     return simpleJdbcInsert.execute(parameters);
9. }
```

Further, to allow the **database to generate the primary key**, we can make use of the *executeAndReturnKey()* API; we'll also need to configure the actual column that is auto-generated:

```
1.  SimpleJdbcInsert simpleJdbcInsert = new SimpleJdbcInsert(dataSource)
2.  .withTableName("EMPLOYEE")
3.
4.  .usingGeneratedKeyColumns("ID");
5.
6.  Number id = simpleJdbcInsert.executeAndReturnKey(parameters);
7.  System.out.println("Generated id - " + id.longValue());
```

Finally – we can also pass in this data by using the *BeanPropertySqlParameterSource* and *MapSqlParameterSource*.

5.2. Stored Procedures with *SimpleJdbcCall*

Also, let's take a look at executing stored procedures – we'll make use of the *SimpleJdbcCall abstraction:*

```
1.  SimpleJdbcCall simpleJdbcCall = new SimpleJdbcCall(dataSource)
2.  .withProcedureName("READ_EMPLOYEE");
3.
4.  public Employee getEmployeeUsingSimpleJdbcCall(int id) {
5.      SqlParameterSource in = new MapSqlParameterSource().
6.  addValue("in_id", id);
7.      Map<String, Object> out = simpleJdbcCall.execute(in);
8.
9.      Employee emp = new Employee();
10.     emp.setFirstName((String) out.get("FIRST_NAME"));
11.     emp.setLastName((String) out.get("LAST_NAME"));
12.
13.     return emp;
14. }
```

Another simple use case – batching multiple operations together.

6.1. Basic batch operations using *JdbcTemplate*

Using *JdbcTemplate*, batch operations can be executed via the *batchUpdate()* API.

The interesting part here is the concise but highly useful *BatchPreparedStatementSetter* implementation:

```
1.  public int[] batchUpdateUsingJdbcTemplate(List<Employee> employees) {
2.      return jdbcTemplate.batchUpdate("INSERT INTO EMPLOYEE VALUES (?, ?,
3.  ?, ?)",
4.              new BatchPreparedStatementSetter() {
5.                  @Override
6.                  public void setValues(PreparedStatement ps, int i) throws
7.  SQLException {
8.                      ps.setInt(1, employees.get(i).getId());
9.                      ps.setString(2, employees.get(i).getFirstName());
10.                     ps.setString(3, employees.get(i).getLastName());
11.                     ps.setString(4, employees.get(i).getAddress();
12.                 }
13.                 @Override
14.                 public int getBatchSize() {
15.                     return 50;
16.                 }
17.             });
18. }
```

We also have the option of batching operations with the *NamedParameterJdbcTemplate – batchUpdate()* API.

This API is simpler than the previous one – no need to implement any extra interfaces to set the parameters, as it has an internal prepared statement setter to set the parameter values.

Instead, the parameter values can be passed to the *batchUpdate()* method as an array of *SqlParameterSource*.

```
1.  SqlParameterSource[] batch = SqlParameterSourceUtils.
2.  createBatch(employees.toArray());
3.  int[] updateCounts = namedParameterJdbcTemplate.batchUpdate(
4.      “INSERT INTO EMPLOYEE VALUES (:id, :firstName, :lastName,
5.  :address)”, batch);
6.  return updateCounts;
```

Spring Boot provides a starter *spring-boot-starter-jdbc* for using JDBC with relational databases. As with every Spring Boot starter, this one also helps us in getting our application up and running quickly.

7.1. Maven Dependency

We'll need the *spring-boot-starter-jdbc dependency* as the primary one as well as a dependency for the database that we'll be using. In our case, this is *MySQL*:

```
<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>
```

7.2. Configuration

Spring Boot configures the data source automatically for us. We just need to provide the properties in a *propertiesfile*:

```
spring.datasource.url=jdbc:mysql://localhost:3306/springjdbc
spring.datasource.username=guest_user
spring.datasource.password=guest_password
```

That's it, just by doing these configurations only, our application is up and running and we can use it for other database operations.

The explicit configuration we saw in the previous section for a standard Spring application is now included as part of Spring Boot auto-configuration.

8. Conclusion

In this chapter, we looked at the JDBC abstraction in the Spring Framework, covering the various capabilities provided by Spring JDBC with practical examples.

Also, we looked into how we can quickly get started with Spring JDBC using a Spring Boot JDBC starter.

The source code for the examples is available over on GitHub.