Lesson 4

Spring ORM
using Hibernate Framework

Spring ORM using Hibernate Framework

- In this model:
  - We use spring framework to create beans and inject session into your DAOs.
  - Inside DAOs methods we use Hibernate classes and interfaces.
- Factory class definition:

```
public class Factory {

    public SessionFactory getSessionFactory() {
        String configFile
                = "com/jediver/spring/model/dal/cfg/hibernate.cfg.xml";
        return new Configuration()
                .configure(configFile)
                .buildSessionFactory();
    }
}
```

Spring ORM using Hibernate Framework (Ex.)

- Beans definition:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="factory"
          class="com.jediver.spring.orm.hibernate.using.dal.cfg.Factory"/>
    <bean id="sessionFactory"
          factory-bean="factory"
          factory-method="getSessionFactory"/>
    <bean id="customerDAO"
          class="com.jediver.spring.orm.hibernate.using.dal.dao.impl.CustomerDAOImpl">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
</beans>
```

Spring ORM using Hibernate Framework (Ex.)

- DAOs Injection for session:

```
public class CustomerDAOImpl implements CustomerDAO {

    private Session session;

    public void setSessionFactory(SessionFactory sessionFactory) {
        session = sessionFactory.openSession();
    }
```

Spring ORM using Hibernate Framework (Ex.)

Querying (SELECT)

- The following function retrieve the number of rows in a customer table:

```
@Override
public long count() {
    String queryString = "select count(c) from Customer c";
    Query query = session.createQuery(queryString);
    return (long) query.uniqueResult();
}
```

Spring ORM using Hibernate Framework (Ex.)

Querying (SELECT)

- The following function retrieve the number of rows in a customer where age is greater than or equal the input parameter:

```
@Override
public long countByAgeGreaterThan(int age) {
    String queryString
            = "select count(c) from Customer c where c.age >= :age";
    Query query = session.createQuery(queryString);
    query.setParameter("age", age);
    return (long) query.uniqueResult();

}
```

Spring ORM using Hibernate Framework (Ex.)

Querying (SELECT)

- The following function retrieve Customer Object (Single Object) by customer id:

```
@Override
public Customer findOne(Integer customerId) {
    return session.load(Customer.class, customerId);
}
```

- The following function retrieve All Customer Objects (Multi-Object) :

```
@Override
public List<Customer> findAll() {
    Query q = session.createQuery("from Customer c");
    return q.list();
}
```

Spring ORM using Hibernate Framework (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public Customer save(Customer customer) {
    session.beginTransaction();
    session.save(customer);
    session.getTransaction().commit();
    return customer;
}
```

Spring ORM using Hibernate Framework (Ex.)

Updating (update)

- The following function update name and age of customer where customer id is passed from parameter:

```
@Override
public void update(Customer customer) {
    session.beginTransaction();
    session.update(customer);
    session.getTransaction().commit();
}
```

Spring ORM using Hibernate Framework (Ex.)

Updating (delete)

- The following function delete customer by customer id which is passed in parameter:

```
@Override
public void delete(Integer customerId) {
    Customer customer = findOne(customerId);
    session.beginTransaction();
    session.delete(customer);
    session.getTransaction().commit();
}
```

- The following function delete customer by customer which is passed in parameter:

```
@Override
public void delete(Customer customer) {
    session.beginTransaction();
    session.delete(customer);
    session.getTransaction().commit();
}
```

Spring ORM using Hibernate Framework (Ex.)

- Notes:
  - In this model you can Also create session factory from beans definition without factory.
  - Also you could inject directly the session into DAOs directly.
  - Spring Framework don’t provide anything to hibernate framework.

Lesson 5

Spring ORM
Hibernate Integration

Object Relational Mapping (ORM) (Ex.)

- Spring-orm under version of 4.2.0.RELEASE
  - Supports Hibernate 3+ and Hibernate 4+.
- Spring-orm from version of 4.2.0.RELEASE to version 4.3.0.RELEASE
  - Supports Hibernate 5+.
- Spring-orm from version of 4.3.0.RELEASE to version 5.0.0.RELEASE
  - Hibernate 3+ becomes deprecated.
- Spring-orm starting from version 5.0.0.RELEASE
  - Hibernate 3+ removed and Hibernate 4+ removed.
  - Only supports Hibernate ORM 5.0+

Object Relational Mapping (ORM) (Ex.)

- You can find the Hibernate 3+ support
  - under the org.springframework.orm.hibernate3 package
- You can find the Hibernate 4+ support
  - under the org.springframework.orm.hibernate4 package
- You can find the Hibernate 5+ support
  - under the org.springframework.orm.hibernate5 package

Object Relational Mapping (ORM) (Ex.)

- To Integrate between Spring and hibernate You Have Different Model:
  1. Using Hibernate Session
     - Query By using Session from Hibernate Session Factory that managed by Spring Container.
     - Update By using Session from Hibernate Session Factory that managed by Spring Container.
  2. Using HibernateTemplate Only
     - Query By using HibernateTemplate.
     - Update By executing updates with HibernateCallback with Hibernate Session.
  3. Using HibernateTemplate and TransactionTemplate
     - Query By using HibernateTemplate.
     - Update By using TransactionTemplate with TransactionCallback with HibernateTemplate.
  4. Using HibernateTemplate and @Transactional
     - Query By using HibernateTemplate.
     - Update By using @Transactional with HibernateTemplate.

Using Hibernate Session Model

- Starting from Hibernate 3+, Spring also includes support for annotation-based mapping.
- To Avoid tying application objects to hard-coded resource lookups.
    1. You can define resources as a JDBC DataSource
    2. Then let Spring Container create his managed Hibernate SessionFactory.
    - So after you create hibernate SessionFactory you can use it:
        - Either Different session management as hibernate supports
        - Or Manually managing session.

Using Hibernate Session Model (Ex.)

- The following definition of your datasource.properties to be used for property-placeholder:

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/customerdb
jdbc.user=root
jdbc.pass=root
```

- The following definition of your property-placeholder:

```
<context:property-placeholder
    location="classpath:com/jediver/spring/model/dal/cfg/datasource.properties" />
```

Using Hibernate Session Model (Ex.)

- The following definition of your JDBC DataSource:

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

- The following definition of your JDBC DataSource using JNDI Name:

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

Using Hibernate Session Model (Ex.)

- You can create Hibernate SessionFactory by spring from

  org.springframework.orm.hibernate5.LocalSessionFactoryBean:

- The following definition of your Hibernate SessionFactory, If you use xml configuration:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
        <list>
            <value>com/jediver/spring/model/dal/cfg/Customer.hbm.xml</value>
        </list>
    </property>
</bean>
```

Using Hibernate Session Model (Ex.)

- You can create Hibernate SessionFactory by spring from

  org.springframework.orm.hibernate5.LocalSessionFactoryBean:

- The following definition of your Hibernate SessionFactory, If you use Annotation configuration:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    </property>
    <property name="annotatedClasses">
        <list>
            <value>com.jediver.spring.model.dal.entity.Customer</value>
        </list>
    </property>
</bean>
```

Using Hibernate Session Model (Ex.)

- You can create Hibernate SessionFactory by spring from

  org.springframework.orm.hibernate5.LocalSessionFactoryBean:

- The following definition of your Hibernate SessionFactory, If you use Annotation

  configuration without define each mapped class:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="com.jediver.spring.model.dal.entity" />
</bean>
```

Using Hibernate Session Model (Ex.)

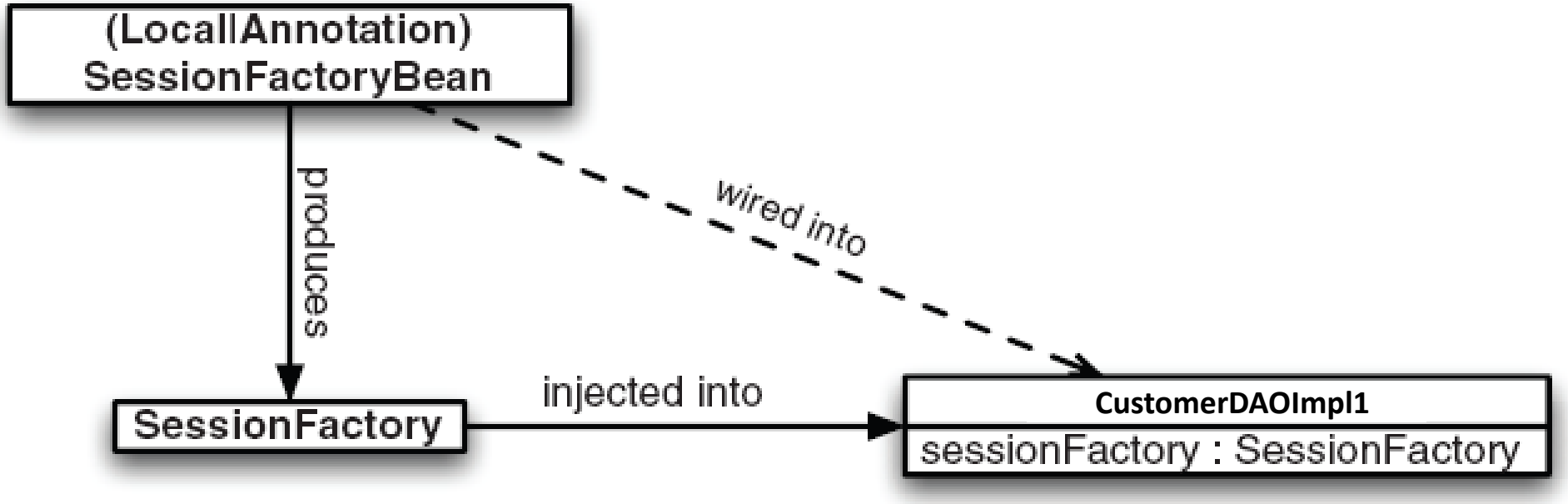


Figure 5.9 Taking advantage of Hibernate 3 contextual sessions, we can wire a `SessionFactory` (produced by a session factory bean) directly into a DAO, thus decoupling the DAO class from the Spring API.

Using Hibernate Session Model (Ex.)

- DAOs Injection for session:

```
public class CustomerDAOImpl implements CustomerDAO {

    private Session session;

    public void setSessionFactory(SessionFactory sessionFactory) {
        session = sessionFactory.openSession();
    }
```

- Beans definition:

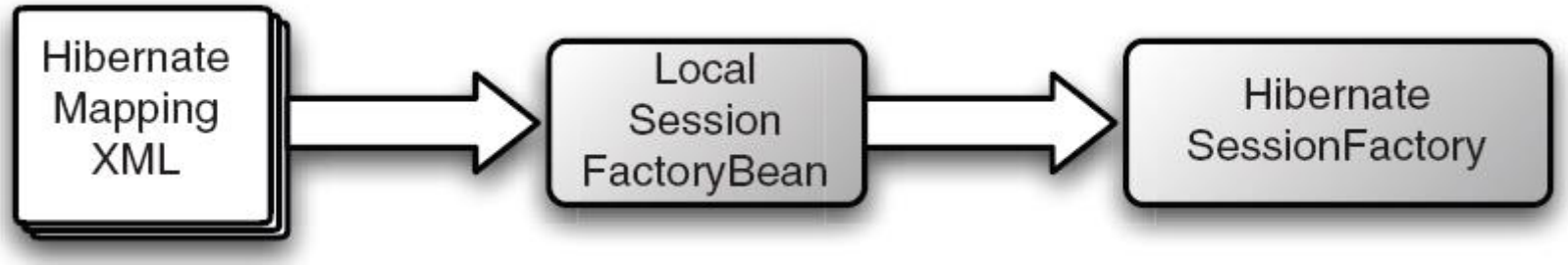
```
<bean id="customerDAO"
        class="com.jediver.spring.orm.hibernate.using.dal.dao.impl.CustomerDAOImpl">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Using Hibernate Session Model (Ex.)

- For Hibernate 3+ there is two classes
  - org.springframework.orm.hibernate3.LocalSessionFactoryBean.
  - This SesssionFactory created using new Configuration() from Hibernate.
  - org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean.
  - This SesssionFactory created using new AnnotationConfiguration() from Hibernate.

Using Hibernate Session Model (Ex.)

- org.springframework.orm.hibernate3.LocalSessionFactoryBean.
- If you are using Hibernate's classic XML mapping files, you'll want to use Spring's LocalSessionFactoryBean.
- LocalSessionFactoryBean is a Spring factory bean that produces a local Hibernate SessionFactory instance that draws its mapping metadata from one or more XML mapping files.
- Using new Configuration() from Hibernate.



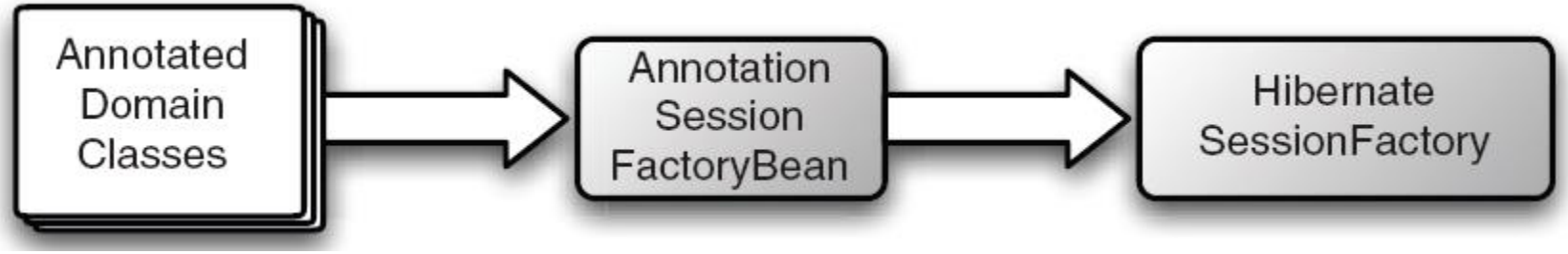
Using Hibernate Session Model (Ex.)

- For Hibernate 3+ there is two classes
  - org.springframework.orm.hibernate3.LocalSessionFactoryBean.
  - org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean.
- The following definition of your Hibernate SessionFactory, If you use xml configuration:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
        <list>
            <value>com/jediver/spring/model/dal/cfg/Customer.hbm.xml</value>
        </list>
    </property>
</bean>
```

Using Hibernate Session Model (Ex.)

- org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean.
- You may choose to use annotations to tag domain objects with persistence metadata.
- Hibernate 3 supports both JPA annotations and Hibernate-specific annotations*
- For annotation-based Hibernate, Spring's AnnotationSessionFactoryBean
  - Works much like LocalSessionFactoryBean.
  - Except that it creates a SessionFactory based on annotations in one or more domain classes
- Using new AnnotationConfiguration() from Hibernate.



Using Hibernate Session Model (Ex.)

- For Hibernate 3+ there is two classes
  - org.springframework.orm.hibernate3.LocalSessionFactoryBean.
  - org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean.
- The following definition of your Hibernate SessionFactory, If you use annotation configuration:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="annotatedClasses">
        <list>
            <value>com.jediver.spring.model.dal.entity.Customer</value>
        </list>
    </property>
</bean>
```

Using Hibernate Session Model (Ex.)

- **Note:**
  - You can perform CRUD operation normally using org.hibernate.Session Object from Hibernate.
  - We strongly recommend such an instance-based setup over the old-school static HibernateUtil class.
  - This Model has
    - Advantages:
    - Easy migration from use hibernate to this model of integration.
    - Disadvantages:
    - Tightly coupled with Hibernate.
    - You are still have to manage session either by hibernate or manually.
    - You are still have to manage transactions either by hibernate or JTA or manually.

Using HibernateTemplate Only

- Spring's HibernateTemplate provides an abstract layer over a Hibernate Session.
- HibernateTemplate's main responsibility is to
  - Simplify the work of opening and closing Hibernate Sessions
  - Convert Hibernate-specific exceptions to one of the Spring ORM exceptions

Using HibernateTemplate Only (Ex.)

- DAOs Injection for hibernateTemplate:
  - Either By use HibernateTemplate composition inside DAO.

```
public class CustomerDAOImpl1 implements CustomerDAO {

    private HibernateTemplate hibernateTemplate;

    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }
```

Using HibernateTemplate Only (Ex.)

- DAOs Injection for hibernateTemplate:
  - Or extends HibernateDaoSupport.
  - Spring offers HibernateDaoSupport, a convenience DAO support class, that enables you to wire a session factory bean directly into the DAO class.
  - Under the covers, HibernateDaoSupport creates a HibernateTemplate that the DAO can use.
  - The first step is to change CustomerDAOImpl1 to extend HibernateDaoSupport:

```
public class CustomerDAOImpl1
        extends HibernateDaoSupport implements CustomerDAO {
```

Using HibernateTemplate Only (Ex.)

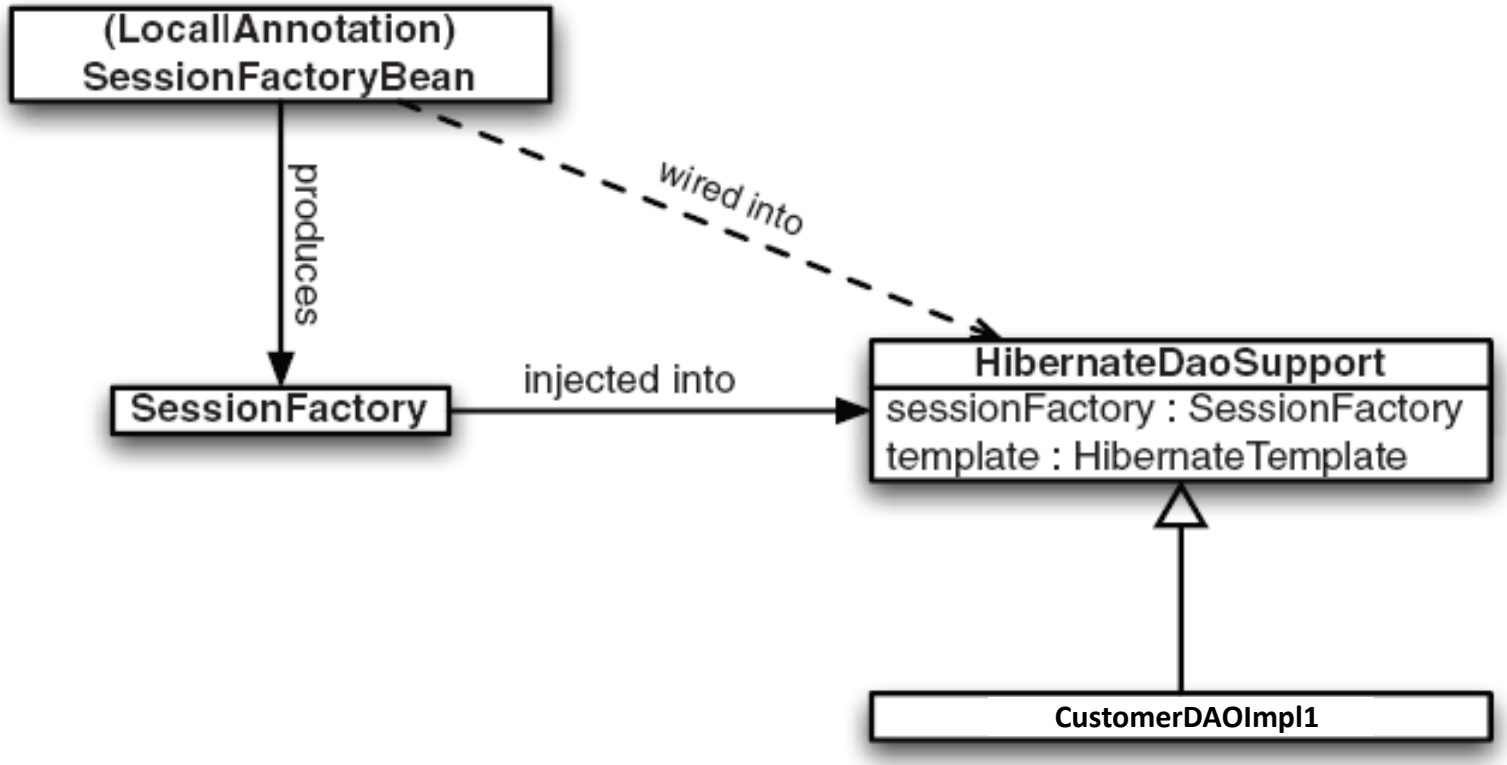


Figure 5.8 `HibernateDaoSupport` is a convenient superclass for a Hibernate-based DAO that provides a `HibernateTemplate` created from an injected `SessionFactory`.

Using HibernateTemplate Only (Ex.)

- HibernateTemplate Bean definition:

```
<bean id="hibernateTemplate"
      class="org.springframework.orm.hibernate5.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

- CustomerDAO Bean definition:

```
<bean id="customerDAO"
      class="com.jediver.spring.orm.hibernate.using.dal.dao.impl.CustomerDAOImpl1">
    <property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>
```

Using HibernateTemplate Only (Ex.)

Querying (SELECT)

- The following function retrieve the number of rows in a customer table:

```
@Override
public long count() {
    String queryString = "select count(c) from Customer c";
    List result = getHibernateTemplate().find(queryString);
    return (long) result.get(0);
}
```

Using HibernateTemplate Only (Ex.)

Querying (SELECT)

- The following function retrieve the number of rows in a customer where age is greater than or equal the input parameter:

```
@Override
public long countByAgeGreaterThan(int age) {
    String queryString
            = "select count(c) from Customer c where c.age >= :age";
    List result = getHibernateTemplate()
            .findByNamedParam(queryString, "age", age);
    return (long) result.get(0);
}
```

Using HibernateTemplate Only (Ex.)

Querying (SELECT)

- The following function retrieve Customer Object (Single Object) by customer id:

```
@Override
public Customer findOne(Integer customerId) {
    return getHibernateTemplate()
            .get(Customer.class, customerId);
}
```

- The following function retrieve All Customer Objects (Multi-Object) :

```
@Override
public List<Customer> findAll() {
    String queryString = "from Customer c";
    return (List<Customer>) getHibernateTemplate()
            .find(queryString);
}
```

Using HibernateTemplate Only (Ex.)

Updating (insert)

- The following function insert new record of type customer
- If auto-commit is enabled:

```
@Override
public Customer save(Customer customer) {
    getHibernateTemplate().saveOrUpdate(customer);
    return customer;
}
```

Using HibernateTemplate Only (Ex.)

Updating (insert)

- The following function insert new record of type customer If auto-commit is disabled:

```
@Override
public Customer save(Customer customer) {
    getHibernateTemplate().execute(new HibernateCallback<Object>() {
        @Override
        public Object doInHibernate(Session session) throws HibernateException {
            session.beginTransaction();
            session.save(customer);
            session.getTransaction().commit();
            return null;
        }
    });
    return customer;
}
```

Using HibernateTemplate Only (Ex.)

Updating (update)

- The following function update customer where customer id is passed from parameter:

```
@Override
public void update(Customer customer) {
    getHibernateTemplate().execute(new HibernateCallback<Object>() {
        @Override
        public Object doInHibernate(Session session) throws HibernateException {
            session.beginTransaction();
            session.update(customer);
            session.getTransaction().commit();
            return null;
        }
    });
}
```

Using HibernateTemplate Only (Ex.)

Updating (delete)

- The following function delete customer by customer which is passed in parameter:

```
@Override
public void delete(Customer customer) {
    getHibernateTemplate().execute(new HibernateCallback<Object>() {
        @Override
        public Object doInHibernate(Session session) throws HibernateException {
            session.beginTransaction();
            session.delete(customer);
            session.getTransaction().commit();
            return null;
        }
    });
}
```

- The following function delete customer customer id which is passed in parameter:

```
@Override
public void delete(Integer customerId) {
    Customer customer = findOne(customerId);
    delete(customer);
}
```

Using HibernateTemplate Only (Ex.)

- **Note:**
  - Notice that CustomerDAOImpl1 extends a Spring-specific class.
  - This may be a problem for you, since intrusion of Spring into their application code.
  - One of the responsibilities of HibernateTemplate is to manage Hibernate Sessions.
  - This involves opening and closing sessions as well as ensuring one session per transaction.

Using HibernateTemplate and TransactionTemplate

- Spring's HibernateTemplate provides an abstract layer over a Hibernate Session.
- HibernateTemplate's main responsibility is to
  - Simplify the work of opening and closing Hibernate Sessions
  - Convert Hibernate-specific exceptions to one of the Spring ORM exceptions
- The TransactionTemplate adopts the same approach as other Spring templates, such as the JdbcTemplate.
- It uses a callback approach and results in code that is intention driven, in that your code focuses solely on what you want to do.

Using HibernateTemplate and TransactionTemplate (Ex.)

- DAOs Injection for hibernateTemplate and transactionTemplate:

```
public class CustomerDAOImpl2 implements CustomerDAO {

    private HibernateTemplate hibernateTemplate;
    private TransactionTemplate transactionTemplate;

    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }

    public void setTransactionTemplate(TransactionTemplate transactionTemplate) {
        this.transactionTemplate = transactionTemplate;
    }
```

Using HibernateTemplate and TransactionTemplate (Ex.)

- HibernateTemplate Bean definition as usual:

```
<bean id="hibernateTemplate"
      class="org.springframework.orm.hibernate5.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

- TransactionManager and TransactionTemplate Bean definition as usual:

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
<bean id="transactionTemplate"
      class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="transactionManager"/>
</bean>
```

Using HibernateTemplate and TransactionTemplate (Ex.)

- CustomerDAO Bean definition as usual:

```
<bean id="customerDAO"
      class="com.jediver.spring.orm.hibernate.using.dal.dao.impl.CustomerDAOImpl2">
    <property name="hibernateTemplate" ref="hibernateTemplate"/>
    <property name="transactionTemplate" ref="transactionTemplate"/>
</bean>
```

Using HibernateTemplate and TransactionTemplate (Ex.)

Querying (SELECT)

- For all selecting queries as normal using normal hibernateTemplate
- The following function retrieve the number of rows in a customer table:

```
@Override
public long count() {
    String queryString = "select count(c) from Customer c";
    List result = getHibernateTemplate().find(queryString);
    return (long) result.get(0);
}
```

Using HibernateTemplate and TransactionTemplate (Ex.)

Updating (insert)

- The following function insert new record of type customer
- If auto-commit is enabled:

```
@Override
public Customer save(Customer customer) {
    getHibernateTemplate().saveOrUpdate(customer);
    return customer;
}
```

Using HibernateTemplate and TransactionTemplate (Ex.)

Updating (insert)

- The following function insert new record of type customer If auto-commit is disabled:

```
@Override
public Customer save(Customer customer) {
    transactionTemplate.execute(new TransactionCallback<Object>() {
        @Override
        public Object doInTransaction(TransactionStatus ts) {
            hibernateTemplate.save(customer);
            return ts;
        }
    });
    return customer;
}
```

Using HibernateTemplate and TransactionTemplate (Ex.)

Updating (update)

- The following function update customer where customer id is passed from parameter:

```
@Override
public void update(Customer customer) {
    transactionTemplate.execute(new TransactionCallback<Object>() {
        @Override
        public Object doInTransaction(TransactionStatus ts) {
            hibernateTemplate.update(customer);
            return ts;
        }
    });
}
```

Using HibernateTemplate and TransactionTemplate (Ex.)

Updating (delete)

- The following function delete customer by customer which is passed in parameter:

```
@Override
public void delete(Customer customer) {
    transactionTemplate.execute(new TransactionCallback<Object>() {
        @Override
        public Object doInTransaction(TransactionStatus ts) {
            hibernateTemplate.delete(customer);
            return ts;
        }
    });
}
```

- The following function delete customer customer id which is passed in parameter:

```
@Override
public void delete(Integer customerId) {
    Customer customer = findOne(customerId);
    delete(customer);
}
```

Using HibernateTemplate and TransactionTemplate (Ex.)

- **Note:**
  - we use TransactionTemplate which is a Spring-specific class.
  - This may be a problem for you, since intrusion of Spring into their application code.
  - One of the responsibilities of HibernateTemplate is to manage Hibernate Sessions.
  - This involves opening and closing sessions as well as ensuring one session per transaction.
  - One of the responsibilities of TransactionTemplate is to manage Transaction (Commits and Rollback) instead of hibernate or user.

Using HibernateTemplate and @Transactional

- Spring's HibernateTemplate provides an abstract layer over a Hibernate Session.
- HibernateTemplate's main responsibility is to
  - Simplify the work of opening and closing Hibernate Sessions
  - Convert Hibernate-specific exceptions to one of the Spring ORM exceptions
- The @Transactional annotation
  - On a class specifies the default transaction semantics for the execution of any public method in the class.
  - On a method within the class overrides the default transaction semantics given by the class annotation (if present).
    - You can annotate any method, regardless of visibility.

Using HibernateTemplate and @Transactional (Ex.)

- DAOs Injection for hibernateTemplate:

```
public class CustomerDAOImpl3 implements CustomerDAO {

    private HibernateTemplate hibernateTemplate;

    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }
```

Using HibernateTemplate and @Transactional (Ex.)

- First of all you must make Spring Context understand @Transactional annotation so you must import it by transaction namespace :

```
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"
```

- <annotation-driven> Tag.

```
<tx:annotation-driven />
```

Using HibernateTemplate and @Transactional (Ex.)

- HibernateTemplate Bean definition as usual:

```
<bean id="hibernateTemplate"
      class="org.springframework.orm.hibernate5.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

- We need only to define bean definition for TransactionManager:

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Using HibernateTemplate and @Transaction (Ex.)

- CustomerDAO Bean definition as usual:

```
<bean id="customerDAO"
      class="com.jediver.spring.orm.hibernate.using.dal.dao.impl.CustomerDAOImpl3">
    <property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>
```

Using HibernateTemplate and @Transactional (Ex.)

Querying (SELECT)

- For all selecting queries as normal using normal hibernateTemplate
- The following function retrieve the number of rows in a customer table:

```
@Override
public long count() {
    String queryString = "select count(c) from Customer c";
    List result = getHibernateTemplate().find(queryString);
    return (long) result.get(0);
}
```

Using HibernateTemplate and @Transactional (Ex.)

Updating (insert)

- The following function insert new record of type customer
- If auto-commit is enabled or disabled :

```
@Transactional
@Override
public Customer save(Customer customer) {
    hibernateTemplate.save(customer);
    return customer;
}
```

Using HibernateTemplate and @Transactional (Ex.)

Updating (update)

- The following function update customer where customer id is passed from parameter:

```
@Transactional
@Override
public void update(Customer customer) {
    hibernateTemplate.update(customer);
}
```

Using HibernateTemplate and @Transactional (Ex.)

Updating (delete)

- The following function delete customer by customer which is passed in parameter:

```
@Transactional
@Override
public void delete(Customer customer) {
    hibernateTemplate.delete(customer);
}
```

- The following function delete customer by customer id which is passed in parameter:

```
@Transactional
@Override
public void delete(Integer customerId) {
    Customer customer = findOne(customerId);
    delete(customer);
}
```

Using HibernateTemplate and @Transactional (Ex.)

- **Note:**
  - One of the responsibilities of HibernateTemplate is to manage Hibernate Sessions.
  - This involves opening and closing sessions as well as ensuring one session per transaction.
  - One of the responsibilities of @Transactional is to manage Transaction (Commits and Rollback) instead of hibernate or user.

It’s the most recommended model to declare transaction so you don’t couple with any implementations neither hibernate, nor JPA, nor spring orm.

How does @Transactional work?

Three separate components are needed:

- The Transactional Aspect
- The Transaction Manager
- Transactional Proxy

How does @Transactional work?

The Transactional Aspect

The Transactional Aspect is an 'around' aspect that gets called both before and after the annotated business method.

At the 'before' moment, the aspect delegates the decision whether to start new transaction or not to the **Transaction Manager**.

At the 'after' moment, the aspect decide if the transaction should be committed, rolled back or left running.

How does @Transactional work?

The Transaction Manager

The transaction manager take the decision for:

- Create new Session or not?
- Start new database transaction or not?

This needs to be decided at the moment the Transactional Aspect 'before' logic is called.

How does @Transactional work?

The Transaction Manager

The transaction manager will decide to start new Transaction based on:

- If there is an ongoing transaction or not
- The value of the propagation attribute of the transactional.

If the transaction manager decides to create a new transaction, then it will:

1. create a new Session
2. bind the Session to the current thread
3. grab a connection from the DB connection pool
4. bind the connection to the current thread

How does @Transactional work?

The Transaction Manager

The Session and the connection are both bound to the current thread using **ThreadLocal** variables.

They are stored in the thread while the transaction is running, and it's up to the Transaction Manager to clean them up when no longer needed.

How does @Transactional work?

The Transactional proxy

The Transactional proxy will be used when the business method calls for example *persist*(), this call is not invoking it directly, Instead the business method calls the proxy, which retrieves the current session from the thread, where the Transaction Manager put it.

@Transactional Propagation Levels

@Transactional(propagation = Propagation.MANDATORY)

- **Required (default)**: My method needs a transaction, either open one for me or use an existing one.
- **Supports**: I don't really care if a transaction is open or not, I can work either way.
- **Mandatory**: I'm not going to open up a transaction myself, but I'm going to cry if no one else opened one up.
- **Require_new:** I want my completely own transaction.
- **Not_Supported:** I really don't like transactions, I will even try and suspend a current, running transaction.
- **Never:** I'm going to cry if someone else started up a transaction .
- **Nested:** Execute within a nested transaction (SavePoints) if a current transaction exists, behave like REQUIRED otherwise.

@Transactional rollback (and default rollback policies)

- The transaction will roll back on RuntimeException and Error but not on checked exceptions.
- The rollback rules Can be customized based on patterns.
- A pattern can be a fully qualified class name or a substring of a fully qualified class name for an exception type (which must be a subclass of Throwable).
- The pattern will be presented to :

rollbackFor()/noRollbackFor() or rollbackForClassName()/noRollbackForClassName() attributes in the @Transactional annotation ;which allow patterns to be specified as Class references or strings

- A thrown exception is considered to be a match for a given rollback rule if the name of thrown exception contains the exception pattern configured for the rollback rule.

```
@Transactional(rollbackFor = example.CustomException.class)
```

```
@Transactional(rollbackForClassName = "example.CustomException")
```