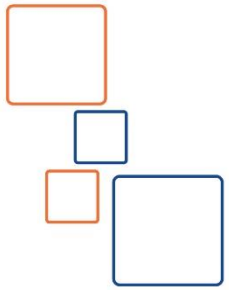
Spring Framework

THE RIGHT TECHNOLOGY STACK FOR THE JOB AT HAND



Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Java™ Education and Technology Services

Course Outline

- Lesson 1: Using JDBC into Spring Framework
- Lesson 2: Spring JDBC Module
- Lesson 3: Spring ORM
- Lesson 4: Spring ORM using Hibernate Framework
- Lesson 5: Spring ORM Hibernate Integration
- Lesson 6: Spring ORM using JPA Framework
- Lesson 7: Spring ORM JPA Integration
- *** **References & Recommended Reading**

Lesson 1

Using JDBC into Spring Framework

Using JDBC into Spring Framework

- Spring in this model only
    - Declare the datasource
    - Manage the lifecycle of this datasource.
- Using JDBC Connection native connection from predefined datasource
- You deal with datasource as Java normal classes
- You are managing every thing connection, transaction, exception handling, business logic

Using JDBC into Spring Framework (Ex.)

- We can also use JDBC API internal inside Spring application by declaring DataSource as it will be like driver manager

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/customerdb" />
    <property name="username" value="root" />
    <property name="password" value="root" />
</bean>
```

Using JDBC into Spring Framework (Ex.)

- You will inject your datasource into your DAO classes or connection factory classes.
- Your DAO Class as Follows:

```
public class JdbcCustomerDAO implements CustomerDAO {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

- Your Bean definition as Follows:

```
<bean id="customerDAO"
      class="com.jediver.jdbc.dal.dao.impl.JdbcCustomerDAO">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Using JDBC into Spring Framework (Ex.)

- You have to manage manually:
  - Define connection parameters.
  - Open the connection.
  - Specify the SQL statement.
  - Declare parameters and provide parameter values
  - Prepare and execute the statement.
  - Set up the loop to iterate through the results (if any).
  - Do the work for each iteration.
  - Process any exception.
  - Handle transactions.
  - Close the connection, the statement, and the resultset.

Using JDBC into Spring Framework (Ex.)

- You can obtain new connection by calling:
  - Connection connection = dataSource.getConnection();

Using JDBC into Spring Framework (Ex.)

```
@Override
public void insert(Customer customer) {
    String sql = "INSERT INTO CUSTOMER (id, name, age) VALUES (?, ?, ?)";
    Connection conn = null;
    try {
        conn = dataSource.getConnection();
        PreparedStatement preparedStatement = conn.prepareStatement(sql);
        preparedStatement.setInt(1, customer.getId());
        preparedStatement.setString(2, customer.getName());
        preparedStatement.setInt(3, customer.getAge());
        preparedStatement.executeUpdate();
        preparedStatement.close();
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
```

Using JDBC into Spring Framework (Ex.)

Java™ Education and Technology Services

```
        finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    System.err.println(e.getMessage());
                }
            }
        }
    }
```

Using JDBC into Spring Framework (Ex.)

```
@Override
public Customer findByCustomerId(int customerId) {
    String sql = "SELECT * FROM CUSTOMER WHERE id = ?";
    Connection conn = null;
    Customer customer = null;
    try {
        conn = dataSource.getConnection();
        PreparedStatement preparedStatement = conn.prepareStatement(sql);
        preparedStatement.setInt(1, customerId);
        ResultSet resultSet = preparedStatement.executeQuery();
        if (resultSet.next()) {
            customer = new Customer();
            customer.setId(resultSet.getInt("id"));
            customer.setName(resultSet.getString("name"));
            customer.setAge(resultSet.getInt("age"));
        }
        resultSet.close();
        preparedStatement.close();
    }
```

Using JDBC into Spring Framework (Ex.)

```
        catch (SQLException e) {
            System.err.println(e.getMessage());
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    System.err.println(e.getMessage());
                }
            }
        }
        return customer;
    }
```

Lesson 2

Spring JDBC Module

Spring JDBC Framework

- Spring Framework JDBC provide abstraction of JDBC Layer.
- The Spring Framework's JDBC framework consists of four different packages:

1. org.springframework.jdbc.core

- The org.springframework.jdbc.core package contains
  - The JdbcTemplate class and its various callback interfaces, plus a variety of related classes.
- A subpackage named org.springframework.jdbc.core.simple contains
  - The SimpleJdbcInsert and SimpleJdbcCall classes.
- Another subpackage named org.springframework.jdbc.core.namedparam contains
  - the NamedParameterJdbcTemplate class and the related support classes.

Spring JDBC Framework

- The Spring Framework's JDBC framework consists of four different packages:

2. org.springframework.jdbc.datasource

- The org.springframework.jdbc.datasource package contains
  - A utility class for easy DataSource access and various simple DataSource implementations that you can use for testing and running unmodified JDBC code outside of a Java EE container.
- A subpackage named org.springfamework.jdbc.datasource.embedded
  - Provides support for creating embedded databases by using Java database engines, such as HSQL, H2, and Derby.

Spring JDBC Framework

- The Spring Framework's JDBC framework consists of four different packages:

3. org.springframework.jdbc.object

- The org.springframework.jdbc.object package contains
  - Classes that represent RDBMS queries, updates, and stored procedures as thread-safe, reusable objects.
- This approach is modeled by JDO (Java Data Objects), although objects returned by queries are naturally disconnected from the database.
- This higher-level of JDBC abstraction depends on the lower-level abstraction in the org.springframework.jdbc.core package

Spring JDBC Framework

- The Spring Framework's JDBC framework consists of four different packages:

4. org.springframework.jdbc.support

- The org.springframework.jdbc.support package
  - Provides SQLException translation functionality and some utility classes.
- Exceptions thrown during JDBC processing are translated to exceptions defined in the org.springframework.dao package.
- This means that code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling.
- All translated exceptions are unchecked, which gives you the option of catching the exceptions from which you can recover while letting other exceptions be propagated to the caller.

Spring JDBC Framework (Ex.)

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, the statement, and the resultset.	X	

Using JdbcTemplate

- JdbcTemplate is the central class in the JDBC core package.
- It handles the creation and release of resources, which helps you avoid common errors, such as forgetting to close the connection.
- It performs the basic tasks of the core JDBC workflow (such as statement creation and execution), leaving application code to provide SQL and extract results.
- The JdbcTemplate class:
  - Runs SQL queries
  - Updates statements and stored procedure calls
  - Performs iteration over ResultSet instances and extraction of returned parameter values.
  - Catches JDBC exceptions and translates them to the generic.

Using JdbcTemplate (Ex.)

- Instances of the JdbcTemplate class are thread safe.
- Single instance of a JdbcTemplate is sufficient.
- Safely inject this shared reference into multiple DAOs

Using JdbcTemplate (Ex.)

- Your DAO Class as Follows:

```
public class JdbcCustomerDAO implements CustomerDAO {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
```

- Your Bean definition as Follows:

```
<bean id="customerDAO"
      class="com.jediver.jdbc.dal.dao.impl.JdbcCustomerDAO">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

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

```
@Override
public int count() {
    String SQL = "select count(*) from Customer";
    int rowCount = jdbcTemplate.queryForObject(SQL, Integer.class);
    return rowCount;
}
```

Using JdbcTemplate (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 int countByAgeGreaterThan(int age) {
    String SQL = "select count(*) from Customer where age >= ?";
    int rowCount = jdbcTemplate.queryForObject(SQL,
            new Object[]{age}, Integer.class);
    return rowCount;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

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

```
@Override
public Customer findByCustomerId(int customerId) {
    String sql = "SELECT * FROM CUSTOMER WHERE id = ?";
    Object[] args = new Object[]{customerId};
    SqlRowSet rowset = jdbcTemplate.queryForRowSet(sql, args);
    Customer customer = null;
    if (rowset.next()) {
        customer = new Customer();
        customer.setId(rowset.getInt("id"));
        customer.setName(rowset.getString("name"));
        customer.setAge(rowset.getInt("age"));
    }
    return customer;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- We find some of challenges to bind the rowset into the entities object, So spring provide some classes to auto-bind between the rowset and the entities.
- Spring Provide auto-bind by org.springframework.jdbc.core.RowMapper interface.
- You can do this by
  - Either by making your class that implement RowMapper interface.
  - Or by using built-in classes that implement RowMapper interface.

Using JdbcTemplate (Ex.)

Querying (SELECT)

- Using BeanPropertyRowMapper that use bean property to identify the relation between the bean property and the column in table.
- The following function retrieve Customer Object (Single Object) by customer id:

```
@Override
public Customer findByCustomerId(int customerId) {
    String sql = "SELECT * FROM CUSTOMER WHERE id = ?";
    Object[] args = new Object[]{customerId};
    Customer customer = jdbcTemplate.queryForObject(
            sql, args, new BeanPropertyRowMapper<>(Customer.class));
    return customer;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- Using Custom Row Mapper that you are define the relation between the bean property and the column in table.

```
public class CustomerRowMapper implements RowMapper {

    @Override
    public Object mapRow(ResultSet rs, int rowNum)
            throws SQLException {
        Customer customer = new Customer();
        customer.setId(rs.getInt("id"));
        customer.setName(rs.getString("name"));
        customer.setAge(rs.getInt("age"));
        return customer;
    }
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

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

```
@Override
public Customer findByCustomerId(int customerId) {
    String sql = "SELECT * FROM CUSTOMER WHERE id = ?";
    Customer customer = (Customer) jdbcTemplate.queryForObject(
            sql, new Object[]{customerId}, new CustomerRowMapper());
    return customer;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

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

```
@Override
public List<Customer> findAll() {
    String sql = "SELECT * FROM CUSTOMER";
    List<Customer> customers = new ArrayList<>();
    List<Map<String, Object>> rows = jdbcTemplate.queryForList(sql);
    for (Map row : rows) {
        Customer customer = new Customer();
        customer.setId((row.get("id")));
        customer.setName((String) row.get("name"));
        customer.setAge((int) row.get("age"));
        customers.add(customer);
    }
    return customers;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- Using Custom Result Set Extractor that you are define the relation between the bean property and the column in table

```
public class CustomerResultSetExtractor
        implements ResultSetExtractor<List<Customer>> {

    @Override
    public List<Customer> extractData(ResultSet resultSet)
            throws SQLException, DataAccessException {
        List<Customer> customers = new ArrayList<>();
        while (resultSet.next()) {
            Customer customer = new Customer();
            customer.setId(resultSet.getInt("id"));
            customer.setName(resultSet.getString("name"));
            customer.setAge(resultSet.getInt("age"));
            customers.add(customer);
        }
        return customers;
    }
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

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

```
@Override
public List<Customer> findAll() {
    String sql = "SELECT * FROM CUSTOMER";
    List<Customer> customers = jdbcTemplate
            .query(sql, new CustomerResultSetExtractor());
    return customers;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

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

```
@Override
public List<Customer> findAll() {
    String sql = "SELECT * FROM CUSTOMER";
    List<Customer> customers = jdbcTemplate.query(
            sql, new BeanPropertyRowMapper<>(Customer.class));
    return customers;
}
```

Using JdbcTemplate (Ex.)

Querying (SELECT)

- The following function retrieve All Customer Objects (Multi-Object) using Custom Row Mapper:

```
@Override
public List<Customer> findAll() {
    String sql = "SELECT * FROM CUSTOMER";
    List<Customer> customer = jdbcTemplate.query(
            sql, new CustomerRowMapper());
    return customer;
}
```

Using JdbcTemplate (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    String sql = "INSERT INTO CUSTOMER (id, name, age) VALUES (?, ?, ?)";
    Object[] args = new Object[]{customer.getId(),
        customer.getName(), customer.getAge()};
    jdbcTemplate.update(sql, args);
}
```

Using JdbcTemplate (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) {
    String sql = "update CUSTOMER set name = ?,age=? where id = ?";
    Object[] args = new Object[]{customer.getName(),
        customer.getAge(), customer.getId()};
    jdbcTemplate.update(sql, args);
}
```

Using JdbcTemplate (Ex.)

Updating (delete)

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

```
@Override
public void delete(int customerId) {
    String sql = "delete from CUSTOMER where id = ?";
    Object[] args = new Object[]{customerId};
    jdbcTemplate.update(sql, args);
}
```

Using JdbcTemplate (Ex.)

- You can use the execute method to run any arbitrary SQL.
- Consequently, the method is often used for DDL statements.
- It is heavily overloaded with variants that take callback interfaces, binding variable arrays, and so on.
- The following example creates a table:

```
@Override
public void createXTable() {
    String sql = "create table xxx (id integer, name varchar(100))";
    jdbcTemplate.execute(sql);
}
```

Using NamedParameterJdbcTemplate

- Spring Also provide another class instead of JdbcTemplate called NamedParameterJdbcTemplate.
- Which adds only the usage of named parameter instead of indexing parameter.
- You can declare it by same way as JdbcTemplate.

```
public class JdbcCustomerDAO1 implements CustomerDAO {

    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        namedParameterJdbcTemplate
                = new NamedParameterJdbcTemplate(dataSource);
    }
```

Using NamedParameterJdbcTemplate (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    String sql = "INSERT INTO CUSTOMER (id, name, age) "
            + "VALUES (:id, :name, :age)";
    Map<String, Object> args = new HashMap<>();
    args.put("id", customer.getId());
    args.put("name", customer.getName());
    args.put("age", customer.getAge());
    namedParameterJdbcTemplate.update(sql, args);
}
```

Using NamedParameterJdbcTemplate (Ex.)

Querying (SELECT)

- If your query didn't take any arguments just pass an empty map
- The following function retrieve the number of rows in a customer table:

```
@Override
public int count() {
    String SQL = "select count(*) from Customer";
    Map<String, Object> args = new HashMap<>();
    int rowCount = namedParameterJdbcTemplate
            .queryForObject(SQL, args, Integer.class);
    return rowCount;
}
```

- Also You could use all query classes as you used before like (RowMapper, BeanPropertyRowMapper, ResultSetExtractor, etc)

Using SimpleJdbcTemplate

- For earlier versions in Spring till version 4
- Spring Provide a class called SimpleJdbcTemplate instead of JdbcTemplate
- This class inherit from JdbcTemplate and provide the dynamic parameter.

```
int update(String sql,Object... parameters)
```

```
String query="update employee set name=? where id=?";
return template.update(query,e.getName(),e.getId());
```

Using JdbcDaoSupport

- Spring Also provide another class Called JdbcDaoSupport.
- You will define your class that inherit from JdbcDaoSupport and use the function getJdbcTemplate() from super class to get the instance of JdbcTemplate.
- Your DAO Class as Follows:

```
public class JdbcCustomerDAO2 extends JdbcDaoSupport
        implements CustomerDAO {
```

- Your Bean definition as Follows:

```
<bean id="customerDAO"
      class="com.jediver.jdbc.dal.dao.impl.JdbcCustomerDAO">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Using JdbcDaoSupport (Ex.)

Querying (SELECT)

- All Configuration are identical as JdbcTemplate, the only difference is obtaining an object from JdbcTemplate from the super class.
- The following function retrieve the number of rows in a customer table:

```
@Override
public int count() {
    String SQL = "select count(*) from Customer";
    int rowCount = getJdbcTemplate().queryForObject(SQL, Integer.class);
    return rowCount;
}
```

- Also You could use all query classes as you used before like (RowMapper, BeanPropertyRowMapper, ResultSetExtractor, etc)

Using JdbcDaoSupport (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    String sql = "INSERT INTO CUSTOMER (id, name, age) "
            + "VALUES (?, ?, ?)";
    Object[] args = new Object[]{customer.getId(),
        customer.getName(), customer.getAge()};
    getJdbcTemplate().update(sql, args);
}
```

Using JdbcDaoSupport (Ex.)

Why ???

- There is another benefit to deal with JdbcDaoSupport instead of JdbcTemplate because:
  - You can get the current connection with JDBC Connection by using getConnection();
  - Also You can release connection by calling releaseConnection(); to force release for the current connection.

Using SimpleJdbcDaoSupport

- For earlier versions in Spring till version 4
- Spring Provide a class called SimpleJdbcDaoSupport instead of JdbcDaoSupport
- This class inherit from JdbcDaoSupport and provide an instance from SimpleJdbcTemplate.
- Your DAO Class as Follows:

```
public class JdbcCustomerDAO extends SimpleJdbcDaoSupport
        implements CustomerDAO {
```

- Your Bean definition as Follows:

```
<bean id="customerDAO"
      class="com.jediver.jdbc.dal.dao.impl.JdbcCustomerDAO">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Using SimpleJdbcDaoSupport (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insertNamedParameter(Customer customer) {
    String sql = "INSERT INTO CUSTOMER "
            + "(id, name, age) VALUES (:id, :name, :age)";
    Map<String, Object> parameters = new HashMap<>();
    parameters.put("id", customer.getCustId());
    parameters.put("name", customer.getName());
    parameters.put("age", customer.getAge());
    getSimpleJdbcTemplate().update(sql, parameters);
}
```

Simplifying JDBC Operations

- The SimpleJdbcInsert and SimpleJdbcCall classes provide a simplified configuration by taking advantage of database metadata that can be retrieved through the JDBC driver.
- This means that you have less to configure up front, although you can override or turn off the metadata processing if you prefer to provide all the details in your code.
- Inserting Data by Using SimpleJdbcInsert.
- Calling a Stored Procedure with SimpleJdbcCall.

Inserting Data by Using SimpleJdbcInsert

- We start by looking at the SimpleJdbcInsert class with the minimal amount of configuration options.
- You should instantiate the SimpleJdbcInsert in the data access layer's initialization method.
- You do not need to subclass the SimpleJdbcInsert class. Instead, you can create a new instance and set the table name by using the withTableName method.
- Configuration methods for this class follow the fluid style that returns the instance of the SimpleJdbcInsert, which lets you chain all configuration methods.

Inserting Data by Using SimpleJdbcInsert (Ex.)

- Your DAO Class as Follows:

```
public class JdbcCustomerDAO3 implements CustomerDAO {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertCustomer;

    public void setDataSource(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
        insertCustomer = new SimpleJdbcInsert(dataSource)
                .withTableName("customer")
                .usingColumns("id", "name", "age");
    }
```

- Your Bean definition as Follows:

```
<bean id="customerDAO"
      class="com.jediver.jdbc.dal.dao.impl.JdbcCustomerDAO">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Inserting Data by Using SimpleJdbcInsert (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    Map<String, Object> parameters = new HashMap<>(3);
    parameters.put("id", customer.getId());
    parameters.put("name", customer.getName());
    parameters.put("age", customer.getAge());
    insertCustomer.execute(parameters);
}
```

Inserting Data by Using SimpleJdbcInsert (Ex.)

- If you have auto-generated keys in database so you could choose to use usingGeneratedKeyColumns() method.
- The following example uses the same insert as the preceding example, but, instead of passing in the id, it retrieves the auto-generated key and sets it on the new Customer object.
- When it creates the SimpleJdbcInsert, in addition to specifying the table name, it specifies the name of the generated key column with the usingGeneratedKeyColumns method.

Inserting Data by Using SimpleJdbcInsert (Ex.)

- Your DAO Class as Follows:

```
public class JdbcCustomerDAO3 implements CustomerDAO {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertCustomer;

    public void setDataSource(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
        insertCustomer = new SimpleJdbcInsert(dataSource)
                .withTableName("customer")
                .usingColumns( "name", "age")
                .usingGeneratedKeyColumns("id");
    }
```

Inserting Data by Using SimpleJdbcInsert (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    Map<String, Object> parameters = new HashMap<>(2);
    parameters.put("name", customer.getName());
    parameters.put("age", customer.getAge());
    insertCustomer.execute(parameters);
}
```

Inserting Data by Using SimpleJdbcInsert (Ex.)

- Using SqlParameterSource to Provide Parameter Values:
- Using a Map to provide parameter values works fine, but it is not the most convenient class to use.
- Spring provides a couple of implementations of the SqlParameterSource interface that you can use instead.
  - The first one is BeanPropertySqlParameterSource, which is a very convenient class if you have a JavaBean-compliant class that contains your values.
  - It uses the corresponding getter method to extract the parameter values.

Inserting Data by Using SimpleJdbcInsert (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    SqlParameterSource parameters
            = new BeanPropertySqlParameterSource(customer);
    insertCustomer.execute(parameters);
}
```

Inserting Data by Using SimpleJdbcInsert (Ex.)

Updating (insert)

- The second one is the MapSqlParameterSource that resembles a Map but provides a more convenient addValue method that can be chained..
- The following function insert new record of type customer:

```
@Override
public void insert(Customer customer) {
    SqlParameterSource parameters = new MapSqlParameterSource()
            .addValue("name", customer.getName())
            .addValue("age", customer.getAge());
    insertCustomer.execute(parameters);
}
```

Lesson 3

Spring ORM

Object Relational Mapping (ORM)

- The Spring Framework supports Integration with
  - Native Hibernate.
  - Integration with the Java Persistence API (JPA).
  - iBATIS
  - Apache OJB,
  - Java Data Objects (JDO),
  - Oracle's TopLink,
  - **With different way:**
    - Data access object (DAO) implementations.
    - Transaction strategies.

Object Relational Mapping (ORM) (Ex.)

- 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 or JPA APIs.
- 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.

Benefits of using Spring to create DAOs

- The benefits of using the Spring Framework to create your ORM DAOs include:
- Easier testing.
  - Spring's IoC 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.

Benefits of using Spring to create DAOs (Ex.)

- 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 lets you 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.

Benefits of using Spring to create DAOs (Ex.)

- General resource management.
  - Spring application contexts can handle the location and configuration of Hibernate SessionFactory instances, JPA EntityManagerFactory instances, JDBC DataSource instances, and other related resources.
  - 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.
  - Spring solves many chronic problems of typical Hibernate usage, for any local or JTA transaction.

Benefits of using Spring to create DAOs (Ex.)

- 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.
  - 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) but that still needs to share common transactions with ORM operations.

Object Relational Mapping (ORM) (Ex.)

- Spring's support for ORM frameworks provides integration points to the frameworks as well as some additional services:
  - Integrated support for Spring declarative transactions
  - Transparent exception handling
  - Thread-safe.
  - Lightweight template classes
  - DAO support classes
  - Resource management

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.)

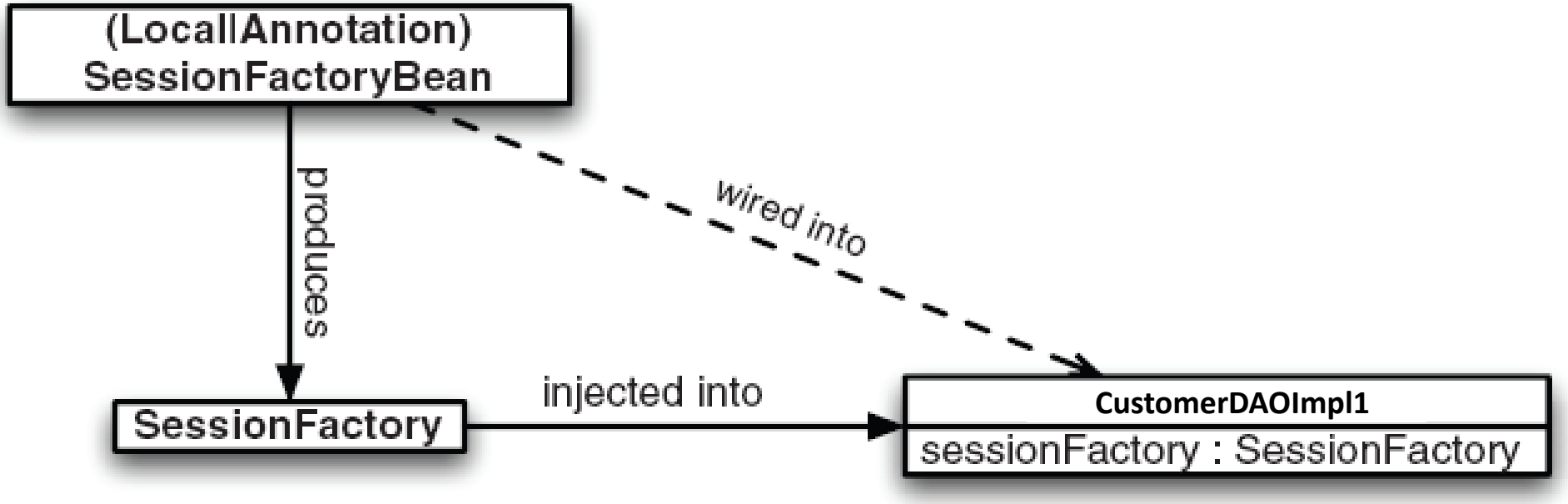


Figure 5.9 Taking advantage of Hibernate 3 contextual sessions, we can wire a `SessionFactory` (produced by a session factory bean) directly into a DAO, thus decoupling the DAO class from the Spring API.

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:

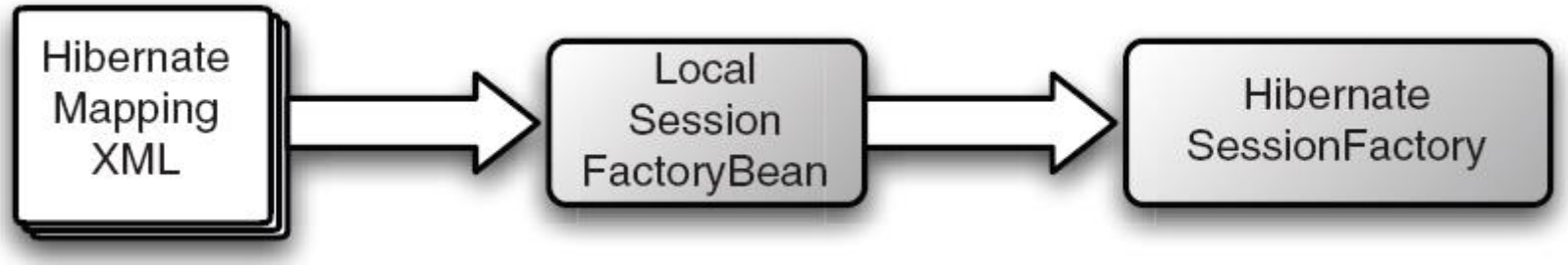
```
<bean id="customerDAO"
        class="com.jediver.spring.orm.hibernate.using.dal.dao.impl.CustomerDAOImpl">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

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.



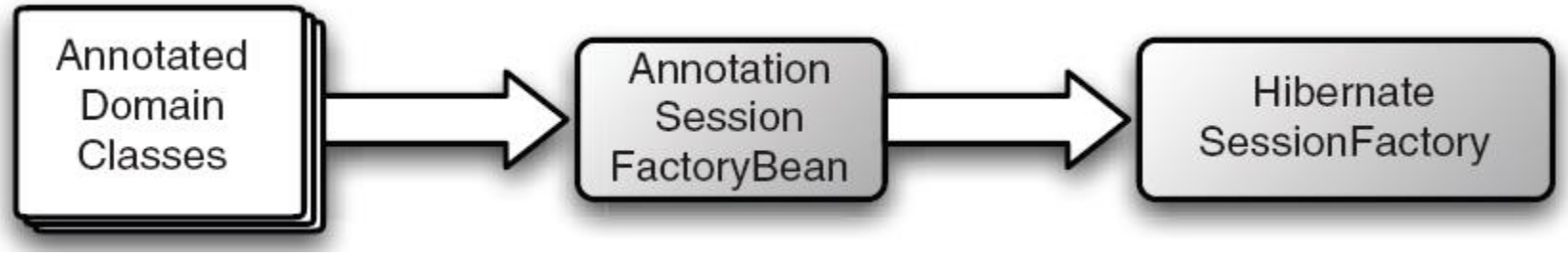
Using Hibernate Session Model (Ex.)

- 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.)

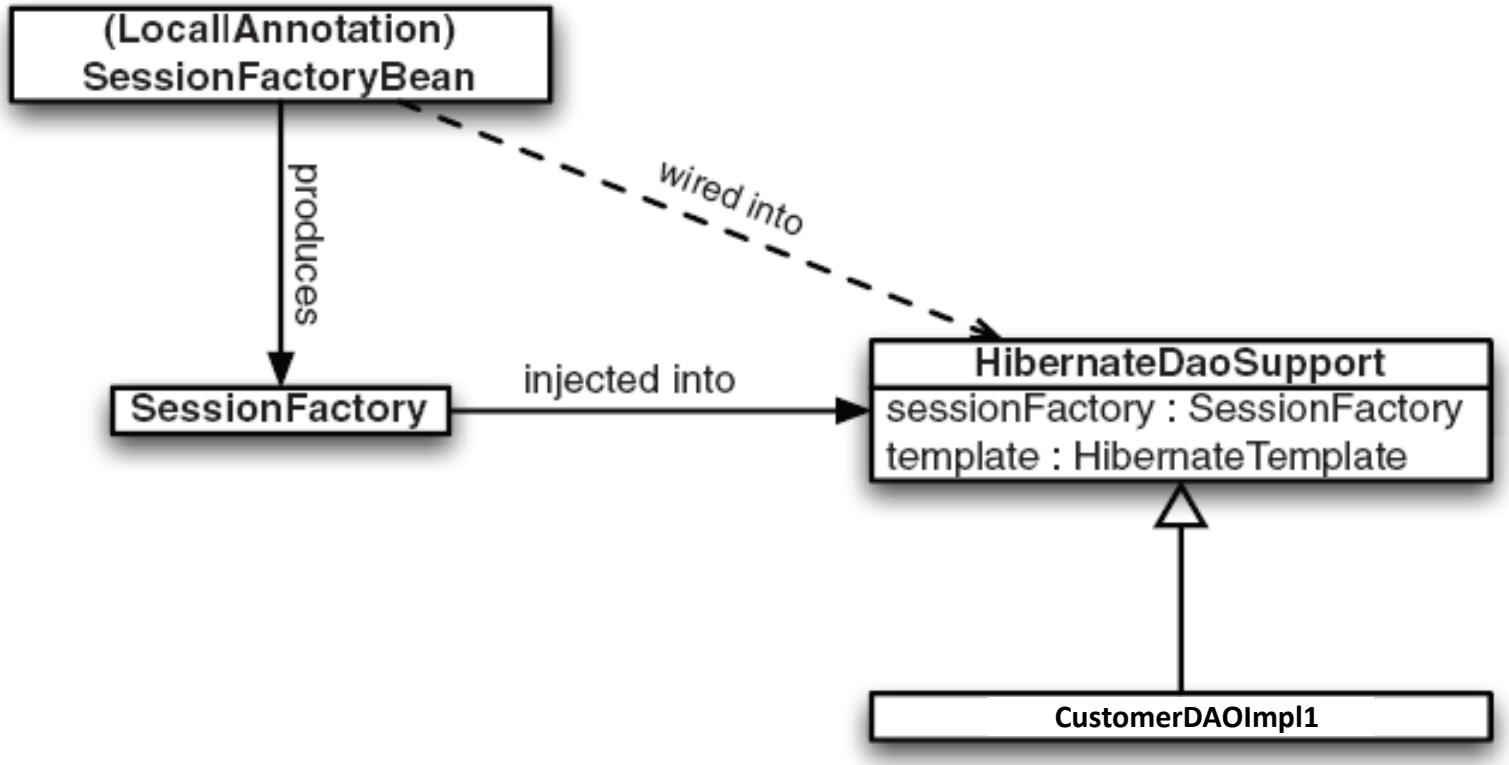


Figure 5.8 `HibernateDaoSupport` is a convenient superclass for a Hibernate-based DAO that provides a `HibernateTemplate` created from an injected `SessionFactory`.

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.

Java™ Education and Technology Services

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")
```

Lesson 6

Spring ORM
using JPA Framework

Spring ORM using JPA Framework

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

```
public class Factory {

    public EntityManagerFactory getEntityManagerFactory() {
        String persistenceUnit = "customerPU";
        return Persistence
                .createEntityManagerFactory(persistenceUnit);
    }
}
```

Spring ORM using JPA 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.jpa.using.dal.cfg.Factory"/>
    <bean id="entityManagerFactory"
          factory-bean="factory"
          factory-method="getEntityManagerFactory"/>
    <bean id="customerDAO"
          class="com.jediver.spring.orm.jpa.using.dal.dao.impl.CustomerDAOImpl">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
    </bean>
</beans>
```

Spring ORM using JPA Framework (Ex.)

- DAOs Injection for entityManager:

```
public class CustomerDAOImpl implements CustomerDAO {

    private EntityManager entityManager;

    public void setEntityManagerFactory(EntityManagerFactory entityManagerFactory) {
        entityManager = entityManagerFactory.createEntityManager();
    }
```

Spring ORM using JPA 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 = entityManager.createQuery(queryString);
    return (long) query.getSingleResult();
}
```

Spring ORM using JPA 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 = entityManager.createQuery(queryString);
    query.setParameter("age", age);
    return (long) query.getSingleResult();

}
```

Spring ORM using JPA Framework (Ex.)

Querying (SELECT)

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

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

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

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

Spring ORM using JPA Framework (Ex.)

Updating (insert)

- The following function insert new record of type customer:

```
@Override
public Customer save(Customer customer) {
    entityManager.getTransaction().begin();
    entityManager.persist(customer);
    entityManager.getTransaction().commit();
    return customer;
}
```

Spring ORM using JPA 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) {
    entityManager.getTransaction().begin();
    entityManager.merge(customer);
    entityManager.getTransaction().commit();
}
```

Spring ORM using JPA Framework (Ex.)

Updating (delete)

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

```
@Override
public void delete(Customer customer) {
    entityManager.getTransaction().begin();
    entityManager.remove(customer);
    entityManager.getTransaction().commit();
}
```

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

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

Spring ORM using JPA Framework (Ex.)

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

Lesson 7

Spring ORM
JPA Integration

Object Relational Mapping (ORM) (Ex.)

- Spring-orm under version of 3.1.0.RELEASE
  - Supports JPATemplate, DAO Support and @Transactional
- Spring-orm from version of 3.1.0.RELEASE to version 3.2.18.RELEASE
  - JPATemplate and DAO Support becomes deprecated.
- Spring-orm from version of 4.0.0.RELEASE
  - JPATemplate and DAO Support becomes not supported anymore .

Object Relational Mapping (ORM) (Ex.)

- You can find the Spring ORM (JPA) support
  - under the org.springframework.orm.jpa package
- Offers comprehensive support for the Java Persistence API like the integration with Hibernate or JDO.
- While being aware of the underlying implementation in order to provide additional features.

Object Relational Mapping (ORM) (Ex.)

- To Integrate between Spring and JPA You Have Different Model:

1. Using JPA EntityManager
   - Query By using EntityManager from JPA EntityManager Factory that managed by Spring Container.
   - Update By using EntityManager from JPA EntityManager Factory that managed by Spring Container.
2. Using JPATemplate Only
   - Query By using JPATemplate.
   - Update By executing updates with JPACallback with JPA EntityManager.
3. Using JPATemplate and TransactionTemplate
   - Query By using JPATemplate.
   - Update By using TransactionTemplate with TransactionCallback with JPATemplate.

Object Relational Mapping (ORM) (Ex.)

- To Integrate between Spring and JPA You Have Different Model:

    4. Using JPA EntityManager and @Transactional

        - Query By using EntityManager By @PersistenceContext.
        - Update By using @Transactional with EntityManager.

Using JPA EntityManager Model

- 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 JPA EntityManagerFactory.

- So after you create JPA EntityManagerFactory you can use it:
  - Either Different EntityManager management as JPA supports
  - Or Manually managing EntityManager.

Using JPA EntityManager 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
hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
hibernate.hbm2ddl.auto=create-drop
```

- The following definition of your property-placeholder:

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

Using JPA EntityManager 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 JPA EntityManager Model (Ex.)

- Spring JPA Integration Provide:
  - JPA Full features.
  - Session managed by JPA.
  - Transaction managed by User or using @Transactional.
  - Stored procedure calls
  - Safely inject this shared reference into multiple DAOs
- You can create JPA EntityManagerFactory by spring from

  org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean:

Using JPA EntityManager Model (Ex.)

- The following definition of your JPA EntityManagerFactory:

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceProvider">
        <bean class="org.hibernate.jpa.HibernatePersistenceProvider" />
    </property>
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="com.jediver.spring.model.dal.entity" />
    <property name="jpaProperties">
        <props>
            <prop key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
            <prop key="hibernate.dialect">${hibernate.dialect}</prop>
        </props>
    </property>
</bean>
```

Using JPA EntityManager Model (Ex.)

- DAOs Injection for EntityManager:

```
public class CustomerDAOImpl implements CustomerDAO {

    private EntityManager entityManager;

    public void setEntityManagerFactory(
            EntityManagerFactory entityManagerFactory) {
        entityManager = entityManagerFactory.createEntityManager();
    }
```

- Beans definition:

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

Using JPA EntityManager Model (Ex.)

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

Using JPATemplate Only

- Spring's JPATemplate provides an abstract layer over a JPA EntityManager.
- JPATemplate's main responsibility is to
  - Simplify the work of opening and closing JPA EntityManager
  - Convert JPA-specific exceptions to one of the Spring ORM exceptions

Using JPATemplate Only (Ex.)

- DAOs Injection for JPATemplate Not supported from 4.0.0.RELEASE:
  - Either By use JPATemplate composition inside DAO.

```
public class CustomerDAOImpl1 implements CustomerDAO {

    JpaTemplate jpaTemplate;

    public void setJpaTemplate(JpaTemplate jpaTemplate) {
        this.jpaTemplate = jpaTemplate;
    }
```

Using JPATemplate Only (Ex.)

- DAOs Injection for JPATemplate Not supported from 4.0.0.RELEASE:
  - Or extends JPADaoSupport.
  - Spring offers JPADaoSupport, a convenience DAO support class, that enables you to wire a EntityManager factory bean directly into the DAO class.
  - Under the covers, JPADaoSupport creates a JPATemplate that the DAO can use.
  - The first step is to change CustomerDAOImpl1 to extend JPADaoSupport:

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

Using JPATemplate Only (Ex.)

- JPATemplate Bean definition:

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

- CustomerDAO Bean definition:

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

Using JPATemplate 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 = getJpaTemplate().find(queryString);
    return (long) result.get(0);
}
```

Using JPATemplate 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";
    Map<String, Object> params = new HashMap<>();
    params.put("age", age);
    return (long) getJpaTemplate()
            .findByNamedParams(queryString, params).get(0);
}
```

Using JPATemplate Only (Ex.)

Querying (SELECT)

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

```
@Override
public Customer findOne(Integer customerId) {
    return getJpaTemplate()
            .find(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>) getJpaTemplate()
            .find(queryString);
}
```

Using JPATemplate Only (Ex.)

Updating (insert)

```
@Override
public Customer save(Customer customer) {
    getJpaTemplate().execute(new JpaCallback<Object>() {
        @Override
        public Object doInJpa(EntityManager entityManager)throws PersistenceException {
            entityManager.getTransaction().begin();
            entityManager.persist(customer);
            entityManager.getTransaction().commit();
            return null;
        }
    });
    return customer;
}
```

Using JPATemplate Only (Ex.)

Updating (update)

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

```
@Override
public void update(Customer customer) {
    getJpaTemplate().execute(new JpaCallback<Object>() {
        @Override
        public Object doInJpa(EntityManager entityManager) throws PersistenceException {
            entityManager.getTransaction().begin();
            entityManager.merge(customer);
            entityManager.getTransaction().commit();
            return null;
        }
    });
}
```

Using JPATemplate Only (Ex.)

Updating (delete)

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

```
@Override
public void delete(Customer customer) {
    getJpaTemplate().execute(new JpaCallback<Object>() {
        @Override
        public Object doInJpa(EntityManager entityManager) throws PersistenceException {
            entityManager.getTransaction().begin();
            entityManager.remove(customer);
            entityManager.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 JPATemplate Only (Ex.)

- **Note:**
  - But 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 JPATemplate is to manage JPA EntityManagers.
  - This involves opening and closing EntityManagers as well as ensuring one EntityManager per transaction.

Using JPATemplate and TransactionTemplate

- Spring's JPATemplate provides an abstract layer over a JPA EntityManager.
- JPATemplate's main responsibility is to
  - Simplify the work of opening and closing JPA EntityManagers
  - Convert JPA-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 JPATemplate and TransactionTemplate (Ex.)

- DAOs Injection for JPATemplate and transactionTemplate:

```
public class CustomerDAOImpl2 implements CustomerDAO {

    private ~~JpaTemplate~~ jpaTemplate;
    private TransactionTemplate transactionTemplate;

    public void setJpaTemplate(~~JpaTemplate~~ jpaTemplate) {
        this.jpaTemplate = jpaTemplate;
    }

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

Using JPATemplate and TransactionTemplate (Ex.)

- JPATemplate Bean definition as usual:

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

- TransactionManager and TransactionTemplate Bean definition as usual:

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

Using JPATemplate and TransactionTemplate (Ex.)

- CustomerDAO Bean definition as usual:

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

Using JPATemplate and TransactionTemplate (Ex.)

Querying (SELECT)

- For all selecting queries as normal using normal JPATemplate
- 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 = jpaTemplate.find(queryString);
    return (long) result.get(0);
}
```

Using JPATemplate and TransactionTemplate (Ex.)

Updating (insert)

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

Using JPATemplate 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) {
            jpaTemplate.merge(customer);
            return ts;
        }
    });
}
```

Using JPATemplate 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) {
            jpaTemplate.remove(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 JPATemplate and TransactionTemplate (Ex.)

- **Note:**
  - But notice that we use TransactionTemplate 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 JPATemplate is to manage JPA EntityManagers.
  - This involves opening and closing EntityManagers as well as ensuring one EntityManager per transaction.
  - One of the responsibilities of TransactionTemplate is to manage Transaction (Commits and Rollback) instead of JPA or user.

Using @PersistenceContext and @Transactional

- Instead of creating JPA EntityManager from Entitymanager Factory.
  - Spring provide implementation for @PersistenceContext to inject directly managed instance from entitymanager.
- 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 @PersistenceContext and @Transactional (Ex.)

- DAOs Injection for EntityManager:

```
public class CustomerDAOImpl1 implements CustomerDAO {

    @PersistenceContext
    private EntityManager entityManager;
```

Using @PersistenceContext 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 />
```

Annotation-based Configuration (Ex.)

- First of all you must enable Spring Context understand @PersistenceContext annotation so you must import it by context namespace:

```
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context.xsd"
```

- <annotation-config> Tag.

```
<context:annotation-config/>
```

Using @PersistenceContext and @Transactional (Ex.)

- We need only to define bean definition for TransactionManager:

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

- CustomerDAO Bean definition as usual:

```
<bean id="customerDAO"
      class="com.jediver.spring.orm.jpa.using.dal.dao.impl.CustomerDAOImpl1"/>
```

Using @PersistenceContext and @Transactional (Ex.)

Querying (SELECT)

- For all selecting queries as normal using normal EntityManager
- 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 = entityManager.createQuery(queryString);
    return (long) query.getSingleResult();
}
```

Using @PersistenceContext 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) {
    entityManager.persist(customer);
    return customer;
}
```

Using @PersistenceContext and @Transactional (Ex.)

Updating (update)

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

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

Using @PersistenceContext and @Transactional (Ex.)

Updating (delete)

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

```
@Transactional
@Override
public void delete(Customer customer) {
    entityManager.remove(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 @PersistenceContext and @Transactional (Ex.)

- **How does @PersistenceContext work?**
- How can @PersistenceContext inject an entity manager only once at container startup time, given that entity managers are so short lived, and that there are usually multiple per request.

- The answer is that it can't: EntityManager is an interface, and what gets injected in the spring bean is not the entity manager itself but a context aware proxy usually SharedEntityManagerInvocationHandler that will delegate to a concrete entity manager at runtime.

Using @PersistenceContext and @Transactional (Ex.)

- **How does @PersistenceContext work?**
- The @PersistenceContext annotation has an optional attribute called type, its default value is PersistenceContextType.TRANSACTION.
- This means that you will receive a shared EntityManager proxy.
- The alternative, PersistenceContextType.EXTENDED, will result in a so-called extended **EntityManager**, which is not thread-safe, so it must not be used in a concurrently accessed component, such as a Spring-managed singleton bean.
- Extended **EntityManager** instances are 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.

Using @PersistenceContext and @Transactional (Ex.)

- **Note:**
  - One of the responsibilities of EntityManager is to manage JPA EntityManagers.
  - This involves opening and closing EntityManagers as well as ensuring one EntityManager per transaction.
  - One of the responsibilities of @Transactional is to manage Transaction (Commits and Rollback) instead of JPA or user.

It’s the most recommended model to declare transaction so you didn’t coupled with any implementations, And follows the JPA Standards.

References & Recommended Reading

References & Recommended Reading

- Spring Framework Documentation Version 5.1.6.RELEASE
- Spring Data Documentation Version 2.1.6.RELEASE
- Spring in Action 5th Edition
- Cloud Native Java
- Learning Spring Boot 2.0
- Spring 5 Recipes: A Problem-Solution Approach