Chapter 12. Object Relational Mapping (ORM) data access

12.1. Introduction

The Spring Framework provides integration with *Hibernate*, *JDO*, *Oracle TopLink*, *iBATIS SQL Maps* and *JPA*: in terms of resource management, DAO implementation support, and transaction strategies. For example for Hibernate, there is first-class support with lots of IoC convenience features, addressing many typical Hibernate integration issues. All of these support packages for O/R (Object Relational) mappers comply with Spring's generic transaction and DAO exception hierarchies. There are usually two integration styles: either using Spring's DAO 'templates' or coding DAOs against plain Hibernate/JDO/TopLink/etc APIs. In both cases, DAOs can be configured through Dependency Injection and participate in Spring's resource and transaction management.

Spring adds significant support when using the O/R mapping layer of your choice to create data access applications. First of all, you should know that once you started using Spring's support for O/R mapping, you don't have to go all the way. No matter to what extent, you're invited to review and leverage the Spring approach, before deciding to take the effort and risk of building a similar infrastructure in-house. Much of the O/R mapping support, no matter what technology you're using may be used in a library style, as everything is designed as a set of reusable JavaBeans. Usage inside a Spring IoC container does provide additional benefits in terms of ease of configuration and deployment; as such, most examples in this section show configuration inside a Spring container.

Some of the benefits of using the Spring Framework to create your ORM DAOs include:

- *Ease of testing.* Spring's IoC approach makes it easy to swap the implementations and config locations of Hibernate `SessionFactory` instances, JDBC `DataSource` instances, transaction managers, and mapper object implementations (if needed). This makes it much easier to isolate and test each piece of persistence-related code in isolation.
- *Common data access exceptions.* Spring can wrap exceptions from your O/R mapping tool of choice, converting them from proprietary (potentially checked) exceptions to a common runtime DataAccessException hierarchy. This allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches/throws, and exception declarations. You can still trap and handle exceptions anywhere you need to. Remember that JDBC exceptions (including DB specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.
- *General resource management.* Spring application contexts can handle the location and configuration of Hibernate `SessionFactory` instances, JDBC `DataSource` instances, iBATIS SQL Maps configuration objects, and other related resources. This makes these values easy to manage and change. Spring offers efficient, easy and safe handling of persistence resources. For example: related code using Hibernate generally needs to use the same Hibernate `Session` for efficiency and proper transaction handling. Spring makes it easy to transparently create and bind a `Session` to the current thread, either by using an explicit 'template' wrapper class at the Java code level or by exposing a current `Session` through the Hibernate `SessionFactory` (for DAOs based on plain Hibernate API). Thus Spring solves many of the issues that repeatedly arise from typical Hibernate usage, for any transaction environment (local or JTA).

- *Integrated transaction management.* Spring allows you to wrap your O/R mapping code with either a declarative, AOP style method interceptor, or an explicit 'template' wrapper class at the Java code level. In either case, transaction semantics are handled for you, and proper transaction handling (rollback, etc) in case of exceptions is taken care of. As discussed below, you also get the benefit of being able to use and swap various transaction managers, without your Hibernate/JDO related code being affected: for example, between local transactions and JTA, with the same full services (such as declarative transactions) available in both scenarios. As an additional benefit, JDBC-related code can fully integrate transactionally with the code you use to do O/R mapping. This is useful for data access that's not suitable for O/R mapping, such as batch processing or streaming of BLOBs, which still needs to share common transactions with ORM operations.

The PetClinic sample in the Spring distribution offers alternative DAO implementations and application context configurations for JDBC, Hibernate, Oracle TopLink, and JPA. PetClinic can therefore serve as working sample app that illustrates the use of Hibernate, TopLink and JPA in a Spring web application. It also leverages declarative transaction demarcation with different transaction strategies.

The JPetStore sample illustrates the use of iBATIS SQL Maps in a Spring environment. It also features two web tier versions: one based on Spring Web MVC, one based on Struts.

Beyond the samples shipped with Spring, there are a variety of Spring-based O/R mapping samples provided by specific vendors: for example, the JDO implementations JPOX (http://www.jpox.org/) and Kodo (http://www.bea.com/kodo/).

12.2. Hibernate

We will start with a coverage of Hibernate 3 in a Spring environment, using it to demonstrate the approach that Spring takes towards integrating O/R mappers. This section will cover many issues in detail and show different variations of DAO implementations and transaction demarcations. Most of these patterns can be directly translated to all other supported ORM tools. The following sections in this chapter will then cover the other ORM technologies, showing briefer examples there.

The following discussion focuses on Hibernate 3: this is the current major production ready version of Hibernate. Hibernate 2.x, which has been supported in Spring since its inception continues to be supported... it is just that the following examples all use the Hibernate 3 classes and configuration. All of this can (pretty much) be applied to Hibernate 2.x as-is, using the analogous Hibernate 2.x support package: `org.springframework.orm.hibernate`, mirroring `org.springframework.orm.hibernate3` with analogous support classes for Hibernate 2.x. Furthermore, all references to the `org.hibernate` package need to be replaced with `net.sf.hibernate`, following the root package change in Hibernate 3. Simply adapt the package names (as used in the examples) accordingly.

12.2.1. Resource management

Typical business applications are often cluttered with repetitive resource management code. Many projects try to invent their own solutions for this issue, sometimes sacrificing proper handling of failures for programming convenience. Spring advocates strikingly simple solutions for proper resource handling, namely IoC via templating; for example infrastructure classes with callback interfaces, or applying AOP interceptors. The infrastructure cares for proper resource handling, and for appropriate conversion of specific API exceptions to an unchecked infrastructure exception hierarchy. Spring

introduces a DAO exception hierarchy, applicable to any data access strategy. For direct JDBC, the `JdbcTemplate` class mentioned in a previous section cares for connection handling, and for proper conversion of `SQLException` to the `DataAccessException` hierarchy, including translation of database-specific SQL error codes to meaningful exception classes. It supports both JTA and JDBC transactions, via respective Spring transaction managers.

Spring also offers Hibernate and JDO support, consisting of a `HibernateTemplate` / `JdoTemplate` analogous to `JdbcTemplate`, a `HibernateInterceptor` / `JdoInterceptor`, and a Hibernate / JDO transaction manager. The major goal is to allow for clear application layering, with any data access and transaction technology, and for loose coupling of application objects. No more business service dependencies on the data access or transaction strategy, no more hard-coded resource lookups, no more hard-to-replace singletons, no more custom service registries. One simple and consistent approach to wiring up application objects, keeping them as reusable and free from container dependencies as possible. All the individual data access features are usable on their own but integrate nicely with Spring's application context concept, providing XML-based configuration and cross-referencing of plain JavaBean instances that don't need to be Spring-aware. In a typical Spring app, many important objects are JavaBeans: data access templates, data access objects (that use the templates), transaction managers, business services (that use the data access objects and transaction managers), web view resolvers, web controllers (that use the business services),and so on.

12.2.2. `SessionFactory` setup in a Spring container

To avoid tying application objects to hard-coded resource lookups, Spring allows you to define resources like a JDBC `DataSource` or a Hibernate `SessionFactory` as beans in an application context. Application objects that need to access resources just receive references to such pre-defined instances via bean references (the DAO definition in the next section illustrates this). The following excerpt from an XML application context definition shows how to set up a JDBC `DataSource` and a Hibernate `SessionFactory` on top of it:

```
<beans>

  <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
  </bean>

  <bean id="mySessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.HSQLDialect
      </value>
    </property>
```

```
  </bean>

</beans>
```

Note that switching from a local Jakarta Commons DBCP `BasicDataSource` to a JNDI-located `DataSource` (usually managed by an application server) is just a matter of configuration:

```
<beans>

  <bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/myds"/>
  </bean>

</beans>
```

You can also access a JNDI-located `SessionFactory`, using Spring's `JndiObjectFactoryBean` to retrieve and expose it. However, that is typically not common outside of an EJB context.

12.2.3. The `HibernateTemplate`

The basic programming model for templating looks as follows, for methods that can be part of any custom data access object or business service. There are no restrictions on the implementation of the surrounding object at all, it just needs to provide a Hibernate `SessionFactory`. It can get the latter from anywhere, but preferably as bean reference from a Spring IoC container - via a simple `setSessionFactory(..)` bean property setter. The following snippets show a DAO definition in a Spring container, referencing the above defined `SessionFactory`, and an example for a DAO method implementation.

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

</beans>

public class ProductDaoImpl implements ProductDao {

    private HibernateTemplate hibernateTemplate;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.hibernateTemplate = new HibernateTemplate(sessionFactory);
    }

    public Collection loadProductsByCategory(String category) throws
DataAccessException {
        return this.hibernateTemplate.find("from test.Product product where
product.category=?", category);
    }
}
```

The `HibernateTemplate` class provides many methods that mirror the methods exposed on the Hibernate `Session` interface, in addition to a number of convenience methods such as the one shown

above. If you need access to the `Session` to invoke methods that are not exposed on the `HibernateTemplate`, you can always drop down to a callback-based approach like so.

```
public class ProductDaoImpl implements ProductDao {

    private HibernateTemplate hibernateTemplate;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.hibernateTemplate = new HibernateTemplate(sessionFactory);
    }

    public Collection loadProductsByCategory(final String category) throws
DataAccessException {
        return this.hibernateTemplate.execute(new HibernateCallback() {
            public Object doInHibernate(Session session) {
                Criteria criteria = session.createCriteria(Product.class);
                criteria.add(Expression.eq("category", category));
                criteria.setMaxResults(6);
                return criteria.list();
            }
        };
    }
}
```

A callback implementation effectively can be used for any Hibernate data access. `HibernateTemplate` will ensure that `Session` instances are properly opened and closed, and automatically participate in transactions. The template instances are thread-safe and reusable, they can thus be kept as instance variables of the surrounding class. For simple single step actions like a single find, load, saveOrUpdate, or delete call, `HibernateTemplate` offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenient `HibernateDaoSupport` base class that provides a `setSessionFactory(..)` method for receiving a `SessionFactory`, and `getSessionFactory()` and `getHibernateTemplate()` for use by subclasses. In combination, this allows for very simple DAO implementations for typical requirements:

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws
DataAccessException {
        return this.getHibernateTemplate().find(
            "from test.Product product where product.category=?", category);
    }
}
```

12.2.4. Implementing Spring-based DAOs without callbacks

As alternative to using Spring's `HibernateTemplate` to implement DAOs, data access code can also be written in a more traditional fashion, without wrapping the Hibernate access code in a callback, while still respecting and participating in Spring's generic `DataAccessException` hierarchy. The `HibernateDaoSupport` base class offers methods to access the current transactional `Session` and to convert exceptions in such a scenario; similar methods are also available as static helpers on the `SessionFactoryUtils` class. Note that such code will usually pass '`false`' as the value of the `getSession(..)` methods '`allowCreate`' argument, to enforce running within a transaction

(which avoids the need to close the returned `Session`, as its lifecycle is managed by the transaction).

```
public class HibernateProductDao extends HibernateDaoSupport implements ProductDao
{

    public Collection loadProductsByCategory(String category) throws
DataAccessException, MyException {
        Session session = getSession(false);
        try {
            Query query = session.createQuery("from test.Product product where
product.category=?");
            query.setString(0, category);
            List result = query.list();
            if (result == null) {
                throw new MyException("No search results.");
            }
            return result;
        }
        catch (HibernateException ex) {
            throw convertHibernateAccessException(ex);
        }
    }
}
```

The advantage of such direct Hibernate access code is that it allows *any* checked application exception to be thrown within the data access code; contrast this to the `HibernateTemplate` class which is restricted to throwing only unchecked exceptions within the callback. Note that you can often defer the corresponding checks and the throwing of application exceptions to after the callback, which still allows working with `HibernateTemplate`. In general, the `HibernateTemplate` class' convenience methods are simpler and more convenient for many scenarios.

12.2.5. Implementing DAOs based on plain Hibernate 3 API

Hibernate 3.0.1 introduced a feature called "contextual Sessions", where Hibernate itself manages one current `Session` per transaction. This is roughly equivalent to Spring's synchronization of one Hibernate `Session` per transaction. A corresponding DAO implementation looks like as follows, based on the plain Hibernate API:

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public Collection loadProductsByCategory(String category) {
        return this.sessionFactory.getCurrentSession()
                .createQuery("from test.Product product where product.category=?")
                .setParameter(0, category)
                .list();
    }
}
```

This style is very similar to what you will find in the Hibernate reference documentation and examples,

except for holding the `SessionFactory` in an instance variable. We strongly recommend such an instance-based setup over the old-school `static` `HibernateUtil` class from Hibernate's CaveatEmptor sample application. (In general, do not keep any resources in `static` variables unless *absolutely* necessary.)

The above DAO follows the Dependency Injection pattern: it fits nicely into a Spring IoC container, just like it would if coded against Spring's `HibernateTemplate`. Of course, such a DAO can also be set up in plain Java (for example, in unit tests): simply instantiate it and call `setSessionFactory(..)` with the desired factory reference. As a Spring bean definition, it would look as follows:

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

</beans>
```

The main advantage of this DAO style is that it depends on Hibernate API only; no import of any Spring class is required. This is of course appealing from a non-invasiveness perspective, and will no doubt feel more natural to Hibernate developers.

However, the DAO throws plain `HibernateException` (which is unchecked, so does not have to be declared or caught), which means that callers can only treat exceptions as generally fatal - unless they want to depend on Hibernate's own exception hierarchy. Catching specific causes such as an optimistic locking failure is not possible without tieing the caller to the implementation strategy. This tradeoff might be acceptable to applications that are strongly Hibernate-based and/or do not need any special exception treatment.

Fortunately, Spring's `LocalSessionFactoryBean` supports Hibernate's `SessionFactory.getCurrentSession()` method for any Spring transaction strategy, returning the current Spring-managed transactional `Session` even with `HibernateTransactionManager`. Of course, the standard behavior of that method remains: returning the current `Session` associated with the ongoing JTA transaction, if any (no matter whether driven by Spring's `JtaTransactionManager`, by EJB CMT, or by JTA).

In summary: DAOs can be implemented based on the plain Hibernate 3 API, while still being able to participate in Spring-managed transactions.

12.2.6. Programmatic transaction demarcation

Transactions can be demarcated in a higher level of the application, on top of such lower-level data access services spanning any number of operations. There are no restrictions on the implementation of the surrounding business service here as well, it just needs a Spring `PlatformTransactionManager`. Again, the latter can come from anywhere, but preferably as bean reference via a `setTransactionManager(..)` method - just like the `productDAO` should be set via a `setProductDao(..)` method. The following snippets show a transaction manager and a business service definition in a Spring application context, and an example for a business method implementation.

```
<beans>
```

```
  <bean id="myTxManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="productDao" ref="myProductDao"/>
  </bean>

</beans>

public class ProductServiceImpl implements ProductService {

    private TransactionTemplate transactionTemplate;
    private ProductDao productDao;

    public void setTransactionManager(PlatformTransactionManager
transactionManager) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void increasePriceOfAllProductsInCategory(final String category) {
        this.transactionTemplate.execute(new TransactionCallbackWithoutResult() {

                public void doInTransactionWithoutResult(TransactionStatus status)
{
                    List productsToChange =
this.productDao.loadProductsByCategory(category);
                    // do the price increase...
                }
            }
        );
    }
}
```

12.2.7. Declarative transaction demarcation

Alternatively, one can use Spring's declarative transaction support, which essentially enables you to replace explicit transaction demarcation API calls in your Java code with an AOP transaction interceptor configured in a Spring container. This allows you to keep business services free of repetitive transaction demarcation code, and allows you to focus on adding business logic which is where the real value of your application lies. Furthermore, transaction semantics like propagation behavior and isolation level can be changed in a configuration file and do not affect the business service implementations.

```
<beans>

  <bean id="myTxManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>
```

```
  <bean id="myProductService"
class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="product.ProductService"/>
    <property name="target">
        <bean class="product.DefaultProductService">
            <property name="productDao" ref="myProductDao"/>
        </bean>
    </property>
    <property name="interceptorNames">
      <list>
        <value>myTxInterceptor</value> <!-- the transaction interceptor
(configured elsewhere) -->
      </list>
    </property>
  </bean>

</beans>

public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    // notice the absence of transaction demarcation code in this method
    // Spring's declarative transaction infrastructure will be demarcating
transactions on your behalf
    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDao.loadProductsByCategory(category);
        // ...
    }
}
```

Spring's `TransactionInterceptor` allows any checked application exception to be thrown with the callback code, while `TransactionTemplate` is restricted to unchecked exceptions within the callback. `TransactionTemplate` will trigger a rollback in case of an unchecked application exception, or if the transaction has been marked rollback-only by the application (via `TransactionStatus`). `TransactionInterceptor` behaves the same way by default but allows configurable rollback policies per method.

The following higher level approach to declarative transactions doesn't use the `ProxyFactoryBean`, and as such may be easier to use if you have a large number of service objects that you wish to make transactional.

Note

You are *strongly* encouraged to read the section entitled Section 9.5, “Declarative transaction management” if you have not done so already prior to continuing.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
```

```
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <!-- SessionFactory, DataSource, etc. omitted -->

  <bean id="myTxManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(*
product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

  <bean id="myProductService" class="product.SimpleProductService">
    <property name="productDao" ref="myProductDao"/>
  </bean>

</beans>
```

12.2.8. Transaction management strategies

Both `TransactionTemplate` and `TransactionInterceptor` delegate the actual transaction handling to a `PlatformTransactionManager` instance, which can be a `HibernateTransactionManager` (for a single Hibernate `SessionFactory`, using a `ThreadLocal` `Session` under the hood) or a `JtaTransactionManager` (delegating to the JTA subsystem of the container) for Hibernate applications. You could even use a custom `PlatformTransactionManager` implementation. So switching from native Hibernate transaction management to JTA, such as when facing distributed transaction requirements for certain deployments of your application, is just a matter of configuration. Simply replace the Hibernate transaction manager with Spring's JTA transaction implementation. Both transaction demarcation and data access code will work without changes, as they just use the generic transaction management APIs.

For distributed transactions across multiple Hibernate session factories, simply combine `JtaTransactionManager` as a transaction strategy with multiple `LocalSessionFactoryBean` definitions. Each of your DAOs then gets one specific `SessionFactory` reference passed into it's respective bean property. If all underlying JDBC data sources are transactional container ones, a business service can demarcate transactions across any

number of DAOs and any number of session factories without special regard, as long as it is using `JtaTransactionManager` as the strategy.

```
<beans>

  <bean id="myDataSource1" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName value="java:comp/env/jdbc/myds1"/>
  </bean>

  <bean id="myDataSource2" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/myds2"/>
  </bean>

  <bean id="mySessionFactory1"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource1"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.MySQLDialect
        hibernate.show_sql=true
      </value>
    </property>
  </bean>

  <bean id="mySessionFactory2"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource2"/>
    <property name="mappingResources">
      <list>
        <value>inventory.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.OracleDialect
      </value>
    </property>
  </bean>

  <bean id="myTxManager"
class="org.springframework.transaction.jta.JtaTransactionManager"/>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory1"/>
  </bean>

  <bean id="myInventoryDao" class="product.InventoryDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory2"/>
  </bean>

  <!-- this shows the Spring 1.x style of declarative transaction configuration
-->
  <!-- it is totally supported, 100% legal in Spring 2.x, but see also above for
```

```
the sleeker, Spring 2.0 style -->
  <bean id="myProductService"

class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="target">
      <bean class="product.ProductServiceImpl">
        <property name="productDao" ref="myProductDao"/>
        <property name="inventoryDao" ref="myInventoryDao"/>
      </bean>
    </property>
    <property name="transactionAttributes">
      <props>
        <prop key="increasePrice*">PROPAGATION_REQUIRED</prop>
        <prop key="someOtherBusinessMethod">PROPAGATION_REQUIRES_NEW</prop>
        <prop key="*">PROPAGATION_SUPPORTS,readOnly</prop>
      </props>
    </property>
  </bean>

</beans>
```

Both `HibernateTransactionManager` and `JtaTransactionManager` allow for proper JVM-level cache handling with Hibernate - without container-specific transaction manager lookup or JCA connector (as long as not using EJB to initiate transactions).

`HibernateTransactionManager` can export the JDBC `Connection` used by Hibernate to plain JDBC access code, for a specific `DataSource`. This allows for high-level transaction demarcation with mixed Hibernate/JDBC data access completely without JTA, as long as you are just accessing one database! `HibernateTransactionManager` will automatically expose the Hibernate transaction as JDBC transaction if the passed-in `SessionFactory` has been set up with a `DataSource` (through the "dataSource" property of the `LocalSessionFactoryBean` class). Alternatively, the `DataSource` that the transactions are supposed to be exposed for can also be specified explicitly, through the "dataSource" property of the `HibernateTransactionManager` class.

12.2.9. Container resources versus local resources

Spring's resource management allows for simple switching between a JNDI `SessionFactory` and a local one, without having to change a single line of application code. The decision as to whether to keep the resource definitions in the container or locally within the application, is mainly a matter of the transaction strategy being used. Compared to a Spring-defined local `SessionFactory`, a manually registered JNDI `SessionFactory` does not provide any benefits. Deploying a `SessionFactory` through Hibernate's JCA connector provides the added value of participating in the J2EE server's management infrastructure, but does not add actual value beyond that.

An important benefit of Spring's transaction support is that it isn't bound to a container at all. Configured to any other strategy than JTA, it will work in a standalone or test environment too. Especially for the typical case of single-database transactions, this is a very lightweight and powerful alternative to JTA. When using local EJB Stateless Session Beans to drive transactions, you depend both on an EJB container and JTA - even if you just access a single database anyway, and just use SLSBs for declarative transactions via CMT. The alternative of using JTA programmatically requires a J2EE environment as well. JTA does not just involve container dependencies in terms of JTA itself and

of JNDI `DataSource` instances. For non-Spring JTA-driven Hibernate transactions, you have to use the Hibernate JCA connector, or extra Hibernate transaction code with the `TransactionManagerLookup` being configured for proper JVM-level caching.

Spring-driven transactions can work with a locally defined Hibernate `SessionFactory` nicely, just like with a local JDBC `DataSource` - if accessing a single database, of course. Therefore you just have to fall back to Spring's JTA transaction strategy when actually facing distributed transaction requirements. Note that a JCA connector needs container-specific deployment steps, and obviously JCA support in the first place. This is far more hassle than deploying a simple web app with local resource definitions and Spring-driven transactions. And you often need the Enterprise Edition of your container, as for example WebLogic Express does not provide JCA. A Spring application with local resources and transactions spanning one single database will work in any J2EE web container (without JTA, JCA, or EJB) - like Tomcat, Resin, or even plain Jetty. Additionally, such a middle tier can be reused in desktop applications or test suites easily.

All things considered: if you do not use EJB, stick with local `SessionFactory` setup and Spring's `HibernateTransactionManager` or `JtaTransactionManager`. You will get all of the benefits including proper transactional JVM-level caching and distributed transactions, without any container deployment hassle. JNDI registration of a Hibernate `SessionFactory` via the JCA connector really only adds value when used in conjunction with EJBs.

12.2.10. Spurious application server warnings when using Hibernate

In some JTA environments with very strict `XADataSource` implementations -- currently only some WebLogic and WebSphere versions -- when using Hibernate configured without any awareness of the JTA `PlatformTransactionManager` object for that environment, it is possible for spurious warning or exceptions to show up in the application server log. These warnings or exceptions will say something to the effect that the connection being accessed is no longer valid, or JDBC access is no longer valid, possibly because the transaction is no longer active. As an example, here is an actual exception from WebLogic:

```
java.sql.SQLException: The transaction is no longer active - status: 'Committed'.
   No further JDBC access is allowed within this transaction.
```

This warning is easy to resolve by simply making Hibernate aware of the JTA `PlatformTransactionManager` instance, to which it will also synchronize (along with Spring). This may be done in two ways:

- If in your application context you are already directly obtaining the JTA `PlatformTransactionManager` object (presumably from JNDI via `JndiObjectFactoryBean`) and feeding it for example to Spring's `JtaTransactionManager`, then the easiest way is to simply specify a reference to this as the value of `LocalSessionFactoryBean`'s *jtaTransactionManager* property. Spring will then make the object available to Hibernate.
- More likely you do not already have the JTA `PlatformTransactionManager` instance (since Spring's `JtaTransactionManager` can find it itself) so you need to instead configure Hibernate to also look it up directly. This is done by configuring an AppServer specific `TransactionManagerLookup` class in the Hibernate configuration, as described in the Hibernate manual.

It is not necessary to read any more for proper usage, but the full sequence of events with and without

Hibernate being aware of the JTA `PlatformTransactionManager` will now be described.

When Hibernate is not configured with any awareness of the JTA `PlatformTransactionManager`, the sequence of events when a JTA transaction commits is as follows:

- JTA transaction commits
- Spring's `JtaTransactionManager` is synchronized to the JTA transaction, so it is called back via an *afterCompletion* callback by the JTA transaction manager.
- Among other activities, this can trigger a callback by Spring to Hibernate, via Hibernate's `afterTransactionCompletion` callback (used to clear the Hibernate cache), followed by an explicit `close()` call on the Hibernate Session, which results in Hibernate trying to `close()` the JDBC Connection.
- In some environments, this `Connection.close()` call then triggers the warning or error, as the application server no longer considers the `Connection` usable at all, since the transaction has already been committed.

When Hibernate is configured with awareness of the JTA `PlatformTransactionManager`, the sequence of events when a JTA transaction commits is instead as follows:

- JTA transaction is ready to commit
- Spring's `JtaTransactionManager` is synchronized to the JTA transaction, so it is called back via a *beforeCompletion* callback by the JTA transaction manager.
- Spring is aware that Hibernate itself is synchronized to the JTA transaction, and behaves differently than in the previous scenario. Assuming the Hibernate `Session` needs to be closed at all, Spring will close it now.
- JTA Transaction commits
- Hibernate is synchronized to the JTA transaction, so it is called back via an *afterCompletion* callback by the JTA transaction manager, and can properly clear its cache.

12.3. JDO

Spring supports the standard JDO 1.0/2.0 API as data access strategy, following the same style as the Hibernate support. The corresponding integration classes reside in the `org.springframework.orm.jdo` package.

12.3.1. `PersistenceManagerFactory` setup

Spring provides a `LocalPersistenceManagerFactoryBean` class that allows for defining a local JDO `PersistenceManagerFactory` within a Spring application context:

```
<beans>

  <bean id="myPmf"
class="org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean">
    <property name="configLocation" value="classpath:kodo.properties"/>
  </bean>

</beans>
```

Alternatively, a `PersistenceManagerFactory` can also be set up through direct instantiation of a `PersistenceManagerFactory` implementation class. A JDO `PersistenceManagerFactory` implementation class is supposed to follow the JavaBeans pattern, just like a JDBC `DataSource` implementation class, which is a natural fit for a Spring bean definition. This setup style usually supports a Spring-defined JDBC `DataSource`, passed into the "connectionFactory" property. For example, for the open source JDO implementation JPOX (http://www.jpox.org):

```
<beans>

 <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>

 <bean id="myPmf" class="org.jpox.PersistenceManagerFactoryImpl"
destroy-method="close">
   <property name="connectionFactory" ref="dataSource"/>
   <property name="nontransactionalRead" value="true"/>
 </bean>

</beans>
```

A JDO `PersistenceManagerFactory` can also be set up in the JNDI environment of a J2EE application server, usually through the JCA connector provided by the particular JDO implementation. Spring's standard `JndiObjectFactoryBean` can be used to retrieve and expose such a `PersistenceManagerFactory`. However, outside an EJB context, there is often no compelling benefit in holding the `PersistenceManagerFactory` in JNDI: only choose such setup for a good reason. See "container resources versus local resources" in the Hibernate section for a discussion; the arguments there apply to JDO as well.

12.3.2. `JdoTemplate` and `JdoDaoSupport`

Each JDO-based DAO will then receive the `PersistenceManagerFactory` through dependency injection. Such a DAO could be coded against plain JDO API, working with the given `PersistenceManagerFactory`, but will usually rather be used with the Spring Framework's `JdoTemplate`:

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

</beans>

public class ProductDaoImpl implements ProductDao {

    private JdoTemplate jdoTemplate;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
```

```
        this.jdoTemplate = new JdoTemplate(pmf);
    }

    public Collection loadProductsByCategory(final String category) throws
DataAccessException {
        return (Collection) this.jdoTemplate.execute(new JdoCallback() {
            public Object doInJdo(PersistenceManager pm) throws JDOException {
                Query query = pm.newQuery(Product.class, "category = pCategory");
                query.declareParameters("String pCategory");
                List result = query.execute(category);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

A callback implementation can effectively be used for any JDO data access. `JdoTemplate` will ensure that `PersistenceManager`s are properly opened and closed, and automatically participate in transactions. The template instances are thread-safe and reusable, they can thus be kept as instance variables of the surrounding class. For simple single-step actions such as a single `find`, `load`, `makePersistent`, or `delete` call, `JdoTemplate` offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenient `JdoDaoSupport` base class that provides a `setPersistenceManagerFactory(..)` method for receiving a `PersistenceManagerFactory`, and `getPersistenceManagerFactory()` and `getJdoTemplate()` for use by subclasses. In combination, this allows for very simple DAO implementations for typical requirements:

```
public class ProductDaoImpl extends JdoDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws
DataAccessException {
        return getJdoTemplate().find(
            Product.class, "category = pCategory", "String category", new Object[]
{category});
    }
}
```

As alternative to working with Spring's `JdoTemplate`, you can also code Spring-based DAOs at the JDO API level, explicitly opening and closing a `PersistenceManager`. As elaborated in the corresponding Hibernate section, the main advantage of this approach is that your data access code is able to throw checked exceptions. `JdoDaoSupport` offers a variety of support methods for this scenario, for fetching and releasing a transactional `PersistenceManager` as well as for converting exceptions.

12.3.3. Implementing DAOs based on the plain JDO API

DAOs can also be written against plain JDO API, without any Spring dependencies, directly using an injected `PersistenceManagerFactory`. A corresponding DAO implementation looks like as follows:

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;
```

```
    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(String category) {
        PersistenceManager pm =
this.persistenceManagerFactory.getPersistenceManager();
        try {
            Query query = pm.newQuery(Product.class, "category = pCategory");
            query.declareParameters("String pCategory");
            return query.execute(category);
        }
        finally {
          pm.close();
        }
    }
}
```

As the above DAO still follows the Dependency Injection pattern, it still fits nicely into a Spring container, just like it would if coded against Spring's `JdoTemplate`:

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

</beans>
```

The main issue with such DAOs is that they always get a new `PersistenceManager` from the factory. To still access a Spring-managed transactional `PersistenceManager`, consider defining a `TransactionAwarePersistenceManagerFactoryProxy` (as included in Spring) in front of your target `PersistenceManagerFactory`, passing the proxy into your DAOs.

```
<beans>

  <bean id="myPmfProxy"

class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy"
>
    <property name="targetPersistenceManagerFactory" ref="myPmf"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmfProxy"/>
  </bean>

</beans>
```

Your data access code will then receive a transactional `PersistenceManager` (if any) from the `PersistenceManagerFactory.getPersistenceManager()` method that it calls. The latter method call goes through the proxy, which will first check for a current transactional `PersistenceManager` before getting a new one from the factory. `close()` calls on the `PersistenceManager` will be ignored in case of a transactional `PersistenceManager`.

If your data access code will always run within an active transaction (or at least within active transaction synchronization), it is safe to omit the `PersistenceManager.close()` call and thus the entire `finally` block, which you might prefer to keep your DAO implementations concise:

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(String category) {
        PersistenceManager pm =
this.persistenceManagerFactory.getPersistenceManager();
        Query query = pm.newQuery(Product.class, "category = pCategory");
        query.declareParameters("String pCategory");
        return query.execute(category);
    }
}
```

With such DAOs that rely on active transactions, it is recommended to enforce active transactions through turning `TransactionAwarePersistenceManagerFactoryProxy`'s "allowCreate" flag off:

```
<beans>

  <bean id="myPmfProxy"

class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy"
>
    <property name="targetPersistenceManagerFactory" ref="myPmf"/>
    <property name="allowCreate" value="false"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmfProxy"/>
  </bean>

</beans>
```

The main advantage of this DAO style is that it depends on JDO API only; no import of any Spring class is required. This is of course appealing from a non-invasiveness perspective, and might feel more natural to JDO developers.

However, the DAO throws plain `JDOException` (which is unchecked, so does not have to be declared or caught), which means that callers can only treat exceptions as generally fatal - unless they want to depend on JDO's own exception structure. Catching specific causes such as an optimistic locking failure is not possible without tying the caller to the implementation strategy. This tradeoff might be acceptable to applications that are strongly JDO-based and/or do not need any special exception treatment.

In summary: DAOs can be implemented based on plain JDO API, while still being able to participate in Spring-managed transactions. This might in particular appeal to people already familiar with JDO, feeling more natural to them. However, such DAOs will throw plain `JDOException`; conversion to Spring's `DataAccessException` would have to happen explicitly (if desired).

12.3.4. Transaction management

To execute service operations within transactions, you can use Spring's common declarative transaction facilities. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="
   http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
   http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <bean id="myTxManager"
class="org.springframework.orm.jdo.JdoTransactionManager">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(*
product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

</beans>
```

Note that JDO requires an active transaction when modifying a persistent object. There is no concept like a non-transactional flush in JDO, in contrast to Hibernate. For this reason, the chosen JDO implementation needs to be set up for a specific environment: in particular, it needs to be explicitly set up for JTA synchronization, to detect an active JTA transaction itself. This is not necessary for local transactions as performed by Spring's `JdoTransactionManager`, but it is necessary for participating in JTA transactions (whether driven by Spring's `JtaTransactionManager` or by EJB CMT / plain JTA).

`JdoTransactionManager` is capable of exposing a JDO transaction to JDBC access code that accesses the same JDBC `DataSource`, provided that the registered `JdoDialect` supports retrieval of the underlying JDBC `Connection`. This is by default the case for JDBC-based JDO 2.0 implementations; for JDO 1.0 implementations, a custom `JdoDialect` needs to be used. See next

section for details on the `JdoDialect` mechanism.

12.3.5. `JdoDialect`

As an advanced feature, both `JdoTemplate` and `interfacename` support a custom `JdoDialect`, to be passed into the "jdoDialect" bean property. In such a scenario, the DAOs won't receive a `PersistenceManagerFactory` reference but rather a full `JdoTemplate` instance instead (for example, passed into `JdoDaoSupport`'s "jdoTemplate" property). A `JdoDialect` implementation can enable some advanced features supported by Spring, usually in a vendor-specific manner:

- applying specific transaction semantics (such as custom isolation level or transaction timeout)
- retrieving the transactional JDBC `Connection` (for exposure to JDBC-based DAOs)
- applying query timeouts (automatically calculated from Spring-managed transaction timeout)
- eagerly flushing a `PersistenceManager` (to make transactional changes visible to JDBC-based data access code)
- advanced translation of `JDOExceptions` to Spring `DataAccessExceptions`

This is particularly valuable for JDO 1.0 implementations, where none of those features are covered by the standard API. On JDO 2.0, most of those features are supported in a standard manner: Hence, Spring's `DefaultJdoDialect` uses the corresponding JDO 2.0 API methods by default (as of Spring 1.2). For special transaction semantics and for advanced translation of exception, it is still valuable to derive vendor-specific `JdoDialect` subclasses.

See the `JdoDialect` Javadoc for more details on its operations and how they are used within Spring's JDO support.

12.4. Oracle TopLink

Since Spring 1.2, Spring supports Oracle TopLink (http://www.oracle.com/technology/products/ias/toplink) as data access strategy, following the same style as the Hibernate support. Both TopLink 9.0.4 (the production version as of Spring 1.2) and 10.1.3 (still in beta as of Spring 1.2) are supported. The corresponding integration classes reside in the `org.springframework.orm.toplink` package.

Spring's TopLink support has been co-developed with the Oracle TopLink team. Many thanks to the TopLink team, in particular to Jim Clark who helped to clarify details in all areas!

12.4.1. `SessionFactory` abstraction

TopLink itself does not ship with a SessionFactory abstraction. Instead, multi-threaded access is based on the concept of a central `ServerSession`, which in turn is able to spawn `ClientSession` instances for single-threaded usage. For flexible setup options, Spring defines a `SessionFactory` abstraction for TopLink, enabling to switch between different `Session` creation strategies.

As a one-stop shop, Spring provides a `LocalSessionFactoryBean` class that allows for defining a TopLink `SessionFactory` with bean-style configuration. It needs to be configured with the location of the TopLink session configuration file, and usually also receives a Spring-managed JDBC `DataSource` to use.

```
<beans>

  <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>

  <bean id="mySessionFactory"
class="org.springframework.orm.toplink.LocalSessionFactoryBean">
    <property name="configLocation" value="toplink-sessions.xml"/>
    <property name="dataSource" ref="dataSource"/>
  </bean>

</beans>

<toplink-configuration>

  <session>
    <name>Session</name>
    <project-xml>toplink-mappings.xml</project-xml>
    <session-type>
      <server-session/>
    </session-type>
    <enable-logging>true</enable-logging>
    <logging-options/>
  </session>

</toplink-configuration>
```

Usually, `LocalSessionFactoryBean` will hold a multi-threaded TopLink `ServerSession` underneath and create appropriate client `Session`s for it: either a plain `Session` (typical), a managed `ClientSession`, or a transaction-aware `Session` (the latter are mainly used internally by Spring's TopLink support). It might also hold a single-threaded TopLink `DatabaseSession`; this is rather unusual, though.

12.4.2. `TopLinkTemplate` and `TopLinkDaoSupport`

Each TopLink-based DAO will then receive the `SessionFactory` through dependency injection, i.e. through a bean property setter or through a constructor argument. Such a DAO could be coded against plain TopLink API, fetching a `Session` from the given `SessionFactory`, but will usually rather be used with Spring's `TopLinkTemplate`:

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

</beans>

public class TopLinkProductDao implements ProductDao {
```

```
    private TopLinkTemplate tlTemplate;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.tlTemplate = new TopLinkTemplate(sessionFactory);
    }

    public Collection loadProductsByCategory(final String category) throws
DataAccessException {
        return (Collection) this.tlTemplate.execute(new TopLinkCallback() {
            public Object doInTopLink(Session session) throws TopLinkException {
                ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
                findOwnersQuery.addArgument("Category");
                ExpressionBuilder builder =
this.findOwnersQuery.getExpressionBuilder();
                findOwnersQuery.setSelectionCriteria(

builder.get("category").like(builder.getParameter("Category")));

                Vector args = new Vector();
                args.add(category);
                List result = session.executeQuery(findOwnersQuery, args);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

A callback implementation can effectively be used for any TopLink data access. `TopLinkTemplate` will ensure that `Session`s are properly opened and closed, and automatically participate in transactions. The template instances are thread-safe and reusable, they can thus be kept as instance variables of the surrounding class. For simple single-step actions such as a single `executeQuery`, `readAll`, `readById`, or `merge` call, `JdoTemplate` offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenient `TopLinkDaoSupport` base class that provides a `setSessionFactory(..)` method for receiving a `SessionFactory`, and `getSessionFactory()` and `getTopLinkTemplate()` for use by subclasses. In combination, this allows for simple DAO implementations for typical requirements:

```
public class ProductDaoImpl extends TopLinkDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws
DataAccessException {
        ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
        findOwnersQuery.addArgument("Category");
        ExpressionBuilder builder = this.findOwnersQuery.getExpressionBuilder();
        findOwnersQuery.setSelectionCriteria(
            builder.get("category").like(builder.getParameter("Category")));

        return getTopLinkTemplate().executeQuery(findOwnersQuery, new Object[]
{category});
    }
}
```

Side note: TopLink query objects are thread-safe and can be cached within the DAO, i.e. created on

startup and kept in instance variables.

As alternative to working with Spring's `TopLinkTemplate`, you can also code your TopLink data access based on the raw TopLink API, explicitly opening and closing a `Session`. As elaborated in the corresponding Hibernate section, the main advantage of this approach is that your data access code is able to throw checked exceptions. `TopLinkDaoSupport` offers a variety of support methods for this scenario, for fetching and releasing a transactional `Session` as well as for converting exceptions.

12.4.3. Implementing DAOs based on plain TopLink API

DAOs can also be written against plain TopLink API, without any Spring dependencies, directly using an injected TopLink `Session`. The latter will usually be based on a `SessionFactory` defined by a `LocalSessionFactoryBean`, exposed for bean references of type `Session` through Spring's `TransactionAwareSessionAdapter`.

The `getActiveSession()` method defined on TopLink's `Session` interface will return the current transactional `Session` in such a scenario. If there is no active transaction, it will return the shared TopLink `ServerSession` as-is, which is only supposed to be used directly for read-only access. There is also an analogous `getActiveUnitOfWork()` method, returning the TopLink `UnitOfWork` associated with the current transaction, if any (returning `null` else).

A corresponding DAO implementation looks like as follows:

```
public class ProductDaoImpl implements ProductDao {

    private Session session;

    public void setSession(Session session) {
        this.session = session;
    }

    public Collection loadProductsByCategory(String category) {
        ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
        findOwnersQuery.addArgument("Category");
        ExpressionBuilder builder = this.findOwnersQuery.getExpressionBuilder();
        findOwnersQuery.setSelectionCriteria(
            builder.get("category").like(builder.getParameter("Category")));

        Vector args = new Vector();
        args.add(category);
        return session.getActiveSession().executeQuery(findOwnersQuery, args);
    }
}
```

As the above DAO still follows the Dependency Injection pattern, it still fits nicely into a Spring application context, analogous to like it would if coded against Spring's `TopLinkTemplate`. Spring's `TransactionAwareSessionAdapter` is used to expose a bean reference of type `Session`, to be passed into the DAO:

```
<beans>

  <bean id="mySessionAdapter"

class="org.springframework.orm.toplink.support.TransactionAwareSessionAdapter">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>
```

```
  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="session" ref="mySessionAdapter"/>
  </bean>

</beans>
```

The main advantage of this DAO style is that it depends on TopLink API only; no import of any Spring class is required. This is of course appealing from a non-invasiveness perspective, and might feel more natural to TopLink developers.

However, the DAO throws plain `TopLinkException` (which is unchecked, so does not have to be declared or caught), which means that callers can only treat exceptions as generally fatal - unless they want to depend on TopLink's own exception structure. Catching specific causes such as an optimistic locking failure is not possible without tying the caller to the implementation strategy. This tradeoff might be acceptable to applications that are strongly TopLink-based and/or do not need any special exception treatment.

A further disadvantage of that DAO style is that TopLink's standard `getActiveSession()` feature just works within JTA transactions. It does *not* work with any other transaction strategy out-of-the-box, in particular not with local TopLink transactions.

Fortunately, Spring's `TransactionAwareSessionAdapter` exposes a corresponding proxy for the TopLink `ServerSession` which supports TopLink's `Session.getActiveSession()` and `Session.getActiveUnitOfWork()` methods for any Spring transaction strategy, returning the current Spring-managed transactional `Session` even with `TopLinkTransactionManager`. Of course, the standard behavior of that method remains: returning the current `Session` associated with the ongoing JTA transaction, if any (no matter whether driven by Spring's `JtaTransactionManager`, by EJB CMT, or by plain JTA).

In summary: DAOs can be implemented based on plain TopLink API, while still being able to participate in Spring-managed transactions. This might in particular appeal to people already familiar with TopLink, feeling more natural to them. However, such DAOs will throw plain `TopLinkException`; conversion to Spring's `DataAccessException` would have to happen explicitly (if desired).

12.4.4. Transaction management

To execute service operations within transactions, you can use Spring's common declarative transaction facilities. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="
   http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
   http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
```

```
  <bean id="myTxManager"
class="org.springframework.orm.toplink.TopLinkTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(*
product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

</beans>
```

Note that TopLink requires an active `UnitOfWork` for modifying a persistent object. (You should never modify objects returned by a plain TopLink `Session` - those are usually read-only objects, directly taken from the second-level cache!) There is no concept like a non-transactional flush in TopLink, in contrast to Hibernate. For this reason, TopLink needs to be set up for a specific environment: in particular, it needs to be explicitly set up for JTA synchronization, to detect an active JTA transaction itself and expose a corresponding active `Session` and `UnitOfWork`. This is not necessary for local transactions as performed by Spring's `TopLinkTransactionManager`, but it is necessary for participating in JTA transactions (whether driven by Spring's `JtaTransactionManager` or by EJB CMT / plain JTA).

Within your TopLink-based DAO code, use the `Session.getActiveUnitOfWork()` method to access the current `UnitOfWork` and perform write operations through it. This will only work within an active transaction (both within Spring-managed transactions and plain JTA transactions). For special needs, you can also acquire separate `UnitOfWork` instances that won't participate in the current transaction; this is hardly needed, though.

`TopLinkTransactionManager` is capable of exposing a TopLink transaction to JDBC access code that accesses the same JDBC `DataSource`, provided that TopLink works with JDBC in the backend and is thus able to expose the underlying JDBC `Connection`. The `DataSource` to expose the transactions for needs to be specified explicitly; it won't be autodetected.

12.5. iBATIS SQL Maps

The iBATIS support in the Spring Framework much resembles the JDBC / Hibernate support in that it supports the same template style programming and just as with JDBC or Hibernate, the iBATIS support works with Spring's exception hierarchy and let's you enjoy the all IoC features Spring has.

Transaction management can be handled through Spring's standard facilities. There are no special transaction strategies for iBATIS, as there is no special transactional resource involved other than a

JDBC `Connection`. Hence, Spring's standard JDBC `DataSourceTransactionManager` or `JtaTransactionManager` are perfectly sufficient.

Note

Spring does actually support both iBatis 1.x and 2.x. However, only support for iBatis 2.x is actually shipped with the core Spring distribution. The iBatis 1.x support classes were moved to the Spring Modules project as of Spring 2.0, and you are directed there for documentation.

12.5.1. Setting up the `SqlMapClient`

If we want to map the previous Account class with iBATIS 2.x we need to create the following SQL map `'Account.xml'`:

```
<sqlMap namespace="Account">

  <resultMap id="result" class="examples.Account">
    <result property="name" column="NAME" columnIndex="1"/>
    <result property="email" column="EMAIL" columnIndex="2"/>
  </resultMap>

  <select id="getAccountByEmail" resultMap="result">
    select ACCOUNT.NAME, ACCOUNT.EMAIL
    from ACCOUNT
    where ACCOUNT.EMAIL = #value#
  </select>

  <insert id="insertAccount">
    insert into ACCOUNT (NAME, EMAIL) values (#name#, #email#)
  </insert>

</sqlMap>
```

The configuration file for iBATIS 2 looks like this:

```
<sqlMapConfig>

  <sqlMap resource="example/Account.xml"/>

</sqlMapConfig>
```

Remember that iBATIS loads resources from the class path, so be sure to add the `'Account.xml'` file to the class path.

We can use the `SqlMapClientFactoryBean` in the Spring container. Note that with iBATIS SQL Maps 2.x, the JDBC `DataSource` is usually specified on the `SqlMapClientFactoryBean`, which enables lazy loading.

```
<beans>

  <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>

  <bean id="sqlMapClient"
class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <property name="configLocation" value="WEB-INF/sqlmap-config.xml"/>
    <property name="dataSource" ref="dataSource"/>
  </bean>

</beans>
```

12.5.2. Using `SqlMapClientTemplate` and `SqlMapClientDaoSupport`

The `SqlMapClientDaoSupport` class offers a supporting class similar to the `SqlMapDaoSupport`. We extend it to implement our DAO:

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport implements AccountDao
{

    public Account getAccount(String email) throws DataAccessException {
        return (Account)
getSqlMapClientTemplate().queryForObject("getAccountByEmail", email);
    }

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapClientTemplate().update("insertAccount", account);
    }
}
```

In the DAO, we use the pre-configured `SqlMapClientTemplate` to execute the queries, after setting up the `SqlMapAccountDao` in the application context and wiring it with our `SqlMapClient` instance:

```
<beans>

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="sqlMapClient" ref="sqlMapClient"/>
  </bean>

</beans>
```

Note that a `SqlMapTemplate` instance could also be created manually, passing in the `SqlMapClient` as constructor argument. The `SqlMapClientDaoSupport` base class simply pre-initializes a `SqlMapClientTemplate` instance for us.

The `SqlMapClientTemplate` also offers a generic `execute` method, taking a custom `SqlMapClientCallback` implementation as argument. This can, for example, be used for batching:

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport implements AccountDao
{

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapClientTemplate().execute(new SqlMapClientCallback() {
            public Object doInSqlMapClient(SqlMapExecutor executor) throws
SQLException {
```

```
                executor.startBatch();
                executor.update("insertAccount", account);
                executor.update("insertAddress", account.getAddress());
                executor.executeBatch();
            }
        });
    }
}
```

In general, any combination of operations offered by the native `SqlMapExecutor` API can be used in such a callback. Any `SQLException` thrown will automatically get converted to Spring's generic `DataAccessException` hierarchy.

12.5.3. Implementing DAOs based on plain iBATIS API

DAOs can also be written against plain iBATIS API, without any Spring dependencies, directly using an injected `SqlMapClient`. A corresponding DAO implementation looks like as follows:

```
public class SqlMapAccountDao implements AccountDao {

    private SqlMapClient sqlMapClient;

    public void setSqlMapClient(SqlMapClient sqlMapClient) {
        this.sqlMapClient = sqlMapClient;
    }

    public Account getAccount(String email) {
        try {
            return (Account) this.sqlMapClient.queryForObject("getAccountByEmail",
email);
        }
        catch (SQLException ex) {
            throw new MyDaoException(ex);
        }
    }

    public void insertAccount(Account account) throws DataAccessException {
        try {
            this.sqlMapClient.update("insertAccount", account);
        }
        catch (SQLException ex) {
            throw new MyDaoException(ex);
        }
    }
}
```

In such a scenario, the `SQLException` thrown by the iBATIS API needs to be handled in a custom fashion: usually, wrapping it in your own application-specific DAO exception. Wiring in the application context would still look like before, due to the fact that the plain iBATIS-based DAO still follows the Dependency Injection pattern:

```
<beans>

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="sqlMapClient" ref="sqlMapClient"/>
  </bean>
```

```
</beans>
```

12.6. JPA

Spring JPA (available under the `org.springframework.orm.jpa` package) offers comprehensive support for the Java Persistence API in a similar manner to the integration with Hibernate or JDO, while being aware of the underlying implementation in order to provide additional features.

12.6.1. JPA setup in a Spring environment

Spring JPA offers three ways of setting up JPA `EntityManagerFactory`:

12.6.1.1. `LocalEntityManagerFactoryBean`

The `LocalEntityManagerFactoryBean` creates an `EntityManagerFactory` suitable for environments which solely use JPA for data access. The factory bean will use the JPA `PersistenceProvider` autodetection mechanism (according to JPA's Java SE bootstrapping) and, in most cases, requires only the persistence unit name to be specified:

```
<beans>

   <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
      <property name="persistenceUnitName" value="myPersistenceUnit"/>
   </bean>

</beans>
```

This is the simplest but also most limited form of JPA deployment. There is no way to link to an existing JDBC `DataSource` and no support for global transactions, for example. Furthermore, weaving (byte-code transformation) of persistent classes is provider-specific, often requiring a specific JVM agent to specified on startup. All in all, this option is only really sufficient for standalone applications and test environments (which is exactly what the JPA specification designed it for).

Only use this option in simple deployment environments like standalone applications and integration tests.

12.6.1.2. `Obtaining an EntityManagerFactory from JNDI`

Obtaining an `EntityManagerFactory` from JNDI (for example in a Java EE 5 environment), is just a matter of changing the XML configuration:

```
<beans>

    <jee:jndi-lookup id="entityManagerFactory"
jndi-name="persistence/myPersistenceUnit"/>

</beans>
```

This assumes standard Java EE 5 bootstrapping, with the Java EE server autodetecting persistence units

(i.e. `META-INF/persistence.xml` files in application jars) and `persistence-unit-ref` entries in the Java EE deployment descriptor (e.g. `web.xml`) defining environment naming context locations for those persistence units.

In such a scenario, the entire persistence unit deployment, including the weaving (byte-code transformation) of persistent classes, is up to the Java EE server. The JDBC `DataSource` is defined through a JNDI location in the `META-INF/persistence.xml` file; EntityManager transactions are integrated with the server's JTA subsystem. Spring merely uses the obtained `EntityManagerFactory`, passing it on to application objects via dependency injection, and managing transactions for it (typically through `JtaTransactionManager`).

Note that, in case of multiple persistence units used in the same application, the bean names of such a JNDI-retrieved persistence units should match the persistence unit names that the application uses to refer to them (e.g. in `@PersistenceUnit` and `@PersistenceContext` annotations).

Use this option when deploying to a Java EE 5 server. Check your server's documentation on how to deploy a custom JPA provider into your server, allowing for a different provider than the server's default.

12.6.1.3. `LocalContainerEntityManagerFactoryBean`

The `LocalContainerEntityManagerFactoryBean` gives full control over `EntityManagerFactory` configuration and is appropriate for environments where fine-grained customization is required. The `LocalContainerEntityManagerFactoryBean` will create a `PersistenceUnitInfo` based on the `persistence.xml` file, the supplied `dataSourceLookup` strategy and the specified `loadTimeWeaver`. It is thus possible to work with custom DataSources outside of JNDI and to control the weaving process.

```
<beans>

 <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="someDataSource"/>
  <property name="loadTimeWeaver">
    <bean
class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/
>
  </property>
 </bean>

</beans>
```

This is the most powerful JPA setup option, allowing for flexible local configuration within the application. It supports links to an existing JDBC `DataSource`, supports both local and global transactions, etc. However, it also imposes requirements onto the runtime environment, such as the availability of a weaving-capable ClassLoader if the persistence provider demands byte-code transformation.

Note that this option may conflict with the built-in JPA capabilities of a Java EE 5 server. So when running in a full Java EE 5 environment, consider obtaining your `EntityManagerFactory` from JNDI. Alternatively, specify a custom "persistenceXmlLocation" on your `LocalContainerEntityManagerFactoryBean` definition, e.g. "META-INF/my-persistence.xml", and only include a descriptor with that name in your application jar

files. Since the Java EE 5 server will only look for default `META-INF/persistence.xml` files, it will ignore such custom persistence units and hence avoid conflicts with a Spring-driven JPA setup upfront. (This applies to Resin 3.1, for example.)

Use this option for full JPA capabilities in a Spring-based application environment. This includes web containers such as Tomcat as well as standalone applications and integration tests with sophisticated persistence requirements.

When is load time weaving required?

Not all JPA providers impose the need of a JVM agent (Hibernate being an example). If your provider does not require an agent or you have other alternatives (for example applying enhancements at build time through a custom compiler or an ant task) the load time weaver **should not** be used.

The `LoadTimeWeaver` interface is a Spring-provided class that allows JPA `ClassTransformer` instances to be plugged in a specific manner depending on the environment (web container/application server). Hooking `ClassTransformers` through a JDK 5.0 agent is typically not efficient - the agents work against the *entire virtual machine* and inspect *every* class that is loaded - something that is typically undesirable in a production server enviroment.

Spring provides a number of `LoadTimeWeaver` implementations for various environments, allowing `ClassTransformer` instances to be applied only *per ClassLoader* and not per VM.

12.6.1.3.1. Tomcat load-time weaving setup (5.0+)

Jakarta Tomcat's default ClassLoader does not support class transformation but allows custom ClassLoaders to be used. Spring offers the `TomcatInstrumentableClassLoader` (inside the `org.springframework.instrument.classloading.tomcat` package) which extends the Tomcat ClassLoader (`WebappClassLoader`) and allows JPA `ClassTransformer` instances to 'enhance' all classes loaded by it. In short, JPA transformers will be applied only inside a specific web application (which uses the `TomcatInstrumentableClassLoader`).

In order to use the custom ClassLoader on:

1. Copy `spring-tomcat-weaver.jar` into *$CATALINA_HOME*/server/lib (where *$CATALINA_HOME* represents the root of the Tomcat installation).
2. Instruct Tomcat to use the custom ClassLoader (instead of the default one) by editing the web application context file:

   ```
   <Context path="/myWebApp" docBase="/my/webApp/location">
      <Loader
   loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstru
   mentableClassLoader" />
   </Context>
   ```

   Tomcat 5.0.x and 5.5.x series support several context locations: server configuration file (*$CATALINA_HOME/conf/server.xml*), the default context configuration (*$CATALINA_HOME/conf/context.xml*) that affects all deployed web applications and per-webapp configurations, deployed on the server *($CATALINA_HOME/conf/[enginename]/[hostname]/my-webapp-context.xml*) side or along with the webapp (*your-webapp.war/META-INF/context.xml*). For efficiency, inside the web-app configuration style is recommended since only applications which use JPA will use the custom ClassLoader. See the Tomcat 5.x documentation for more details about available context locations.

Note that versions prior to 5.5.20 contained a bug in the XML configuration parsing preventing usage of `Loader` tag inside *server.xml* (no matter if a ClassLoader is specified or not (be it the official or a custom one). See Tomcat's bugzilla for more details.

If you are using Tomcat 5.5.20+ you can set *useSystemClassLoaderAsParent* to `false` to fix the problem:

```
<Context path="/myWebApp" docBase="/my/webApp/location">
    <Loader
loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstru
mentableClassLoader"
            useSystemClassLoaderAsParent="false"/>
</Context>
```

1. Copy `spring-tomcat-weaver.jar` into *$CATALINA_HOME*/lib (where *$CATALINA_HOME* represents the root of the Tomcat installation).
2. Instruct Tomcat to use the custom ClassLoader (instead of the default one) by editing the web application context file:

   ```
   <Context path="/myWebApp" docBase="/my/webApp/location">
       <Loader
   loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstru
   mentableClassLoader"/>
   </Context>
   ```

   Tomcat 6.0.x (similar to 5.0.x/5.5.x) series support several context locations: server configuration file (*$CATALINA_HOME/conf/server.xml*), the default context configuration (*$CATALINA_HOME/conf/context.xml*) that affects all deployed web applications and per-webapp configurations, deployed on the server *($CATALINA_HOME/conf/[enginename]/[hostname]/my-webapp-context.xml*) side or along with the webapp (*your-webapp.war/META-INF/context.xml*). For efficiency, inside the web-app configuration style is recommended since only applications which use JPA will use the custom ClassLoader. See the Tomcat 5.x documentation for more details about available context locations.

- Tomcat 5.0.x/5.5.x
- Tomcat 6.0.x

The last step required on all Tomcat versions, is to use the appropriate the `LoadTimeWeaver` when configuring `LocalContainerEntityManagerFactoryBean`:

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="loadTimeWeaver">
    <bean
class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
  </property>
</bean>
```

Using this technique, JPA applications relying on instrumentation, can run in Tomcat without the need of an agent. This is important especially when hosting applications which rely on different JPA implementations since the JPA transformers are applied only at ClassLoader level and thus, are isolated from each other.

Note

If TopLink is being used a JPA provider under Tomcat, please place the toplink-essentials jar under *$CATALINA_HOME*/shared/lib folder instead of your war.

12.6.1.3.2. OC4J load-time weaving setup (10.1.3.1+)

As Oracle's OC4J ClassLoader has native bytecode transformation support, switching from an JDK agent to a `LoadTimeWeaver` can be done just through the application Spring configuration:

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="loadTimeWeaver">
    <bean
class="org.springframework.instrument.classloading.oc4j.OC4JLoadTimeWeaver"/>
  </property>
</bean>
```

12.6.1.3.3. GlassFish load-time weaving setup

GlassFish application server provides out of the box, an instrumentation cable ClassLoader. Spring supports it through `GlassFishLoadTimeWeaver`:

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="loadTimeWeaver">
    <bean
class="org.springframework.instrument.classloading.glassfish.GlassFishLoadTimeWeav
er"/>
  </property>
</bean>
```

12.6.1.3.4. Resin load-time weaving setup (3.1+)

Caucho Resin 3.1 sesies ClassLoader provides native bytecode capabilities which can be used by Spring through `ReflectiveLoadTimeWeaver`:

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="loadTimeWeaver">
    <bean
class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
  </property>
</bean>
```

12.6.1.3.5. General `LoadTimeWeaver`

For environments where class instrumentation is required but are not supported by the existing LoadTimeWeaver implementations, a JDK agent can be the only solution. For such cases, Spring provides `InstrumentationLoadTimeWeaver` which requires a Spring-specific (but very general) VM agent (`spring-agent.jar`):

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="loadTimeWeaver">
```

```
    <bean
class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/
>
  </property>
</bean>
```

Note that the virtual machine has to be started with the Spring agent, by supplying the following JVM options:

```
-javaagent:/path/to/spring-agent.jar
```

12.6.1.4. Dealing with multiple persistence units

For applications that rely on multiple persistence units locations (stored in various jars in the classpath for example), Spring offers the `PersistenceUnitManager` to act as a central repository and avoid the (potentially expensive) persistence units discovery process. The default implementation allows multiple locations to be specified (by default, the classpath is searched for `'META-INF/persistence.xml'` files) which are parsed and later on retrieved through the persistence unit name:

```
<bean id="persistenceUnitManager"
class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager">
  <property name="persistenceXmlLocation">
    <list>
     <value>org/springframework/orm/jpa/domain/persistence-multi.xml</value>
     <value>classpath:/my/package/**/custom-persistence.xml</value>
     <value>classpath*:META-INF/persistence.xml</value>
    </list>
  </property>
  <property>
   <map>
    <entry key="localDataSource" value-ref="local-db"/>
    <entry key="remoteDataSource" value-ref="remote-db"/>
   </map>
  </property>
  <!-- if no datasource is specified, use this one -->
  <property name="defaultDataSource" ref="remoteDataSource"/>

</bean>

<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
 <property name="persistenceUnitManager" ref="persistenceUnitManager"/>
</bean>
```

Note that the default implementation allows customization of the persistence unit infos before feeding them to the JPA provider declaratively through its properties (which affect *all* housed units) or programmatically, through the `PersistenceUnitPostProcessor` (which allow persistence unit selection). If no persistenceUnitManager is specified, one will be created and used internally by `LocalContainerEntityManagerFactoryBean`.

12.6.2. `JpaTemplate` and `JpaDaoSupport`

Each JPA-based DAO will then receive a `EntityManagerFactory` via dependency injection. Such

a DAO can be coded against plain JPA and work with the given `EntityManagerFactory` or through Spring's `JpaTemplate`:

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
  </bean>

</beans>
```

```
public class JpaProductDao implements ProductDao {

    private JpaTemplate jpaTemplate;

    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.jpaTemplate = new JpaTemplate(emf);
    }

    public Collection loadProductsByCategory(final String category) throws
DataAccessException {
        return (Collection) this.jpaTemplate.execute(new JpaCallback() {
            public Object doInJpa(EntityManager em) throws PersistenceException {
                Query query = em.createQuery("from Product as p where p.category =
:category");
                query.setParameter("category", category);
                List result = query.getResultList();
                // do some further processing with the result list
                return result;
            }
        });
    }
}
```

The `JpaCallback` implementation allows any type of JPA data access. The `JpaTemplate` will ensure that `EntityManager`s are properly opened and closed and automatically participate in transactions. Moreover, the `JpaTemplate` properly handles exceptions, making sure resources are cleaned up and the appropriate transactions rolled back. The template instances are thread-safe and reusable and they can be kept as instance variable of the enclosing class. Note that `JpaTemplate` offers single-step actions such as find, load, merge, etc along with alternative convenience methods that can replace one line callback implementations.

Furthermore, Spring provides a convenient `JpaDaoSupport` base class that provides the `get/setEntityManagerFactory` and `getJpaTemplate()` to be used by subclasses:

```
public class ProductDaoImpl extends JpaDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws
DataAccessException {
        Map<String, String> params = new HashMap<String, String>();
        params.put("category", category);
        return getJpaTemplate().findByNamedParams("from Product as p where
p.category = :category", params);
    }
}
```

Besides working with Spring's `JpaTemplate`, one can also code Spring-based DAOs against the JPA, doing one's own explicit `EntityManager` handling. As also elaborated in the corresponding Hibernate section, the main advantage of this approach is that your data access code is able to throw checked exceptions. `JpaDaoSupport` offers a variety of support methods for this scenario, for retrieving and releasing a transaction `EntityManager`, as well as for converting exceptions.

JpaTemplate mainly exists as a sibling of JdoTemplate and HibernateTemplate, offering the same style for people used to it. For newly started projects, consider adopting the native JPA style of coding data access objects instead, based on a "shared EntityManager" reference obtained through the JPA `@PersistenceContext` annotation (using Spring's `PersistenceAnnotationBeanPostProcessor`; see below for details.)

12.6.3. Implementing DAOs based on plain JPA

Note

While `EntityManagerFactory` instances are thread-safe, `EntityManager` instances are not. The injected JPA `EntityManager` behave just like an `EntityManager` fetched from an application server's JNDI environment, as defined by the JPA specification. It will delegate all calls to the current transactional `EntityManager`, if any; else, it will fall back to a newly created `EntityManager` per operation, making it thread-safe.

It is possible to write code against the plain JPA without using any Spring dependencies, using an injected `EntityManagerFactory` or `EntityManager`. Note that Spring can understand `@PersistenceUnit` and `@PersistenceContext` annotations both at field and method level if a `PersistenceAnnotationBeanPostProcessor` is enabled. A corresponding DAO implementation might look like this:

```
public class ProductDaoImpl implements ProductDao {

    private EntityManagerFactory emf;

    @PersistenceUnit
    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.emf = emf;
    }

    public Collection loadProductsByCategory(String category) {
        EntityManager em = this.emf.createEntityManager();
        try {
             Query query = em.createQuery("from Product as p where p.category = ?
1");
             query.setParameter(1, category);
             return query.getResultList();
        }
        finally {
            if (em != null) {
                em.close();
            }
        }
    }
}
```

The DAO above has no dependency on Spring and still fits nicely into a Spring application context, just like it would if coded against Spring's `JpaTemplate`. Moreover, the DAO takes advantage of annotations to require the injection of the default `EntityManagerFactory`:

```
<beans>

  <!-- bean post-processor for JPA annotations -->
  <bean
class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"
/>

  <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

The main issue with such a DAO is that it always creates a new `EntityManager` via the factory. This can be easily overcome by requesting a transactional `EntityManager` (also called "shared EntityManager", since it is a shared, thread-safe proxy for the actual transactional EntityManager) to be injected instead of the factory:

```
public class ProductDaoImpl implements ProductDao {

    @PersistenceContext
    private EntityManager em;

    public Collection loadProductsByCategory(String category) {
       Query query = em.createQuery("from Product as p where p.category =
:category");
       query.setParameter("category", category);
       return query.getResultList();
    }
}
```

Note that the `@PersistenceContext` annotation has an optional attribute `type`, which defaults to `PersistenceContextType.TRANSACTION`. This default is what you need to receive a "shared EntityManager" proxy. The alternative, `PersistenceContextType.EXTENDED`, is a completely different affair: This results in a so-called "extended EntityManager", which is *not thread-safe* and hence must not be used in a concurrently accessed component such as a Spring-managed singleton bean. Extended EntityManagers are only supposed to be used in stateful components that, for example, reside in a session, with the lifecycle of the EntityManager not tied to a current transaction but rather being completely up to the application.

Method and Field level Injection

Annotations that indicate dependency injections (such as `@PersistenceUnit` and `@PersistenceContext`) can be applied on field or methods inside a class, therefore the expression "method/field level injection". Field-level annotations concise and easier to use while method-level allow for processing the injected dependency. In both cases the member visibility (public, protected, private) does not matter.

What about class level annotations?

On JEE 5 platform, they are used for dependency declaration and not for resource injection.

The injected `EntityManager` is Spring-managed (aware of the ongoing transaction). It is important to note that even though the new implementation prefers method level injection (of an

`EntityManager` instead of an `EntityManagerFactory)`, no change is required in the application context XML due to annotation usage.

The main advantage of this DAO style is that it depends on Java Persistence API; no import of any Spring class is required. Moreover, as the JPA annotations are understood, the injections are applied automatically by the Spring container. This is of course appealing from a non-invasiveness perspective, and might feel more natural to JPA developers.

12.6.4. Exception Translation

However, the DAO throws the plain `PersistenceException` exception class (which is unchecked, and so does not have to be declared or caught) but also `IllegalArgumentException` and `IllegalStateException`, which means that callers can only treat exceptions as generally fatal - unless they want to depend on JPA's own exception structure. Catching specific causes such as an optimistic locking failure is not possible without tying the caller to the implementation strategy. This tradeoff might be acceptable to applications that are strongly JPA-based and/or do not need any special exception treatment. However, Spring offers a solution allowing exception translation to be applied transparently through the `@Repository` annotation:

```
@Repository
public class ProductDaoImpl implements ProductDao {

    // class body here...

}

<beans>

  <!-- Exception translation bean post processor -->
  <bean
class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProce
ssor"/>

  <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

The postprocessor will automatically look for all exception translators (implementations of the `PersistenceExceptionTranslator` interface) and advice all beans maked with the `@Repository` annotation so that the discovered translators can intercept and apply the appropriate translation on the thrown exceptions.

In summary: DAOs can be implemented based on the plain Java Persistence API and annotations, while still being able to benefit from Spring-managed transactions, dependency injection, and transparent exception conversion (if desired) to Spring's custom exception hierarchies.

12.7. Transaction Management

To execute service operations within transactions, you can use Spring's common declarative transaction facilities. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <bean id="myTxManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(*
product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

</beans>
```

Spring JPA allows a configured `JpaTransactionManager` to expose a JPA transaction to JDBC access code that accesses the same JDBC `DataSource`, provided that the registered `JpaDialect` supports retrieval of the underlying JDBC `Connection`. Out of the box, Spring provides dialects for the Toplink, Hibernate and OpenJPA JPA implementations. See the next section for details on the `JpaDialect` mechanism.

12.8. `JpaDialect`

As an advanced feature `JpaTemplate`, `JpaTransactionManager` and subclasses of `AbstractEntityManagerFactoryBean` support a custom `JpaDialect`, to be passed into the "jpaDialect" bean property. In such a scenario, the DAOs won't receive an `EntityManagerFactory` reference but rather a full `JpaTemplate` instance instead (for example, passed into `JpaDaoSupport`'s "jpaTemplate" property). A `JpaDialect` implementation can enable some advanced features supported by Spring, usually in a vendor-specific manner:

- applying specific transaction semantics (such as custom isolation level or transaction timeout)
- retrieving the transactional JDBC `Connection` (for exposure to JDBC-based DAOs)
- advanced translation of `PersistenceExceptions` to Spring `DataAccessExceptions`

This is particularly valuable for special transaction semantics and for advanced translation of exception. Note that the default implementation used (`DefaultJpaDialect`) doesn't provide any special capabilities and if the above features are required, the appropriate dialect has to be specified.

See the `JpaDialect` Javadoc for more details of its operations and how they are used within Spring's JPA support.