Bringing Advanced Transaction Management Capabilities to Spring Applications

Frances Zhao, Paul Parkinson
May 2007

Abstract

The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

- Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Oracle TopLink, Hibernate, JPA, and JDO.
- Supports declarative transaction management.
- Provides a simple API for programmatic transaction management rather than a number of complex transaction APIs such as JTA.
- Integrates very well with Spring's various data access abstractions.[1]

This article discusses Spring's transaction management facilities and the common use cases in Spring where an external transaction manager is required. An application is used to illustrate the transactional aspects and features. The focus is on leveraging JTA transaction management in the Spring framework for enterprise applications. The article shows how Spring's transaction services can seamlessly expose and interact with a JAVA EE application server's transaction manager such as Oracle Application Server and the OC4JJtaTransactionManager.

Let us start with an overview of the related technologies, Spring basics, transactioning basics, and transactioning features within Spring. Then we will look at the implementation strategy and use the sample application to show how it works together in more detail.

1. Spring Basics and Features

As a JAVA EE developer, you may be thinking "not another framework."[2] The Spring framework simplifies development with its modular architecture and handles configuration in a consistent manner. It achieves this simplification by its use of inversion of control that allows enterprise functionality to be built into POJOs thus making it powerful as well. Spring is a state-of-the-art technology in terms of making JAVA EE and other existing technologies easier to use and provides an abstract level for use of JTA or other transaction strategies as well as other JAVA EE components such as data sources.

---

[1] Excerpt from the Spring Framework Transaction Management documentation
[2] Rod Johnson used “not another framework” in his Spring framework article

The Spring framework can be integrated with different application servers, such as Oracle Application Server, WebLogic, and WebSphere.

Spring provides many features. Let's look at the following major areas in detail.

1.1 Inversion of Control Container and Dependency Injection

First, let's take a look at how to simplify development by using inversion of control and dependency injection.

A main abstraction of inversion of control is the bean factory, which is a generic factory that enables objects to be retrieved by name and which manages the relationships between objects.

As Rod Johnson explained in his article on Spring framework, the concept behind inversion of control is often expressed in the Hollywood Principal: "Don't call me, I'll call you." Inversion of control moves the responsibility for making things happen into the framework and away from application code.

Dependency injection is a form of inversion of control that removes explicit dependence on container APIs. Ordinary Java methods are used to inject dependencies such as collaborating objects or configuration values into application object instances.

Dependency injection is not a new concept, although it's only recently made prime time in the JAVA EE community. The definition of dependency injection between the JAVA EE community and the Spring framework is the same but obtained via different mechanisms.

1.2 XML Bean Definitions (ApplicationContext)

You can configure your Spring applications in XML bean definition files that are in some ways similar to the JAVA EE platform where you use XML configuration files and XML deployment descriptors to define the relationship of the resources and how they are to be deployed. The root of the XML bean definition is a <beans> element. that can contain one or more <bean> definitions.

The following example shows the configuration of the application objects, which is similar to the objects relationship we are familiar with in JAVA EE applications. We will define a JAVA EE DataSource, bankDataSource; a DAO, bankDAO; and a business object that uses the DAO, assetManagementService. The following examples are from the sample bank account transfer application that shows the relationships between bankDataSource, bankDAO, and assetManagementService.

First let us look at the bankDataSource definition in XML format. We could use Spring's JNDI location FactoryBean to get the data source from Oracle Application Server, as follows. There would be no impact on Java code or any other bean definitions.

```
<beans>
	<bean id="bankDataSource"
		class="org.springframework.jndi.JndiObjectFactoryBean">
		<property name="jndiName">
			<value>jdbc/bankDataSource</value>
		</property>
	</bean>
<beans>
```

Now, we define the DAO object that has a bean reference to the bankDataSource. Relationships between beans are specified using the “ref” attribute or <ref> element.

```
<beans>
	<bean id="bankDAO" class="how.to.spring.tx.BankImpl">
		<property name="dataSource">
			<ref bean="bankDataSource"/>
		</property>
	</bean>
<beans>
```

The business object, assetManagementService, has a reference to the DAO as in the following example.

```
<beans>
	<bean id="assetManagementService"
		class="how.to.spring.tx.AssetManagementServiceImpl">
		<property name="brokerage">
			<ref local="brokerageDAO"/>
		</property>
		<property name="bank">
			<ref local="bankDAO"/>
		</property>
	</bean>
<beans>
```

1.3 ContextLoaderListener and DispatcherServlet

ContextLoaderListener is the bootstrap listener to start up Spring's root webApplicationContext when integrating with a JAVA EE Web container. As the example shows, the JAVA EE standard web-app descriptor, web.xml, can include a Spring ContextLoaderListener listener that causes the WEB-INF/applicationContext.xml

specified by the contextConfigLocation <context-param> to be loaded by the Spring framework. The Spring DispatcherServlet Servlet deployed with the servlet-name jta-spring causes the jta-spring-servlet.xml to be loaded by the Spring framework.

```
<web-app>
   <display-name>JTA Spring Integration WebApp</display-name>
   <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>WEB-INF/applicationContext.xml</param-value>
   </context-param>

  <listener>
        <listener- class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

   <servlet>
      <servlet-name>jta-spring</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
            <load-on-startup>2</load-on-startup>
   </servlet>

   <servlet-mapping>
            <servlet-name>jta-spring</servlet-name>
            <url-pattern>JTADispatcherServlet</url-pattern>
   </servlet-mapping>
</web-app>
```

2. Transactioning Basics

When purchasing stocks from a broker, money is transferred from a bank account to the brokerage. A series of related operations ensures that the stocks are added to the purchaser's portfolio and the brokerage, in turn, receives the purchase money. If a single operation in the series fails during the exchange, the entire exchange fails. You do not get the stocks, and the broker does not get your money. Transaction processing makes the exchange balanced and predictable even in the face of failures in any of the systems and resources involved.

2.1 ACID Properties

Transaction processing systems provide the guarantee of ACID properties.

ACID properties include Atomicity, Consistency, Isolation and Durability.

- Atomicity: All changes within the scope of a transaction (the unit of work) are either committed or rolledback.

- Consistency: The state (data) of the system moves from valid state to another from the beginning of the transaction to its completion.
- Isolation: The effects of one transaction are not visible to another until the transaction completes.
- Durability: Changes made within the scope of the transaction must be made permanent.

It's entirely possible to make these guarantees without any supporting infrastructure, but this would require a considerable amount of error prone and repetitive work by the application developer and generally a less flexible design. Transaction processing systems, and the application servers they run within, provide this service implicitly.

Many transactioning systems and applications allow for the relaxing of one or more of the ACID properties. This is most often done in order to provide better performance where a risk assessment has been conducted and/or an acceptable tolerance established. Isolation is the most commonly relaxed property

2.2 Isolation Levels

An isolation level defines how concurrent transactions that access a shared resource are isolated from one another for read purposes.

Dirty reads, non-repeatable reads, and phantom reads are the three main conditions where an application reads data within a transaction that has been altered outside of the transaction.

Dirty reads occur when data that has been updated in a transaction and not yet committed is read by another transaction.

Non-repeatable reads occur when a transaction reads data, a second transaction subsequently updates that data, and the first transaction reads the data again after the second transaction's update.

Phantom reads occur when a transaction reads a range of data (rows), a second transaction subsequently deletes or inserts data (a row) in this range, and the first transaction reads the range of data again after the second transaction's delete or insert.

The read issues described may or may not be a concern for an application. Whether it is a problem or not depends entirely upon the business context. There are also resource and performance costs involved that must be considered in the design of the system. Isolation levels can be used to prevent (or allow) these situations.

The most common use of isolation levels is when accessing a database. Different vendors have proprietary isolation levels, locking mechanisms, and other behaviors that are well beyond the scope of this article and so we will briefly explain the standard

isolation levels defined in the JDBC API only. Note that this list is provided in order of the weakest to strongest isolation with an inverse correlation as far as performance is concerned.

- TRANSACTION_NONE: transactions are not supported.
- TRANSACTION_READ_UNCOMMITTED: dirty reads, non-repeatable reads and phantom reads can occur.
- TRANSACTION_READ_COMMITTED: dirty reads are prevented; non-repeatable reads and phantom reads can occur.
- TRANSACTION_REPEATABLE_READ: reads and non-repeatable reads are prevented; phantom reads can occur.
- TRANSACTION_SERIALIZABLE: dirty reads, non-repeatable reads and phantom reads are prevented.

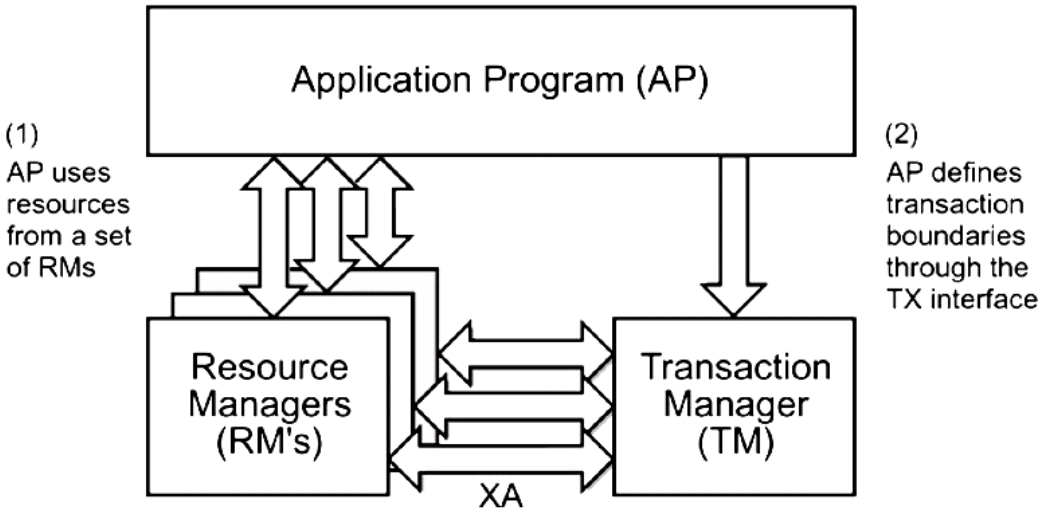
2.3 Local Transactions and JTA Global Transactions

The complexity of a transaction increases with the number of resources the application enlists within the transaction.

A local transaction involves only one resource and the transaction activity is scoped and coordinated locally to the resource itself.

A global transaction may enlist more than one resource manager including multiple databases, message systems, legacy mainframe systems, etc.. In order to achieve atomic outcomes in the global transaction, coordination between a transaction Manager and these resources is required. This coordination is achieved via the distributed transaction protocol defined in the DTP and XA specifications published by the OpenGroup.

The following diagram is a DTP model that illustrates the relationship and interaction between the application, resource managers, and the transaction manager.



Applications communicate with the transaction manager to begin and end transactions and to enlist resources. When the application requests that a transaction be committed, the transaction manager (such as Oracle Application Server, , WebLogic, WebSphere) coordinates the two-phase commit protocol. The transaction manager mediates between applications and resource managers to delineate the boundaries of units of work and to perform a termination protocol that communicates the outcome of the transaction to all participants.

A component can begin a JTA transaction programmatically using the UserTransaction interface or it can be started by the EJB container as specified in the transaction-attributes of the EJB's deployment descriptor.

3. Why Spring with JTA TransactionManager?

The transaction requirements for a majority of applications are met by either a Spring transaction management strategy or a JTA transaction manager in the middle tier. There are cases where applications must support remote calls where a transaction context is propagated over multiple processes in which case using the facilities inherent in the EJB distributed component model with container managed JTA transactioning support is appropriate. If you are looking for a JAVA EE framework that provides declarative transaction management and a flexible persistence engine, Spring is a great choice. It lets you choose the features you want without the added complexities of EJB.

Like JAVA EE, Spring provides support for programmatic transaction demarcation, however, a more dynamic application design comes from use of demarcation specified either by annotation or AOP. These two techniques are illustrated in the example provided and complete details can be found in the Spring documentation.

Another unique and clever feature provided by Spring is the ability to switch from using a local transaction (manager), such as a database datasource, to a JTA transaction manager by simply manipulating a few lines of configuration in order to use a different PlatformTransactionManager implementation. Migration of this nature, that is the need to add another resource within the scope of a transaction previously designed for only resource local transactional work, is very common in the extended lifecycle of an application and generally non-trivial and dangerously error-prone in conventional JAVA EE applications. Note that the datasource being used in this scenario must be one that has a contract with the JTA implementation being used in order to provide enlistment in the global transaction. This is generally the case if the (XA) datasource is obtained from the application server using JNDI. This is shown in the example provided by using Spring's JndiObjectFactoryBean that frees the application code from JNDI dependency.

Most enterprise applications such as high-end financial applications and highly available telecom systems require comprehensive transactional support. By using Spring in tandem with an enterprise JTA implementation, powerful (and generally proprietary) quality of service features such as high-availability, clustering and grid support, fail-over, peer recovery, non-stop transactioning, integration, interoperability (such as OTS and WS-TX), monitoring and administration, and other features present in enterprise JTA implementations and application server environments can be exploited while the application itself retains all of the benefits of Spring previously described.

Two examples where Spring has provided support and integration beyond standard JTA are transaction names and per-transaction isolation levels.

Due to the complex nature of transactions, in particular global transactions where a number of systems are involved in a single activity, it becomes critical to have meaningful information for administration, monitoring, and debugging. Named transactions provide a way of identifying and grouping transactions by a type. Spring provides the name of the class and method that initiated the transaction as this transaction name identifier. This is very useful for the reasons mentioned, particularly when this is imported into a transactioning system that can then correlate this information with other data. Suddenly, the enterprise system has a debuggable holistically monitored transactional business process rather than a cryptic log of Xid byte arrays.

As discussed in the section on isolation levels, it becomes apparent that a number of factors including performance, strict versus relaxed ACID requirements, data representation and usage, and even vendor implementation behaviors dictate that different isolation levels must be used for different cases and this leads to the need for fine-grained control of isolation levels (particularly when connection related resources are at a premium– which they generally are). Again this is an area where littering application code with common API calls, in this case to set and reset transaction isolation levels, is not optimal and where Spring presents a dynamic solution by exposing the per-transaction isolation-level features present in some extended JTA implementations. Spring provides this in the exact same fashion as transaction demarcation (where demarcation is specified as a “propagation” attribute and isolation level is specified as an “isolation” attribute). These settings are described in detail in the example application as well as the Spring documentation.

Let’s take a step-by-step look at our sample trading application that showcases the common use cases in Spring where an external transaction manager is required.

Let’s use the Oracle Application Server transaction manager as the example that demonstrates the integration of JTA with Spring's OC4JJtaTransactionManager. The application demonstrates the classic distributed two-phase commit transaction use case requiring ACID properties: the bank account transfer. Funds are debited from one account and credited to another. Either both the debit and credit must occur or neither must occur. In the case of this example, the transfer is from a bank account to a brokerage account in order to purchase individual stocks. The example includes a very simple MVC

style application consisting of a test controller, financial service, asset management service, and two DAO objects representing a bank and a brokerage. Container-manager transactions are used. The example adds additional aspects to this scenario in order to demonstrate the extended features of the OC4JJtaTransactionManager that includes named transactions and per-transaction isolation-level designation.

The following HowToJTASpringController implements the Spring Controller and InitializingBean interfaces. Note that the setFinancial method provides the FinancialService implementation (as specified in applicationContext.xml).

```
public class HowToJTASpringController implements InitializingBean,
Controller {
      private FinancialService m_financial;

      public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
            try {
               FinancialReport financialReport =
        m_financial.processFinancials();
               request.setAttribute("financialReport", financialReport);
               return new ModelAndView("/jsp/success.jsp");
            }
            catch (Exception e) {
               request.setAttribute("error", e.getMessage());
               return new ModelAndView("/jsp/error.jsp");
            }
      }
```

The FinancialServiceImpl class implements the Spring InitializingBean interface as well as the FinancialService interface. The setAssetManagement method is called by the Spring framework, which also provides the AssetManagementService implementation (as specified in applicationContext.xml), using dependency injection. The Transactional class-level annotation (transaction annotation support is specified in applicationContext.xml) designates that business methods of this class, namely processFinancials, have a propagation value of REQUIRED. That is, the methods execute within a transaction if one exists or a transaction is started if none exists. The annotation also specifies that the transaction is to be readOnly and that the isolation-level of any connections used within the transaction are set to SERIALIZABLE.

```
@Transactional(readOnly = true, propagation = Propagation.REQUIRED,
isolation = Isolation.SERIALIZABLE)
public class FinancialServiceImpl implements InitializingBean, FinancialService
{
        AssetManagementService m_assetManagementService;


        public FinancialReport processFinancials() {
            AssetReport assetReportBeforeStockPurchase =
         m_assetManagementService.reportAllAssets();
            StockPurchaseReport stockPurchaseReport =
         m_assetManagementService.purchaseNewStockAndReport();
            AssetReport assetReportAfterStockPurchase =
         m_assetManagementService.reportAllAssets();
            return new FinancialReport(assetReportBeforeStockPurchase,
         stockPurchaseReport, assetReportAfterStockPurchase);
           }


           public final void afterPropertiesSet() throws Exception {
              if (m_assetManagementService == null)
                 throw new BeanCreationException("NoAssetManagementService
         was set.  Verify context xml.");
           }

           public void setAssetManagement(AssetManagementService
         assetManagementService) {
              m_assetManagementService = assetManagementService;
           }
        }
```

The AssetManagementServiceImpl class implements the Spring InitializingBean interface as well as the AssetManagementService interface. The setBank and setBrokerage methods are called by the Spring framework providing the Bank and Brokerage DAO implementations (as specified in applicationContext.xml) using dependency injection.

The Transactional method-level annotation (transaction annotation support is specified in applicationContext.xml) designates that the purchaseNewStockAndReport method has a propagation value of REQUIRED, that is, executes within a transaction if one exists or a transaction is started if none exists. The annotation also specifies the isolation-level of any connections used within the transaction be set to READ_COMMITTED.

Another method-level Transactional annotation designates that the reportAllAssets method has a propagation value of SUPPORTS. That method executes within a transaction if one exists but does not throw an exception or start a transaction if none exists. The annotation also specifies the noRollbackFor be set to ConcurrencyFailureException.class, which indicates that if a transaction exists and this Spring DAO RuntimeException is thrown, the transaction should not rollback as a result.

```
public class AssetManagementServiceImpl implements InitializingBean,
AssetManagementService {
        private Bank m_bank;
        private Brokerage m_brokerage;

        @Transactional(propagation = Propagation.SUPPORTS,
    noRollbackFor = ConcurrencyFailureException.class)
        public AssetReport reportAllAssets() {
            return new AssetReport(m_bank.selectBalance(),
    m_brokerage.selectAllStocks());
        }

        @Transactional(propagation = Propagation.REQUIRES_NEW,
    isolation = Isolation.READ_COMMITTED)
        public StockPurchaseReport purchaseNewStockAndReport() {
            int stockAmount = 10;
            String stockSymbol = "ABC";
            m_bank.updateBalance(m_bank.selectBalanceForUpdate() -
    stockAmount);
            m_brokerage.insertStock(stockSymbol, stockAmount);
            return new StockPurchaseReport(stockSymbol, stockAmount);
        }

        public final void afterPropertiesSet() throws Exception {
            if (m_bank == null) throw new BeanCreationException("No Bank
    was set.  Verify context xml.");
            if (m_brokerage == null) throw new BeanCreationException("No
    Brokerage was set.  Verify context xml.");
        }
    public void setBank(Bank bank) {
            m_bank = bank;
        }

        public void setBrokerage(Brokerage brokerage) {
            m_brokerage = brokerage;
        }
```

```
}
```

The BankImpl class extends the Spring JdbcDaoSupport class and uses the Spring JdbcTemplate to act upon the bankDataSource datasource.

```
public class BankImpl extends JdbcDaoSupport implements Bank {

        public int selectBalance() {
            return getJdbcTemplate().queryForInt("select balance from bank
    where account = '101'");
        }

        public int selectBalanceForUpdate() {
            return getJdbcTemplate().queryForInt("select balance from bank
    where account = '101' for update");
        }

        public void updateBalance(int amount) {
            getJdbcTemplate().execute("update bank set balance = " + amount +
    " where account = '101'");
        }
}
```

The BrokerageImpl class extends the Spring JdbcDaoSupport class and uses the Spring JdbcTemplate to act upon the brokerageDataSource datasource.

```
public class BrokerageImpl extends JdbcDaoSupport implements Brokerage {

        public List selectAllStocks() {
            return getJdbcTemplate().queryForList("select * from brokerage");
        }

        public void insertStock(String symbol, int amount) {
            getJdbcTemplate().execute("insert into brokerage values
```

```
        ('"+symbol+"', "'+amount+"' )");
            }
}
```

Let's look at the configuration files. The JAVA EE standard web-app descriptor web.xml includes a Spring ContextLoaderListener. The ContextLoaderListener causes the WEB-INF/applicationContext.xml specified by the contextConfigLocation context-param to be loaded by the Spring framework.

The Spring DispatcherServlet servlet deployed with the servlet-name jta-spring causes the jta-spring-servlet.xml to be loaded by the Spring framework.

```
<web-app>
  <display-name>JTA Spring Integration WebApp</display-name>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>WEB-INF/applicationContext.xml</param-value>
  </context-param>

   <listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
   </listener>

   <servlet>
        <servlet-name>jta-spring</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>jta-spring</servlet-name>
        <url-pattern>JTADispatcherServlet</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
        <welcome-file>index.html</welcome-file>
```

```
        </welcome-file-list>
    </web-app>
```

The descriptor jta-spring-servlet.xml contains a bean definition for the HowToJTASpringController, namely the financialService bean named financial (the property name corresponds to the setter in HowToJTASpringController).

```
<beans>

    <bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="urlMap">
    <map>
        <entry key="*"><ref local="testController"/></entry>
    </map>
      </property>
    </bean>

    <bean id="testController"
class="how.to.spring.tx.HowToJTASpringController">
        <property name="financial"><ref bean="financialService"/></property>
   </bean>

</beans>
```

The descriptor ApplicationContext.xml contains bean definitions for the FinancialServiceImpl, AssetManagementServiceImpl, BankImpl, and BrokerageImpl classes. The <tx:annotation-driven/> element specifies support for annotationdriven demarcation of transactions.

Finally, the descriptor specifies OC4JJtaTransactionManager as the transactionManager to be used.

```
<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="attributes"
class="org.springframework.metadata.commons.CommonsAttributes"/>

  <bean id="bankDAO" class="how.to.spring.tx.BankImpl">
    <property name="dataSource">
      <ref bean="bankDataSource"/>
    </property>
  </bean>

  <bean id="bankDataSource"
class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
      <value>jdbc/bankDataSource</value>
    </property>
  </bean>

  <bean id="brokerageDAO" class="how.to.spring.tx.BrokerageImpl">
    <property name="dataSource">
      <ref bean="brokerageDataSource"/>
  </property>
  </bean>

<bean id="brokerageDataSource"
class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
      <value>jdbc/brokerageDataSource</value>
    </property>
  </bean>

  <bean id="assetManagementService"
class="how.to.spring.tx.AssetManagementServiceImpl">
    <property name="brokerage">
      <ref local="brokerageDAO"/>
    </property>
```

```
      <property name="bank">
        <ref local="bankDAO"/>
      </property>
    </bean>

    <bean id="financialService"
class="how.to.spring.tx.FinancialServiceImpl">
      <property name="assetManagement">
        <ref local="assetManagementService"/>
      </property>
    </bean>

    <!-- enable the configuration of transactional behavior based on
annotations -->
    <tx:annotation-driven/>

    <bean id="transactionManager"
class="org.springframework.transaction.jta.OC4JJtaTransactionManager"
/>

<!-- enable the configuration of transactional behavior based on xml
    <aop:config>
      <aop:pointcut id="financialOperations" expression="execution(*
how.to.spring.tx.*.*(..))"/>
      <aop:advisor pointcut-ref="financialOperations" advice-
ref="txAdvice"/>
    </aop:config>
    <tx:advice id="txAdvice">
      <tx:attributes>
        <tx:method name="processFinancials" read-only="true"
propagation = "REQUIRED" isolation = "SERIALIZABLE"/>
        <tx:method name="reportAllAssets" propagation = "SUPPORTS"
no-rollback-
for="org.springframework.dao.ConcurrencyFailureException"/>
        <tx:method name="purchaseNewStockAndReport" propagation =
"REQUIRES_NEW" isolation = "READ_COMMITTED"/>
      </tx:attributes>
    </tx:advice>
-->
</beans>
```

In this example we have discussed how to develop a simple Spring enabled JTA application using the OC4JJtaTransactionManager to automatically provide named transactions and per-transaction isolation levels.

5. Summary

Spring aims to make JAVA EE development easier by using inversion of control as one of its central features. This enables you to develop enterprise applications using simple Java objects that collaborate with each other through interfaces. These beans are wired together at runtime by the Spring container.

Staying true to aiding enterprise development and filling out its support for the middle tier, Spring offers integration with various JAVA EE services. Spring offers integration to several transaction strategies and supports a variety of transaction scenarios, including integration with enterprise JTA transactioning systems.