Spring Labs

# Using JPA

In this lab you will start using JPA. First by using JPA standalone to explore the API and later integrated with Spring.

## Step 1 - Project Configuration and first test

Check the solution with your own code.

## Step 2 - Finding books

Add a new test method to test retrieval of a book using the find method. The tables are recreated before every test, so you can be sure to have a clean database in every test. This does mean the table is empty! Insert some books to be able to find books. Your test could look as follows.

@Test

public void testFind() throws Exception {

createTestBooks();

Book book = em.find(Book.class, 1L);

assertNotNull(book);

assertThat(book.getTitle(), is("Angels and demons"));

}

private void createTestBooks() {

em.getTransaction().begin();

em.persist(new Book("Angels and demons"));

em.persist(new Book("Digital Fortress"));

em.persist(new Book("The Da Vinci code"));

em.persist(new Book("The Lost Symbol"));

em.persist(new Book("Deception Point"));

em.getTransaction().commit();

em.close();

em = emf.createEntityManager();

}

## Step 3 - Editing managed books

Create another test that test if managed books can be edited. Always use a new EntityManager and transaction to test for database changes!

## Step 4 - Editing detached books

## Retrieve a book using the find method and detach it by closing the EntityManager. Try if changes to the instance are persisted to the database. Now use the merge method to synchronize the changes with the database.

## Step 5 - Listing books

Write a simple query to select all books.

## Step 6 - Mapping books

Add the following fields to Book:

-Date releaseDate (Date in the database)

-Enum Category

-String summary (should be a longtext in the datase)

Also make the title unique.

Add mapping configuration to make sure that the correct datatypes are used in the database.

## Step 7 - Deleting books

Use the remove method to remove a book.

## Step 8 - One to One Promotion

Create a new class Promotion. A promotion can be a temporarily lowered price for a limited amount of time. Add the following properties.

-String description

-BigDecimal newPrice

-Date beginDate

-Date endDate

Map a bi-directional relationship between Book and Promotion. Test if saving two related entities and subsequent fetching from both sides work. Don’t forget to close the entitymanager. Use the find method to get a managed instance from the entitymanager. Use cascading to be able to save a new book with a new promotion without persisting the promotion explicitly. Write tests to test bi-directional behavior and to test cascading remove.

## Step 9 - One to Many reviews

Create a new class Review and map a bi-directional one-to-many relation with Book. Add the following properties to Review.

-String reviewerName

-Date reviewDate

-int rating

-String text

Test adding, cascading, lazy loading and bi-directional behavior. Also test cascading remove. Create a separate test to test a join fetch query for explicit eager loading.

## Step 10 - Many to Many authors

Create a new class Author and configure a bi-directional Many-to-Many relation with Book with cascading persist. Test if persisting and retrieval works correctly.

## Step 11 - Inheritance and ElementCollections

Create a new class EBook and a new enum EBookFormat. The EBookFormat represents formats such as EPUB and PDF. An EBook is a subclass of Book and contains a List of available EBookFormats. To map the List you’ll need an @ElementCollection mapping. Test if EBooks can be persisted correctly.

## Step 12 - Details embeddable

Create a new class Details with the following properties.

-int pages

-int isbn10

-int isbn13

-String language

Make Details @Embeddable and add a reference to it in Book. Make sure you initialize an empty details by default in Book, otherwise it’s impossible to save a book without Details. Create a test to check if Details are persisted correctly on a Book.

## Step 13 - Query for books written by more than one author

Write and test a JPQL query to retrieve books that are written by at least two authors. Don’t forget to insert some test data first!

**Step 14 - Query for the average rating for each Author**

Write and test a JPQL query to retrieve the average rating for each author. The results should be returned as a value object Rating.

public class Rating {

private String authorName;

private double avgRating;

public Rating(String authorName, double avgRating) {

this.authorName = authorName;

this.avgRating = avgRating;

}

public String getAuthorName() {

return authorName;

}

public double getAvgRating() {

return avgRating;

}

}

Don’t forget to add some authors and ratings before executing the query!

## Step 15 - Criteria API

Write and test a method that accepts a title and author name filter. If both arguments are empty, all books should be returned, including books without an author. If both are non-empty, an AND filter should be used and if only one argument is non-empty only that field should be filtered. The method should use the Type Safe Criteria API to build this dynamic query. Order results ascending by title.

## Step 16 - Bean validation

Add bean validation to the Book class to make sure books always have a title. Write a test that expects a ConstraintViolationException.

## Step 17 - Custom validator

## Write a new bean validator annotation and implementation class to disallow certain words in a title. The annotation should be configurable:

@NotExplicit(filter = {"sex", "drugs", "rock & roll"})

private String title;

# Spring JPA

Besides the persistence.xml configuration you’ll also need some Spring configuration now. First of all you’ll need a database connection. Open dao-context.xml and add the following configuration.

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

<context:property-placeholder location="classpath:jdbc.properties"/>

The property-placeholder reads the jdbc.properties file for the actual connection properties.

Now create a transaction manager that can be used with JPA.

<bean id="transactionManager" class="...JpaTransactionManager">

<property name="entityManagerFactory" ref="myEmf" />

</bean>

<bean id="myEmf" class="...LocalContainerEntityManagerFactoryBean">

<property name="dataSource" ref="dataSource" />

</bean>

That’s all configuration needed for JPA/Hibernate. The EntityManagerFactory could now be used to create EntityManager instances, or the @PersistenceContext annotation can be used to let Spring create shared transaction scoped EntityManagers.

There’s still a problem though. With the current configuration the transaction manager configured for JPA is not available for JDBC. This means it’s not possible to run JDBC and JPA code in the same transaction. That also means you can’t use countRowsInTable in your tests for example. With some additional configuration you can enable JDBC to use JPA transactions. You’ll need to configure a JpaDialect for this to work. The full configuration is as follows.

<bean id="transactionManager" class="...JpaTransactionManager">

<property name="entityManagerFactory" ref="myEmf" />

</bean>

<bean id="jpaDialect" class="...HibernateJpaDialect"/>

<bean id="myEmf" class="...LocalContainerEntityManagerFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="jpaDialect" ref="jpaDialect"/>

</bean>

## Step 2 - Implement a JPA Dao

You’ll create a dao for managing books. The Book entity from the previous lab can be used again.

Implement the BookDao interface completely. When working with JPA you’ll need an EntityManager. Spring can inject a transaction scoped EntityManager into a bean. Based on the EntityManager you can easily query for Entities. Don’t forget to make the dao implementation a Spring bean and transactional.

@PersistenceContext

EntityManager em;

@Override

@SuppressWarnings("unchecked")

@Transactional(readOnly=true)

public List<Book> listBooks() {

Query q = em.createQuery("select b from Book b");

return q.getResultList();

}

## Step 3 - Testing the Dao

Create a new Test Case class for the BookDao. Extend from AbstractTransactionalJUnit4SpringContextTests to bootstrap the container and activate transaction management in the tests.

@ContextConfiguration("classpath:dao-context.xml")

public class JpaBookDaoImplTest extends AbstractTransactionalJUnit4SpringContextTests {

First you need some test data. Create another class TestDataInserter in the test folder. Add a method that adds a few books using JPA. This method should first delete all existing books. Now there is a small problem. If the test data is re-inserted before each test the generated ids are incremented each test. You can prevent this by resetting the index before each test.

private void removeBooks() {

em.createQuery("delete from Book b").executeUpdate();

simpleJdbcTemplate.update("ALTER TABLE Book AUTO\_INCREMENT = 1");

}

Now write a test for each DAO method. Remember that the transaction is rolled back after each test so you should not see any changes in the database. That also means that some JPA operations, such as a remove, should be flushed explicitly to execute. This can be done in the test, the production code should not change.

@Test

public void testRemoveBookById() throws Exception {

bookDao.removeBook(1);

em.flush();

assertThat(countRowsInTable("book"), is(4));

}

## Step 4 - Simple JDBC Template

Create a new class BookStatsDao. This class uses a SimpleJdbcTemplate to do some “advanced” queries. Add a method to count books. Remember to use JDBC, not JPA! Test the method in a new test method.