15. Object Relational Mapping (ORM) Data Access

15.1 Introduction to ORM with Spring

The Spring Framework supports integration with Hibernate, Java Persistence API (JPA), Java Data Objects (JDO) and iBATIS SQL Maps for resource management, data access object (DAO) implementations, and transaction strategies. For example, for Hibernate there is first-class support with several convenient IoC features that address many typical Hibernate integration issues. You can configure all of the supported features for O/R (object relational) mapping tools through Dependency Injection. They can participate in Spring's resource and transaction management, and they comply with Spring's generic transaction and DAO exception hierarchies. The recommended integration style is to code DAOs against plain Hibernate, JPA, and JDO APIs. The older style of using Spring's DAO templates is no longer recommended; however, coverage of this style can be found in the Section A.1, “Classic ORM usage” in the appendices.

Spring adds significant enhancements to the ORM layer of your choice when you create data access applications. You can leverage as much of the integration support as you wish, and you should compare this integration effort with the cost and risk of building a similar infrastructure in-house. You can use much of the ORM support as you would a library, regardless of technology, because everything is designed as a set of reusable JavaBeans. ORM in a Spring IoC container facilitates configuration and deployment. Thus most examples in this section show configuration inside a Spring container.

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

- *Easier testing.* Spring's IoC approach makes it easy to swap the implementations and configuration locations of Hibernate `SessionFactory` instances, JDBC `DataSource` instances, transaction managers, and mapped object implementations (if needed). This in turn makes it much easier to test each piece of persistence-related code in isolation.

- *Common data access exceptions.* Spring can wrap exceptions from your ORM tool, converting them from proprietary (potentially checked) exceptions to a common runtime DataAccessException hierarchy. This feature 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 as necessary. 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, JPA `EntityManagerFactory` 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 that uses Hibernate generally needs to use the same Hibernate `Session` to ensure efficiency and proper transaction handling. Spring makes it easy to create and bind a `Session` to the current thread transparently, by exposing a current `Session` through the Hibernate `SessionFactory`. Thus Spring solves many chronic problems of typical Hibernate usage, for any local or JTA transaction environment.
- *Integrated transaction management.* You can wrap your ORM code with a declarative, aspect-oriented programming (AOP) style method interceptor either through the `@Transactional` annotation or by explicitly configuring the transaction AOP advice in an XML configuration file. In both cases, transaction semantics and exception handling (rollback, and so on) are handled for you. As discussed below, in Resource and transaction management, you can also swap various transaction managers, without affecting your ORM-related code. For example, you can swap between local transactions and JTA, with the same full services (such as declarative transactions) available in both scenarios. Additionally, JDBC-related code can fully integrate transactionally with the code you use to do ORM. This is useful for data access that is not suitable for ORM, such as batch processing and BLOB streaming, which still need to share common transactions with ORM operations.

TODO: provide links to current samples

15.2 General ORM integration considerations

This section highlights considerations that apply to all ORM technologies. The Section 15.3, “Hibernate” section provides more details and also show these features and configurations in a concrete context.

The major goal of Spring's ORM integration is 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 need not be Spring-aware. In a typical Spring application, many important objects are JavaBeans: data access templates, data access objects, 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.

15.2.1 Resource and transaction management

Typical business applications are cluttered with repetitive resource management code. Many projects try to invent their own solutions, sometimes sacrificing proper handling of failures for programming convenience. Spring advocates simple solutions for proper resource handling, namely IoC through templating in the case of JDBC and applying AOP interceptors for the ORM technologies.

The infrastructure provides proper resource handling and 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 provides connection handling and proper conversion of `SQLException` to the `DataAccessException` hierarchy, including translation of database-specific SQL error codes to meaningful exception classes. For ORM technologies, see the next section for how to get the same exception translation benefits.

When it comes to transaction management, the `JdbcTemplate` class hooks in to the Spring transaction support and supports both JTA and JDBC transactions, through respective Spring transaction managers. For the supported ORM technologies Spring offers Hibernate, JPA and JDO support through the Hibernate, JPA, and JDO transaction managers as well as JTA support. For details on transaction support, see the Chapter 12, *Transaction Management* chapter.

15.2.2 Exception translation

When you use Hibernate, JPA, or JDO in a DAO, you must decide how to handle the persistence technology's native exception classes. The DAO throws a subclass of a `HibernateException`, `PersistenceException` or `JDOException` depending on the technology. These exceptions are all run-time exceptions and do not have to be declared or caught. You may also have to deal with `IllegalArgumentException` and `IllegalStateException`. This means that callers can only treat exceptions as generally fatal, unless they want to depend on the persistence technology'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 trade off might be acceptable to applications that are strongly ORM-based and/or do not need any special exception treatment. However, Spring enables 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.PersistenceExceptionTranslationPostProcessor"/>

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

</beans>
```

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

In summary: you can implement DAOs based on the plain persistence technology's API and annotations, while still benefiting from Spring-managed transactions, dependency injection, and transparent exception conversion (if desired) to Spring's custom exception hierarchies.

15.3 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 demarcation. 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.

As of Spring 3.0, Spring requires Hibernate 3.2 or later.

15.3.1 `SessionFactory` setup in a Spring container

To avoid tying application objects to hard-coded resource lookups, you can define resources such as a JDBC `DataSource` or a Hibernate `SessionFactory` as beans in the Spring container. Application objects that need to access resources receive references to such predefined instances through bean references, as illustrated in the DAO definition in the next section.

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>
```

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>

  <jee:jndi-lookup id="myDataSource" jndi-name="java:comp/env/jdbc/myds"/>

</beans>
```

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

15.3.2 Implementing DAOs based on plain Hibernate 3 API

Hibernate 3 has a feature called contextual sessions, wherein 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 resembles the following example, 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 similar to that of 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 as 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, the DAO would resemble the following:

```
<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 tying the caller to the implementation strategy. This trade off 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 the return of the current `Session` associated with the ongoing JTA transaction, if any. This behavior applies regardless of whether you are using Spring's `JtaTransactionManager`, EJB container managed transactions (CMTs), or JTA.

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

15.3.3 Declarative transaction demarcation

We recommend that you use Spring's declarative transaction support, which enables you to replace explicit transaction demarcation API calls in your Java code with an AOP transaction interceptor. This transaction interceptor can be configured in a Spring container using either Java annotations or XML.This declarative transaction capability allows you to keep business services free of repetitive transaction demarcation code and to focus on adding business logic, which is the real value of your application.

Prior to continuing, you are *strongly* encouraged to read Section 12.5, “Declarative transaction management” if you have not done so.

Furthermore, transaction semantics like propagation behavior and isolation level can be changed in a configuration file and do not affect the business service implementations.

The following example shows how you can configure an AOP transaction interceptor, using XML, for a simple service class:

```
<?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.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx.xsd

       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop.xsd">

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

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

This is the service class that is advised:

```
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);
        // ...
    }
}
```

We also show an attribute-support based configuration, in the following example. You annotate the service layer with @Transactional annotations and instruct the Spring container to find these annotations and provide transactional semantics for these annotated methods.

```
public class ProductServiceImpl implements ProductService {
```

```
    private ProductDao productDao;

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

    @Transactional
    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDao.loadProductsByCategory(category);
        // ...
    }

    @Transactional(readOnly = true)
    public List<Product> findAllProducts() {
        return this.productDao.findAllProducts();
    }

}
```

As you can see from the following configuration example, the configuration is much simplified, compared to the XML example above, while still providing the same functionality driven by the annotations in the service layer code. All you need to provide is the TransactionManager implementation and a "<tx:annotation-driven/>" entry.

```
<?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.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx.xsd
```

```
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

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

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

  <tx:annotation-driven/>

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

</beans>
```

15.3.4 Programmatic transaction demarcation

You can demarcate transactions in a higher level of the application, on top of such lower-level data access services spanning any number of operations. Nor do restrictions exist on the implementation of the surrounding business service; it just needs a Spring `PlatformTransactionManager`. Again, the latter can come from anywhere, but preferably as a bean reference through a `setTransactionManager(..)` method, just as the `productDAO` should be set by 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...
                }
            }
        );
    }
}
```

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` triggers a rollback in case of an unchecked application exception, or if the transaction is marked rollback-only by the application (via `TransactionStatus`). `TransactionInterceptor` behaves the same way by default but allows configurable rollback policies per method.

15.3.5 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 can even use a custom `PlatformTransactionManager` implementation. 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, because 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 DAO then gets one specific `SessionFactory` reference passed into its corresponding 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>

  <jee:jndi-lookup id="dataSource1" jndi-name="java:comp/env/jdbc/myds1"/>

  <jee:jndi-lookup id="dataSource2" jndi-name="java:comp/env/jdbc/myds2"/>

  <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>

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

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

`HibernateTransactionManager` can export the Hibernate JDBC `Connection` to plain JDBC access code, for a specific `DataSource`. This capability allows for high-level transaction demarcation with mixed Hibernate and JDBC data access completely without JTA, if you are accessing only one database. `HibernateTransactionManager` automatically exposes the Hibernate transaction as a JDBC transaction if you have set up the passed-in `SessionFactory` with a `DataSource` through the

`dataSource` property of the `LocalSessionFactoryBean` class. Alternatively, you can specify explicitly the `DataSource` for which the transactions are supposed to be exposed through the `dataSource` property of the `HibernateTransactionManager` class.

15.3.6 Comparing container-managed and locally defined resources

You can switch between a container-managed JNDI `SessionFactory` and a locally defined one, without having to change a single line of application code. Whether to keep resource definitions in the container or locally within the application is mainly a matter of the transaction strategy that you use. 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 Java EE server's management infrastructure, but does not add actual value beyond that.

Spring's transaction support is not bound to a container. Configured with any strategy other than JTA, transaction support also works in a stand-alone or test environment. Especially in the typical case of single-database transactions, Spring's single-resource local transaction support is a lightweight and powerful alternative to JTA. When you use local EJB stateless session beans to drive transactions, you depend both on an EJB container and JTA, even if you access only a single database, and only use stateless session beans to provide declarative transactions through container-managed transactions. Also, direct use of JTA programmatically requires a Java EE environment as well. JTA does not involve only 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` configured for proper JVM-level caching.

Spring-driven transactions can work as well with a locally defined Hibernate `SessionFactory` as they do with a local JDBC `DataSource` if they are accessing a single database. Thus you only have to use Spring's JTA transaction strategy when you have distributed transaction requirements. A JCA connector requires container-specific deployment steps, and obviously JCA support in the first place. This configuration requires more work than deploying a simple web application with local resource definitions and Spring-driven transactions. Also, you often need the Enterprise Edition of your container if you are using, for example, WebLogic Express, which does not provide JCA. A Spring application with local resources and transactions spanning one single database works in any Java EE web container (without JTA, JCA, or EJB) such as Tomcat, Resin, or even plain Jetty. Additionally, you can easily reuse such a middle tier in desktop applications or test suites.

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

15.3.7 Spurious application server warnings with Hibernate

In some JTA environments with very strict `XADataSource` implementations -- currently only some WebLogic Server and WebSphere versions -- when Hibernate is configured without regard to 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 indicate 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.
```

You resolve this warning by simply making Hibernate aware of the JTA `PlatformTransactionManager` instance, to which it will synchronize (along with Spring). You have two options for doing this:

- If in your application context you are already directly obtaining the JTA `PlatformTransactionManager` object (presumably from JNDI through `JndiObjectFactoryBean` or `<jee:jndi-lookup>`) and feeding it, for example, to Spring's `JtaTransactionManager`, then the easiest way is to specify a reference to the bean defining this JTA `PlatformTransactionManager` instance as the value of the jtaTransactionManager property for `LocalSessionFactoryBean.` Spring then makes the object available to Hibernate.
- More likely you do not already have the JTA `PlatformTransactionManager` instance, because Spring's `JtaTransactionManager` can find it itself. Thus you need to configure Hibernate to look up JTA `PlatformTransactionManager` directly. You do this by configuring an application server- specific `TransactionManagerLookup` class in the Hibernate configuration, as described in the Hibernate manual.

The remainder of this section describes the sequence of events that occur with and without Hibernate's awareness of the JTA `PlatformTransactionManager`.

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

1. The JTA transaction commits.
2. Spring's `JtaTransactionManager` is synchronized to the JTA transaction, so it is called back through an *afterCompletion* callback by the JTA transaction manager.
3. Among other activities, this synchronization can trigger a callback by Spring to Hibernate, through Hibernate's `afterTransactionCompletion` callback (used to clear the Hibernate cache), followed by an explicit `close()` call on the Hibernate Session, which causes Hibernate to attempt to `close()` the JDBC Connection.
4. 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, because the transaction has already been committed.

When Hibernate is configured with awareness of the JTA `PlatformTransactionManager`, the following events occur when a JTA transaction commits:

1. the JTA transaction is ready to commit.
2. Spring's `JtaTransactionManager` is synchronized to the JTA transaction, so the transaction is called back through a *beforeCompletion* callback by the JTA transaction manager.
3. 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.
4. The JTA transaction commits.
5. Hibernate is synchronized to the JTA transaction, so the transaction is called back through an *afterCompletion* callback by the JTA transaction manager, and can properly clear its cache.

15.4 JDO

Spring supports the standard JDO 2.0 and 2.1 APIs as data access strategy, following the same style as the Hibernate support. The corresponding integration classes reside in the `org.springframework.orm.jdo` package.

15.4.1 `PersistenceManagerFactory` setup

Spring provides a `LocalPersistenceManagerFactoryBean` class that allows you to define 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, you can set up a `PersistenceManagerFactory` through direct instantiation of a `PersistenceManagerFactory` implementation class. A JDO `PersistenceManagerFactory` implementation class follows the JavaBeans pattern, just like a JDBC `DataSource` implementation class, which is a natural fit for a configuration that uses Spring. This setup style usually supports a Spring-defined JDBC `DataSource`, passed into the `connectionFactory` property. For example, for the open source JDO implementation DataNucleus (formerly JPOX) (http://www.datanucleus.org/), this is the XML configuration of the `PersistenceManagerFactory` implementation:

```
<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.datanucleus.jdo.JDOPersistenceManagerFactory" destroy-method="close">
   <property name="connectionFactory" ref="dataSource"/>
   <property name="nontransactionalRead" value="true"/>
 </bean>

</beans>
```

You can also set up JDO `PersistenceManagerFactory` in the JNDI environment of a Java EE application server, usually through the JCA connector provided by the particular JDO implementation. Spring's standard `JndiObjectFactoryBean` or `<jee:jndi-lookup>` can be used to retrieve and expose such a `PersistenceManagerFactory`. However, outside an EJB context, no real benefit exists in holding the `PersistenceManagerFactory` in JNDI: only choose such a setup for a good reason. See Section 15.3.6, “Comparing container-managed and locally defined resources” for a discussion; the arguments there apply to JDO as well.

15.4.2 Implementing DAOs based on the plain JDO API

DAOs can also be written directly against plain JDO API, without any Spring dependencies, by using an injected `PersistenceManagerFactory`. The following is an example of a corresponding DAO implementation:

```
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();
        }
    }
}
```

Because the above DAO follows the dependency injection pattern, it fits nicely into a Spring container, just as it would if coded against Spring's `JdoTemplate`:

```
<beans>

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

</beans>
```

The main problem with such DAOs is that they always get a new `PersistenceManager` from the factory. To access a Spring-managed transactional `PersistenceManager`, define a `TransactionAwarePersistenceManagerFactoryProxy` (as included in Spring) in front of your target `PersistenceManagerFactory`, then passing a reference to that proxy into your DAOs as in the following example:

```
<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 receive a transactional `PersistenceManager` (if any) from the `PersistenceManagerFactory.getPersistenceManager()` method that it calls. The latter method call goes through the proxy, which first checks for a current transactional `PersistenceManager` before getting a new one from the factory. Any `close()` calls on the `PersistenceManager` are ignored in case of a transactional `PersistenceManager`.

If your data access code always runs 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 do 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 that you enforce active transactions through turning off `TransactionAwarePersistenceManagerFactoryProxy`'s `allowCreate` flag:

```
<beans>
```

```
<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 fatal, unless you 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 trade off might be acceptable to applications that are strongly JDO-based and/or do not need any special exception treatment.

In summary, you can DAOs based on the plain JDO API, and they can still participate in Spring-managed transactions. This strategy might appeal to you if you are already familiar with JDO. However, such DAOs throw plain `JDOException`, and you would have to convert explicitly to Spring's `DataAccessException` (if desired).

15.4.3 Transaction management

You are *strongly* encouraged to read Section 12.5, “Declarative transaction management” if you have not done so, to get a more detailed coverage of Spring's declarative transaction support.

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.xsd
   http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop.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>
```

JDO requires an active transaction to modify a persistent object. The non-transactional flush concept does not exist in JDO, in contrast to Hibernate. For this reason, you need to set up the chosen JDO implementation for a specific environment. Specifically, you need to set it up explicitly 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 to participate in JTA transactions, whether driven by Spring's `JtaTransactionManager` or by EJB CMT and 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 the case for JDBC-based JDO 2.0 implementations by default.

15.4.4 `JdoDialect`

As an advanced feature, both `JdoTemplate` and `JdoTransactionManager` support a custom `JdoDialect` that can be passed into the `jdoDialect` bean property. In this scenario, the DAOs will not receive a `PersistenceManagerFactory` reference but rather a full `JdoTemplate` instance (for example, passed into the `jdoTemplate` property of `JdoDaoSupport`). Using a `JdoDialect` implementation, you can enable 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, which are automatically calculated from Spring-managed transaction timeouts
- Eagerly flushing a `PersistenceManager,` to make transactional changes visible to JDBC-based data access code
- Advanced translation of `JDOExceptions` to Spring `DataAccessExceptions`

See the `JdoDialect` Javadoc for more details on its operations and how to use them within Spring's JDO support.

15.5 JPA

The 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.

15.5.1 Three options for JPA setup in a Spring environment

The Spring JPA support offers three ways of setting up the JPA `EntityManagerFactory` that will be used by the application to obtain an entity manager.

`LocalEntityManagerFactoryBean`

Only use this option in simple deployment environments such as stand-alone applications and integration tests.

The `LocalEntityManagerFactoryBean` creates an `EntityManagerFactory` suitable for simple deployment environments where the application uses only JPA for data access. The factory bean uses the JPA `PersistenceProvider` autodetection mechanism (according to JPA's Java SE bootstrapping) and, in most cases, requires you to specify only the persistence unit name:

```
<beans>

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

</beans>
```

This form of JPA deployment is the simplest and the most limited. You cannot refer to an existing JDBC `DataSource` bean definition and no support for global transactions exists. Furthermore, weaving (byte-code transformation) of persistent classes is provider-specific, often requiring a specific JVM agent to specified on startup. This option is sufficient only for stand-alone applications and test environments, for which the JPA specification is designed.

Obtaining an `EntityManagerFactory` from JNDI

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.

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

```
<beans>

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

</beans>
```

This action assumes standard Java EE 5 bootstrapping: the Java EE server autodetects persistence units (in effect, `META-INF/persistence.xml` files in application jars) and `persistence-unit-ref` entries in the Java EE deployment descriptor (for example, `web.xml`) and defines 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 through dependency injection, and managing transactions for the persistence unit, typically through `JtaTransactionManager`.

If multiple persistence units are used in the same application, the bean names of such JNDI-retrieved persistence units should match the persistence unit names that the application uses to refer to them, for example, in `@PersistenceUnit` and `@PersistenceContext` annotations.

`LocalContainerEntityManagerFactoryBean`

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

The `LocalContainerEntityManagerFactoryBean` gives full control over `EntityManagerFactory` configuration and is appropriate for environments where fine-grained customization is required. The `LocalContainerEntityManagerFactoryBean` creates a `PersistenceUnitInfo` instance based on the `persistence.xml` file, the supplied `dataSourceLookup` strategy, and the specified `loadTimeWeaver`. It is thus possible to work with custom data sources outside of JNDI and to control the weaving process. The following example shows a typical bean definition for a `LocalContainerEntityManagerFactoryBean`:

```
<beans>

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

</beans>
```

The following example shows a typical `persistence.xml` file:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">

  <persistence-unit name="myUnit" transaction-type="RESOURCE_LOCAL">
    <mapping-file>META-INF/orm.xml</mapping-file>
    <exclude-unlisted-classes/>
  </persistence-unit>

</persistence>
```

The `exclude-unlisted-classes` element always indicates that *no* scanning for annotated entity classes is supposed to occur, in order to support the `<exclude-unlisted-classes/>` shortcut. This is in line with the JPA specification, which suggests that shortcut, but unfortunately is in conflict with the JPA XSD, which implies `false` for that shortcut. Consequently, `<exclude-unlisted-classes> false </exclude-unlisted-classes/>` is not supported. Simply omit the `exclude-unlisted-classes` element if you want entity class scanning to occur.

Using the `LocalContainerEntityManagerFactoryBean` 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, and so on. However, it also imposes requirements on the runtime environment, such as the availability of a weaving-capable class loader if the persistence provider demands byte-code transformation.

This option may conflict with the built-in JPA capabilities of a Java EE 5 server. In a full Java EE 5 environment, consider obtaining your `EntityManagerFactory` from JNDI. Alternatively, specify a custom `persistenceXmlLocation` on your `LocalContainerEntityManagerFactoryBean` definition, for example, META-INF/my-persistence.xml, and only include a descriptor with that name in your application jar files. Because the Java EE 5 server only looks for default `META-INF/persistence.xml` files, it ignores such custom persistence units and hence avoid conflicts with a Spring-driven JPA setup upfront. (This applies to Resin 3.1, for example.)

When is load-time weaving required?

Not all JPA providers require a JVM agent ; Hibernate is an example of one that does not. If your provider does not require an agent or you have other alternatives, such as 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 whether the environment is a web container or application server. Hooking `ClassTransformers` through a Java 5 agent typically is not efficient. The agents work against the *entire virtual machine* and inspect *every* class that is loaded, which is usually undesirable in a production server environment.

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

Refer to the section called “Spring configuration” in the AOP chapter for more insight regarding the `LoadTimeWeaver` implementations and their setup, either generic or customized to various platforms (such as Tomcat, WebLogic, OC4J, GlassFish, Resin and JBoss).

As described in the aforementioned section, you can configure a context-wide `LoadTimeWeaver` using the `@EnableLoadTimeWeaving` annotation of `context:load-time-weaver` XML element. Such a global weaver is picked up by all JPA `LocalContainerEntityManagerFactoryBeans` automatically. This is the preferred way of setting up a load-time weaver, delivering autodetection of the platform (WebLogic, OC4J, GlassFish, Tomcat, Resin, JBoss or VM agent) and automatic propagation of the weaver to all weaver-aware beans:

```
<context:load-time-weaver/>
<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    ...
</bean>
```

However, if needed, one can manually specify a dedicated weaver through the `loadTimeWeaver` property:

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

```
  </property>
</bean>
```

No matter how the LTW is configured, using this technique, JPA applications relying on instrumentation can run in the target platform (ex: Tomcat) without needing an agent. This is important especially when the hosting applications rely on different JPA implementations because the JPA transformers are applied only at class loader level and thus are isolated from each other.

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 to avoid the persistence units discovery process, which can be expensive. The default implementation allows multiple locations to be specified that are parsed and later retrieved through the persistence unit name. (By default, the classpath is searched for `META-INF/persistence.xml` files.)

```
<bean id="pum" class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager">
  <property name="persistenceXmlLocations">
    <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 name="dataSources">
   <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="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
```

```
  <property name="persistenceUnitManager" ref="pum"/>
  <property name="persistenceUnitName" value="myCustomUnit"/>
</bean>
```

The default implementation allows customization of the `PersistenceUnitInfo` instances, before they are fed to the JPA provider, declaratively through its properties, which affect *all* hosted units, or programmatically, through the `PersistenceUnitPostProcessor`, which allows persistence unit selection. If no `PersistenceUnitManager` is specified, one is created and used internally by `LocalContainerEntityManagerFactoryBean`.

15.5.2 Implementing DAOs based on plain JPA

Although `EntityManagerFactory` instances are thread-safe, `EntityManager` instances are not. The injected JPA `EntityManager` behaves like an `EntityManager` fetched from an application server's JNDI environment, as defined by the JPA specification. It delegates all calls to the current transactional `EntityManager`, if any; otherwise, it falls back to a newly created `EntityManager` per operation, in effect making its usage thread-safe.

It is possible to write code against the plain JPA without any Spring dependencies, by using an injected `EntityManagerFactory` or `EntityManager`. Spring can understand `@PersistenceUnit` and `@PersistenceContext` annotations both at field and method level if a `PersistenceAnnotationBeanPostProcessor` is enabled. A plain JPA DAO implementation using the `@PersistenceUnit` annotation 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. 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>
```

As an alternative to defining a `PersistenceAnnotationBeanPostProcessor` explicitly, consider using the Spring `context:annotation-config` XML element in your application context configuration. Doing so automatically registers all Spring standard post-processors for annotation-based configuration, including `CommonAnnotationBeanPostProcessor` and so on.

```
<beans>

  <!-- post-processors for all standard config annotations -->
```

```
  <context:annotation-config/>

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

</beans>
```

The main problem with such a DAO is that it always creates a new `EntityManager` through the factory. You can avoid this by requesting a transactional `EntityManager` (also called "shared EntityManager" because 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();
    }
}
```

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, hence the expressions *method-level injection* and *field-level injection*. Field-level annotations are concise and easier to use while method-level allows for further processing of the injected dependency. In both cases the member visibility (public, protected, private) does not matter.

What about class-level annotations?

On the Java EE 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 DAO implementation uses 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 only 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 appealing from a non-invasiveness perspective, and might feel more natural to JPA developers.

15.5.3 Transaction Management

You are *strongly* encouraged to read Section 12.5, “Declarative transaction management” if you have not done so, to get a more detailed coverage of Spring's declarative transaction support.

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.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.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.

15.5.4 `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 do not receive an `EntityManagerFactory` reference but rather a full `JpaTemplate` instance (for example, passed into the jpaTemplate property `of JpaDaoSupport`). 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. The default implementation used (`DefaultJpaDialect`) does not provide any special capabilities and if the above features are required, you have to specify the appropriate dialect.

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

15.6 iBATIS SQL Maps

The iBATIS support in the Spring Framework much resembles the JDBC support in that it supports the same template style programming, and as with JDBC and other ORM technologies, the iBATIS support works with Spring's exception hierarchy and lets you enjoy Spring's IoC features.

Transaction management can be handled through Spring's standard facilities. No special transaction strategies are necessary for iBATIS, because no special transactional resource involved other than a JDBC `Connection`. Hence, Spring's standard JDBC `DataSourceTransactionManager` or `JtaTransactionManager` are perfectly sufficient.

Spring supports iBATIS 2.x. The iBATIS 1.x support classes are no longer provided.

15.6.1 Setting up the `SqlMapClient`

Using iBATIS SQL Maps involves creating SqlMap configuration files containing statements and result maps. Spring takes care of loading those using the `SqlMapClientFactoryBean`. For the examples we will be using the following `Account` class:

```
public class Account {

    private String name;
    private String email;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

To map this `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. This is the configuration needed for these bean definitions:

```
<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>
```

15.6.2 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>
```

An `SqlMapTemplate` instance can also be created manually, passing in the `SqlMapClient` as constructor argument. The `SqlMapClientDaoSupport` base class simply preinitializes a `SqlMapClientTemplate` instance for us.

The `SqlMapClientTemplate` 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 thrown `SQLException` is converted automatically to Spring's generic `DataAccessException` hierarchy.

15.6.3 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`. The following example shows a corresponding DAO implementation:

```
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 this scenario, you need to handle the `SQLException` thrown by the iBATIS API in a custom fashion, usually by wrapping it in your own application-specific DAO exception. Wiring in the application context would still look like it does in the example for the `SqlMapClientDaoSupport`, 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>
```

Prev Up Next

14. Data access with JDBC Home 16. Marshalling XML using O/X Mappers