Spring Hibernate Integration & Tx Mgt

Rajeev Gupta MTech CS Java Trainer
& Consultant

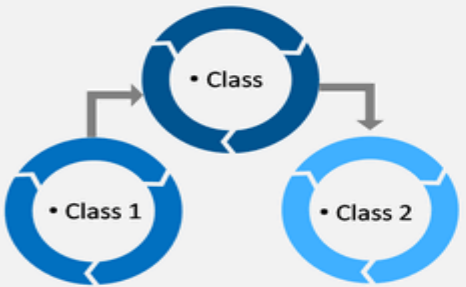
Hibernate framework Overview

Hibernate ORM

OBJECT RELATIONAL MAPPING

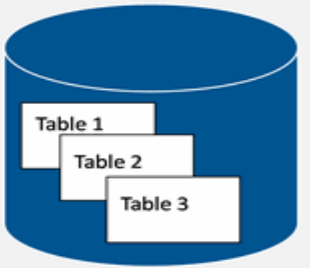
What Does ORM Do ?

Object Model



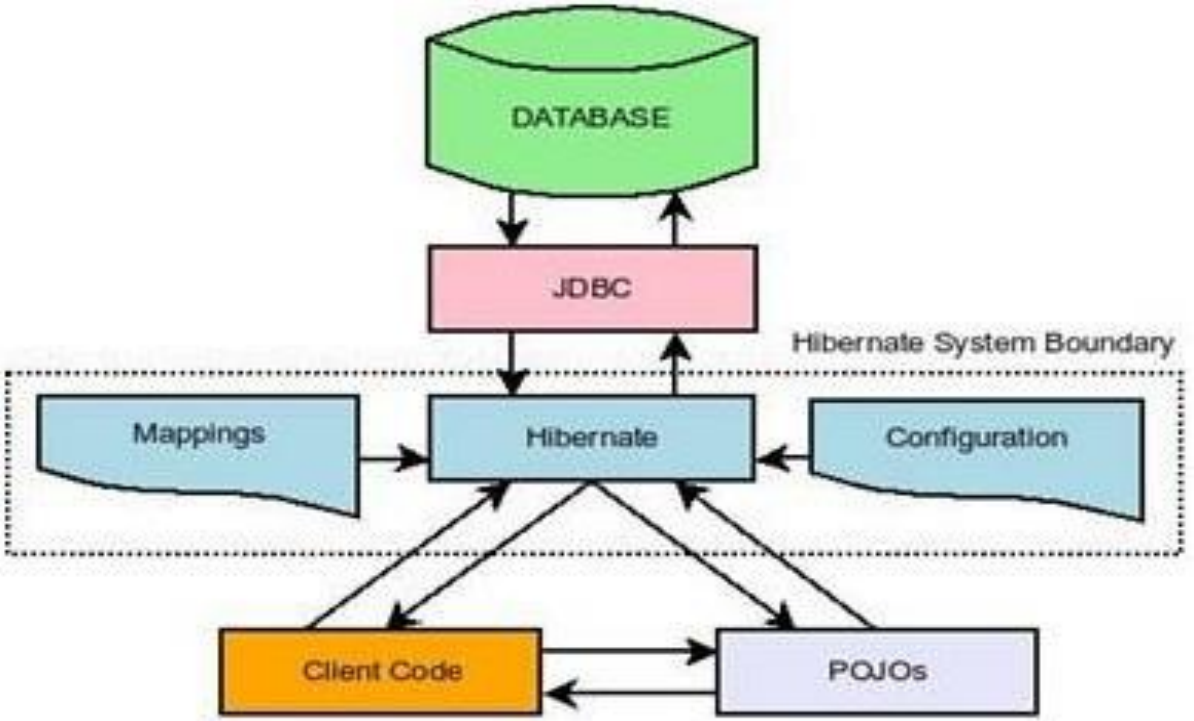
ORM

Data Model



- Maps Object Model to Relational Model.
- Resolve impedance mismatch
- Resolve mapping of scalar and non-scalar.
- Database – Independent applications.

Hibernate **architecture**



Hibernate **Configuration**

```
ServiceRegistry serviceRegistry=new StandardServiceRegistryBuilder().configure().build();
SessionFactory factory=new MetadataSources(serviceRegistry).buildMetadata().buildSessionFactory();

Session  session=factory.openSession();

Transaction tx=session.getTransaction();

Account account=new Account("ekta", 6000);
try {
tx.begin();
session.save(account);
tx.commit();
}catch(HibernateException ex) {
    ex.printStackTrace();
    tx.rollback();
}
```

Hibernate 5 configuration

Hibernate Spring Integration

Problem with hibernate without spring

Problem with hibrenate without spring

Each DAO method must:

---

1. Obtain a EntityManager/session factory instance
2. Start a transaction
3. Perform the persistence operation
4. commit the transaction.
5. Each DAO method should include its own duplicated exception-handling implementation.

These are exactly the problems that motivate us to use Spring with Hibernate

---

"template design patten"

Spring Hibernate **Configuration**

```
<context:annotation-config />

<context:component-scan base-package="com.bankapp" />

<bean id="dataSoruce"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="url" value="${url}" />
    <property name="driverClassName" value="${driverClassName}" />
    <property name="username" value="${username}" />
    <property name="password" value="${password}" />
</bean>

<bean
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="classpath:db.properties"></property>
</bean>
```

Spring Hibernate **Configuration**

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSoruce" />
    <property name="packagesToScan">
        <list>
            <value>com.bankapp.model.persistance</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
        </props>
    </property>
</bean>

<bean id="transactionManager"
    class="org.springframework.orm.hibernate4.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

<tx:annotation-driven transaction-manager="transactionManager" />
```

Spring Hibernate **Configuration**

```java
@Configuration
@EnableTransactionManagement
@ComponentScan({ "com.yms" })
@PropertySource(value = { "classpath:application.properties" })
public class HibernateConfiguration {
    @Autowired
    private Environment environment;
    @Bean
    @Autowired
    public LocalSessionFactoryBean sessionFactory(DataSource ds) {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(ds);
        sessionFactory
                .setPackagesToScan(new String[] { "com.yms.bankapp.pesistance" });
        sessionFactory.setHibernateProperties(hibernateProperties());
        return sessionFactory;
    }

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(environment
                .getRequiredProperty("jdbc.driverClassName"));
        dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));
        dataSource
                .setUsername(environment.getRequiredProperty("jdbc.username"));
        dataSource
                .setPassword(environment.getRequiredProperty("jdbc.password"));
        return dataSource;
    }
```

Spring Hibernate **Configuration**

```
private Properties hibernateProperties() {
    Properties properties = new Properties();
    properties.put("hibernate.dialect",
            environment.getRequiredProperty("hibernate.dialect"));
    properties.put("hibernate.show_sql",
            environment.getRequiredProperty("hibernate.show_sql"));
    properties.put("hibernate.format_sql",
            environment.getRequiredProperty("hibernate.format_sql"));
    properties.put("hibernate.hbm2ddl.auto",
            environment.getRequiredProperty("hibernate.hbm2ddl.auto"));
    return properties;
}

@Bean
@Autowired
public HibernateTransactionManager transactionManager(SessionFactory s) {
    HibernateTransactionManager txManager = new HibernateTransactionManager();
    txManager.setSessionFactory(s);
    return txManager;
}
```

Spring Tx Mgt

Need of Transaction Management

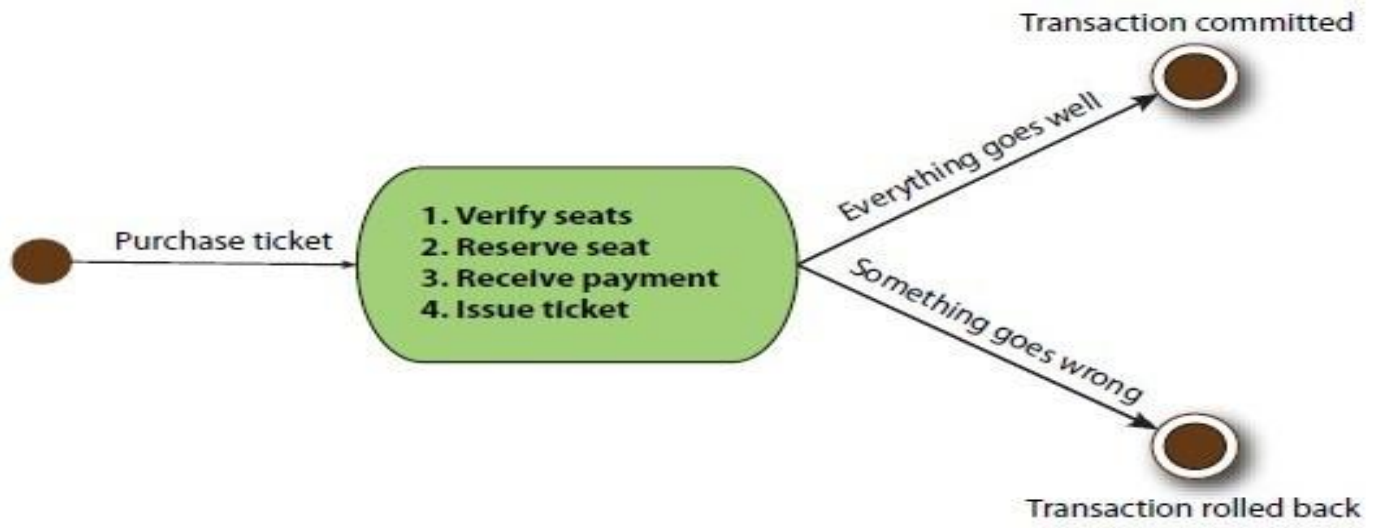


Figure 6.1 **The steps involved when purchasing a movie ticket should be all or nothing. If every step is successful, then the entire transaction is successful. Otherwise, the steps should be rolled back—as if they never happened.**

Transactions

- Transaction is a series of actions that must occur as a group. If any portion of the group of actions fails, the entire Transaction should fails.
- A.C.I.D. is the well-known acronym for the characteristics of a successful and valid transaction.

Atomicity
Consistency
Isolation
Durability

> In software, all-or-nothing operations are called *transactions*. Transactions allow you to group several operations into a single unit of work that either fully happens or fully doesn't happen. If everything goes well, then the transaction is a success. But if anything goes wrong, the slate is wiped clean and it's as if nothing ever happened.

ACID properties

Property	Description
Atomicity	A transaction is composed of one or more operations grouped in a unit of work. At the conclusion of the transaction, either these operations are all performed successfully (commit) or none of them is performed at all (rollback) if something unexpected or irrecoverable happens.
Consistency	At the conclusion of the transaction, the data are left in a consistent state.
Isolation	The intermediate state of a transaction is not visible to external applications.
Durability	Once the transaction is committed, the changes made to the data are visible to other applications.

Spring tx support

- Spring doesn't directly manage transactions.
- Spring comes with a selection of transaction managers that delegate responsibility for transaction management to a platform- specific transaction implementation provided by either JTA or the persistence mechanism.
- Each of these transaction managers acts as a facade to a platform- specific transaction implementation.

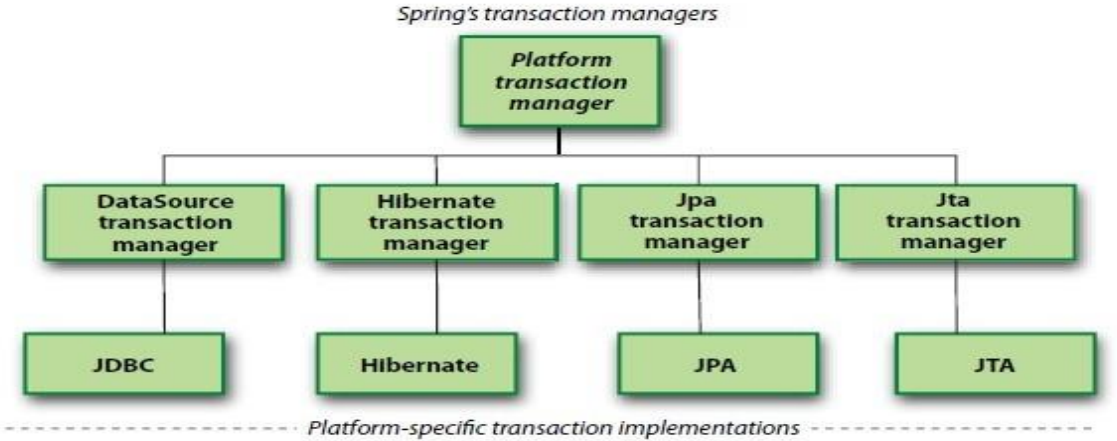


Figure 6.2 Spring's transaction managers delegate transaction-management responsibility to platform-specific transaction implementations.

Spring supported transaction managers

Transaction manager (org.springframework.*)	Use it when...
jca.cci.connection.CciLocalTransactionManager	Using Spring's support for Java EE Connector Architecture (JCA) and the Common Client Interface (CCI).
jdbc.datasource.DataSourceTransactionManager	Working with Spring's JDBC abstraction support. Also useful when using iBATIS for persistence.
jms.connection.JmsTransactionManager	Using JMS 1.1+.
jms.connection.JmsTransactionManager102	Using JMS 1.0.2.
orm.hibernate3.HibernateTransactionManager	Using Hibernate 3 for persistence.
orm.jdo.JdoTransactionManager	Using JDO for persistence.
orm.jpa.JpaTransactionManager	Using the Java Persistence API (JPA) for persistence.
transaction.jta.JtaTransactionManager	You need distributed transactions or when no other transaction manager fits the need.
transaction.jta.OC4JJtaTransactionManager	Using Oracle's OC4J JEE container.
transaction.jta.WebLogicJtaTransactionManager	You need distributed transactions and your application is running within WebLogic.
transaction.jta.WebSphereUowTransactionManager	You need transactions managed by a UOWManager in WebSphere.

JDBC Transaction

- If we are using JDBC, springs DataSourceTransactionManager will handle transactional boundaries for us.
- To use DataSourceTransactionManager, wire it into your application's context definition using the following XML

```
<bean id="transactionManager" class="org.springframework.jdbc.
      ➥datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

- Behind the scenes, DataSourceTransactionManager manages transactions by making calls on the java.sql.Connection object retrieved from the DataSource.
- For instance, a successful transaction is committed by calling the commit() method on the connection. Likewise, a failed transaction is rolled back by calling the rollback() method.

Hibernate transactions

- If we are using Hibernate then we need to configure HibernateTransactionManager

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

- The sessionFactory property should be wired with a Hibernate SessionFactory, here cleverly named sessionFactory

Java Persistence API transactions

- If we are using JPA then we need to configure JpaTransactionManager

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

- We need to provide entitymanagerfactory configureation

Defining transaction attributes

- In Spring, declarative transactions are defined with *transaction attributes.*
- ***A transaction attribute is a description of how transaction policies should be applied to a method.***
- *There are **five** facets of a transaction attribute*
- *Although Spring provides several mechanisms for declaring transactions, all of them rely on these five parameters to govern how transaction policies are administered.*
- *Regardless of which declarative transaction mechanism we use, we have to define these attributes*

Figure 6.3 Declarative transactions are defined in terms of propagation behavior, isolation level, read-only hints, timeout, and rollback rules.

PROPAGATION BEHAVIOR

- *Propagation behaviour defines the boundaries of the transaction with respect to the client and to the method being called.*

Table 6.2 Propagation rules define when a transaction is created or when an existing transaction can be used. Spring provides several propagation rules to choose from.

Propagation behavior	What it means
PROPAGATION_MANDATORY	Indicates that the method must run within a transaction. If no existing transaction is in progress, an exception will be thrown.
PROPAGATION_NESTED	Indicates that the method should be run within a nested transaction if an existing transaction is in progress. The nested transaction can be committed and rolled back individually from the enclosing transaction. If no enclosing transaction exists, behaves like PROPAGATION_REQUIRED. Vendor support for this propagation behavior is spotty at best. Consult the documentation for your resource manager to determine if nested transactions are supported.
PROPAGATION_NEVER	Indicates that the current method shouldn't run within a transactional context. If an existing transaction is in progress, an exception will be thrown.
PROPAGATION_NOT_SUPPORTED	Indicates that the method shouldn't run within a transaction. If an existing transaction is in progress, it'll be suspended for the duration of the method. If using JTATransactionManager, access to TransactionManager is required.
PROPAGATION_REQUIRED	Indicates that the current method must run within a transaction. If an existing transaction is in progress, the method will run within that transaction. Otherwise, a new transaction will be started.
PROPAGATION_REQUIRES_NEW	Indicates that the current method must run within its own transaction. A new transaction is started and if an existing transaction is in progress, it'll be suspended for the duration of the method. If using JTATransactionManager, access to TransactionManager is required.
PROPAGATION_SUPPORTS	Indicates that the current method doesn't require a transactional context, but may run within a transaction if one is already in progress.

ISOLATION LEVELS

- An isolation level defines how much a transaction may be impacted by the activities of other concurrent transactions. Another way to look at a transaction's isolation level is to think of it as how selfish the transaction is with the transactional data.

In a typical application, multiple transactions run concurrently, often working with the same data to get their jobs done. Concurrency, while necessary, can lead to the following problems:

- *Dirty reads* occur when one transaction reads data that has been written but not yet committed by another transaction. If the changes are later rolled back, the data obtained by the first transaction will be invalid.
- *Nonrepeatable reads* happen when a transaction performs the same query two or more times and each time the data is different. This is usually due to another concurrent transaction updating the data between the queries.
- *Phantom reads* are similar to nonrepeatable reads. These occur when a transaction (T1) reads several rows, and then a concurrent transaction (T2) inserts rows. Upon subsequent queries, the first transaction (T1) finds additional rows that weren't there before.

ISOLATION LEVELS

- **Isolation levels determine to what degree a transaction may be impacted by other transactions being performed in parallel.**

Isolation level	What it means
ISOLATION_DEFAULT	Use the default isolation level of the underlying data store.
ISOLATION_READ_UNCOMMITTED	Allows you to read changes that haven't yet been committed. May result in dirty reads, phantom reads, and nonrepeatable reads.
ISOLATION_READ_COMMITTED	Allows reads from concurrent transactions that have been committed. Dirty reads are prevented, but phantom and nonrepeatable reads may still occur.
ISOLATION_REPEATABLE_READ	Multiple reads of the same field will yield the same results, unless changed by the transaction itself. Dirty reads and nonrepeatable reads are prevented, but phantom reads may still occur.
ISOLATION_SERIALIZABLE	This fully ACID-compliant isolation level ensures that dirty reads, nonrepeatable reads, and phantom reads are all prevented. This is the slowest of all isolation levels because it's typically accomplished by doing full table locks on the tables involved in the transaction.

ISOLATION_READ_UNCOMMITTED is the most efficient isolation level, but isolates the transaction the least, leaving the transaction open to dirty, nonrepeatable, and phantom reads. At the other extreme, ISOLATION_SERIALIZABLE prevents all forms of isolation problems but is the least efficient.

READ-ONLY

- If a transaction performs only read operations against the underlying data store, the data store may be able to apply certain optimizations that take advantage of the read-only nature of the transaction.
- By declaring a transaction as read-only, you give the underlying data store the opportunity to apply those optimizations as it sees fit.

TRANSACTION TIMEOUT

- *Suppose that our transaction becomes unexpectedly long-running. Because transactions may involve locks on the underlying data store, long-running transactions can tie up database resources unnecessarily.*
- *Instead of waiting it out, you can declare a transaction to automatically roll back after a certain number of seconds.*
- *Because the timeout clock begins ticking when a transaction starts, it only makes sense to declare a transaction timeout on methods with propagation behaviors that may start a new transaction (PROPAGATION_REQUIRED, PROPAGATION_REQUIRES_NEW, and PROPAGATION_NESTED).*

ROLLBACK RULES

- By default, transactions are rolled back only on runtime exceptions and not on checked exceptions. (This behaviour is consistent with rollback behaviour in EJBs.)
- But you can declare that a transaction be rolled back on specific checked exceptions as well as runtime exceptions.
- Likewise, you can declare that a transaction not roll back on specified exceptions, even if those exceptions are runtime exceptions. Now that you've had an overview of how transaction attributes shape the behaviour of a transaction, let's see how to use these attributes when declaring transactions in Spring.

Declaring transactions in XML

- Spring offers a tx configuration namespace that greatly simplifies declarative transactions in Spring.

Using the tx namespace involves adding it to your Spring configuration XML file:

```
<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-3.0.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

Defining annotation-driven transactions

- <tx:advice> element, the tx namespace provides the <tx:annotation- driven> element.
- Using <tx:annotation-driven> is often as simple as the following line of XML:

  - <tx:annotation-driven/> That's it!
  - <tx:annotation-driventransaction manager="txManager"/>

The `<tx:annotation-driven>` configuration element tells Spring to examine all beans in the application context and to look for beans that are annotated with `@Transactional`, either at the class level or at the method level. For every bean that is `@Transactional`, `<tx:annotation-driven>` will automatically advise it with transaction advice. The transaction attributes of the advice will be defined by parameters of the `@Transactional` annotation.

For example, the following shows `SpitterServiceImpl`, updated to include the `@Transactional` annotations.

Defining annotation-driven transactions

Listing 6.3 Annotating the spitter service to be transactional

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class SpitterServiceImpl implements SpitterService {
...
  @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
  public void addSpitter(Spitter spitter) {
...
  }
...
}
```

At the class level, `SpitterServiceImpl` has been annotated with an `@Transactional` annotation that says that all methods will support transaction and be read-only. At the method level, the `saveSpittle()` method has been annotated to indicate that this method requires a transactional context.