Using JPA and JTA with Spring

When building a web application, we will sooner or later need to somehow store data entered by users and retrieve it later. In most cases the best place to keep this data is a database because it additionally provides many useful features including transactions.

Therefore, in this article I would like to show how to extend our previous Spring MVC application to use JPA and JTA to access database and manage transactions. The configuration details strongly depend on the database and application server being used. In our case it will be Oracle 11gR2 database and JBoss 7 Application Server.

Configuring entity manager factory

Before we start we have to configure entity manager factory in Spring. There are three different ways to do this:

- creating *LocalEntityManagerFactoryBean*
- obtaining *EntityManagerFactory* from JNDI
- creating *LocalContainerEntityManagerFactoryBean*

The first and the second option have several limitations so I will concentrate only on the last one which provides full JPA capabilities in a Spring-based application.

First, we have to create instance of *LocalContainerEntityManagerFactoryBean* in our Spring configuration file:

```
<bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFact
```

```
oryBean">
    <property name="jtaDataSource" ref="dataSource" />
    <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
</bean>
```

In property *jtaDataSource* we refer to JTA data source configured in our application server and exposed via JNDI with name *java:/orcl*:

```
<jee:jndi-lookup id="dataSource" jndi-name="java:/orcl" />
```

The second property *jpaVendorAdapter* gives us possibility to configure options specific to JPA provider implementation. There are several adapters to choose:

- HibernateJpaVendorAdapter
- OpenJpaVendorAdapter
- EclipseLinkJpaVendorAdapter
- TopLinkJpaVendorAdapter

but we choose Hibernate because it is available by default in JBoss:

```
<bean id="jpaVendorAdapter"
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapte
r">
    <property name="database" value="ORACLE" />
    <property name="showSql" value="true" />
    <property name="generateDdl" value="true" />
</bean>
```

In property *database* we specify that we use Oracle database, then we inform JPA implementation to print issued SQL commands to the server log and to generate necessary objects (like tables) in the database. Please, note that the last option cannot be used in the production code because it may drop already existing tables in the database. We use it only to simplify the example.

Configuring persistence.xml

The last step to configure entity manager factory is to create *META-INF/persistence.xml* file with our persistence unit:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="mainPU" transaction-type="JTA">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>com.example.springjpa.Person</class>
    <properties>
      <property name="hibernate.dialect"
value="org.hibernate.dialect.Oracle10gDialect"/>
      <property name="hibernate.transaction.jta.platform"
value="org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform"/>
    </properties>
  </persistence-unit>
</persistence>
```

We have one persistence unit with name *mainPU* which supports JTA transactions. We configure Hibernate specific persistence provider class and we configure 2 properties required by it. As you could see the exact values are very database and application server specific. The rest of the properties is set by Spring via *jpaVendorAdapter* bean created before. At last we specify a single class *com.example.springjpa.Person* which can be persisted by this persistence unit.

Configuring transaction support

Because we plan to use JTA transactions and declare them using annotations, we have to add two additional lines to our Spring configuration:

```
<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager"
/>
<tx:annotation-driven transaction-manager="transactionManager"/>
```

The first one informs Spring to instantiate JTA-specific *JtaTransationManager* transaction manager which uses JTA implementation provided by the application server. The second line tells Spring to scan all classes for @*Transactional* annotation on a class or method level and associate them with given transaction manager.

Enable injecting with @PersistenceContext

If we want to inject instances of *EntityManager* using @*PersistenceContext* annotation, we have to enable annotation bean processor in Spring configuration:

```
<bean
class="org.springframework.orm.jpa.support.PersistenceAnnotationBe
anPostProcessor" />
```

Usually this line is optional because a default *PersistenceAnnotationBeanPostProcessor* will be registered by the <*context:annotation-config*> and <*context:component-scan*> XML tags.

Using annotations

After we have finished the configuration of persistence, we can add standard JPA annotations to our *Person* entity class:

```
package com.example.springjpa;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
@Entity
@Table(name = "springjpa_person")
@NamedQuery(name = "Person.selectAll", query = "select o from
Person o")
public class Person implements Serializable {
    private static final long serialVersionUID = 3297423984732894L;
    @Id
    @GeneratedValue
    private int id;
    private String firstName;
    private String lastName;
    private Integer age;
    // constructor, getters and setters
}
```

```
package com.example.springjpa;
import java.io.Serializable;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
@Repository@Transactional
public class PersonList implements Serializable {
    private static final long serialVersionUID = 324589274837L;
    @PersistenceContext
    private EntityManager em;
    @Transactional
    public void addPerson(Person person) {
        em.persist(person);
    }
    @Transactional
    public List<Person> getAll() {
        TypedQuery<Person> query =
em.createNamedQuery("Person.selectAll", Person.class);
        return query.getResultList();
    }
}
```

In this class we obtain reference to correct *EntityManager* instance using standard *@PersistenceContext* annotation. Additionally, both methods of this class are annotated with *@Transactional* to inform Spring that these methods should be executed in a transaction.

Conclusion

The configuration of JPA and JTA in Spring is not very difficult but require setting several parameters specific to the environment (database, persistence provider and application server) in XML configuration files. The rest of the code is totally independent of it so changes necessary to switch to the other database or application server are limited to several lines of configuration.

The complete source code for the example can be found at GitHub.