

Spring Data Access (Persistence)

Sang Shin

www.javapassion.com

“Learn with Passion!”



Disclaimer

- Many slides of this presentation are based on the Spring Framework Reference Documentation
 - > <http://static.springsource.org/spring/docs/3.0.x/reference/index.html>

Topics

- DAO support
- *@Repository* annotation
- Data access through JDBC
 - > *JdbcTemplate* class
 - > *NamedParameterJdbcTemplate* class
 - > *SimpleJdbcTemplate* and *SimpleJdbcDaoSupport* classes
- Data access through ORM
 - > Hibernate
 - > JPA

DAO Support

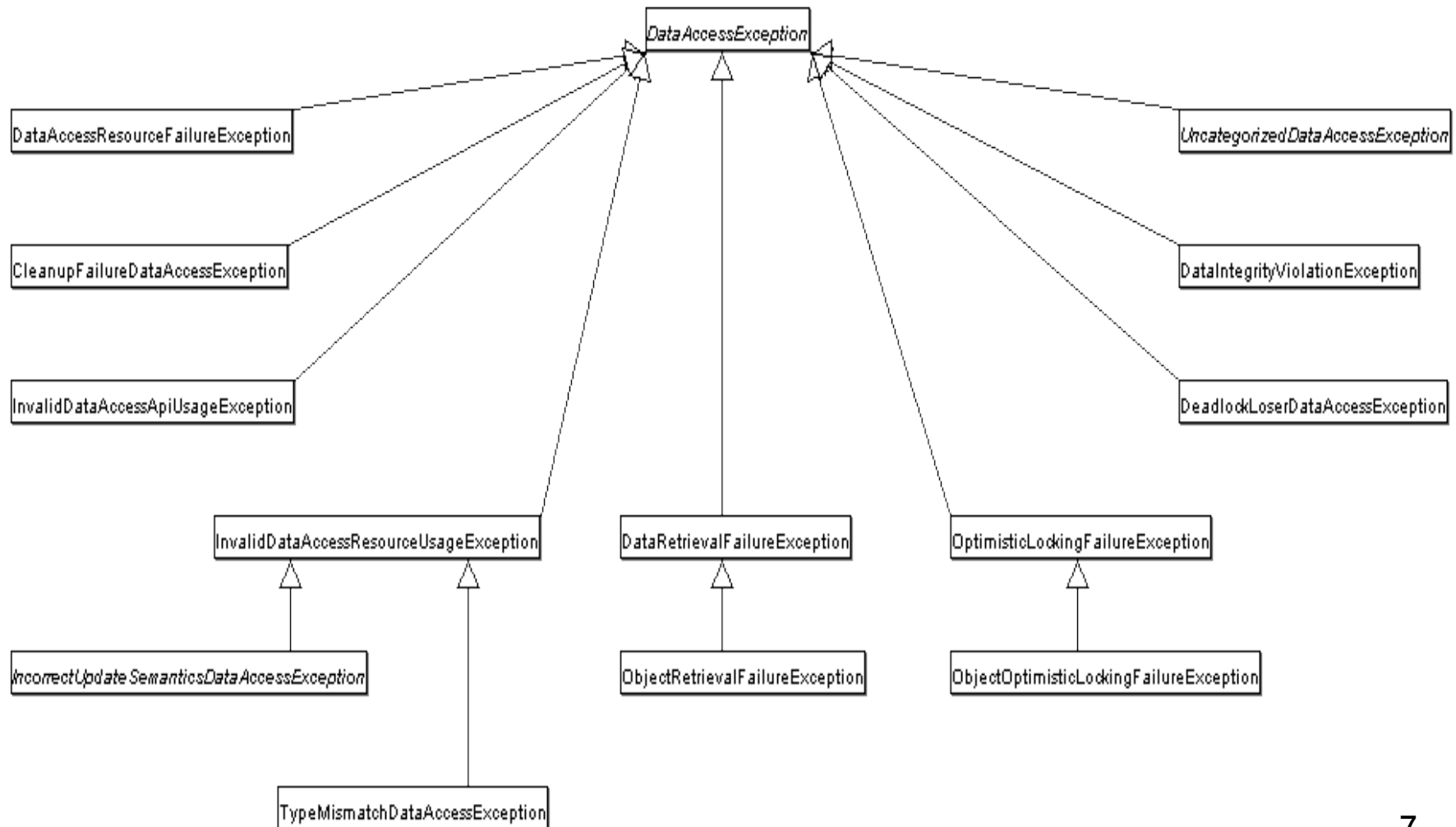
Why Dao?

- The Data Access Object (DAO) support in Spring is aimed at providing consistency/portability to your code regardless what data access(persistence) technologies like JDBC, Hibernate, JPA or JDO are used underneath
 - > Switching to a different persistence technology is a matter of changing a few lines in the configuration file
 - > It also allows one to code without worrying about catching exceptions that are specific to each persistence technology

Consistent Exception Hierarchy

- Spring provides a convenient translation from persistence technology-specific exceptions like *SQLException* to its own exception class hierarchy with the *DataAccessException* as the root exception.
- These exceptions wrap the original exception so there is never any risk that one might lose any information as to what might have gone wrong
- Handles both JDBC and Hibernate exceptions

Exception Hierarchy



@Repository Annotation

@Repository annotation

- Guarantees that your Data Access Objects (DAOs) or repositories provide exception translation
- Allows the component scanning support to find and configure your DAOs and repositories without having to provide XML configuration entries for them
 - > @Repository is a stereotype of @Component

@Repository

```
public class SomeMovieFinder implements MovieFinder {  
  
    // ...  
  
}
```

Access to a persistence resource

- Any DAO or repository implementation will need to access to a persistence resource, depending on the persistence technology used
 - > JDBC-based repository need access to a *JDBC DataSource*
 - > Hibernate-based repository need access to a *SessionFactory*
 - > JPA-based repository need access to an *EntityManager*
- The easiest way to accomplish this is to have this resource dependency injected using
 - > *@Autowired* for *JDBC DataSource* and *Hibernate SessionFactory*
 - > *@PersistenceContext* for *JPA EntityManager*

Injecting JDBC DataSource

- You would have the *DataSource* object injected into an initialization method where you would create a *JdbcTemplate* and other data access support classes like *SimpleJdbcCall*

```
@Repository
public class JdbcMovieFinder implements MovieFinder {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void init(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // ...

}
```

Injecting Hibernate SessionFactory

```
@Repository
public class HibernateMovieFinder implements MovieFinder {

    private SessionFactory sessionFactory;

    @Autowired
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    // ...

}
```

Injecting JPA Entity Manager

```
@Repository  
public class JpaMovieFinder implements MovieFinder {  
  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    // ...  
}
```

Data Access with JDBC

Spring Value-Add to JDBC

- Spring JDBC - who does what? - The Spring Framework takes care of all the low-level JDBC details

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, statement and resultset.	X	

Options to Choose for JDBC Access

- You can use *JdbcTemplate* - 3 flavors
 - > *JdbcTemplate*
 - > *NamedParameterJdbcTemplate*
 - > *SimpleJdbcTemplate*
- *SimpleJdbcInsert* and *SimpleJdbcCall*
 - > Optimizes database metadata
- RDMBX Objects including *MappingSqlQuery*, *SqlUpdate* and *StoredProcedure*
 - > More object-oriented approach similar to that of JDO Query design

JdbcTemplate Class

What does JdbcTemplate do?

- Handles the creation and release of resources, which helps you avoid common errors such as forgetting to close the connection.
- Performs the basic tasks of the core JDBC workflow statement creation and execution, leaving application code to provide SQL and extract results.
- Executes SQL queries, update statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values
- Catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy

JdbcTemplate Query Examples

```
// Simple query for getting the number of rows
int rowCount = this.jdbcTemplate.queryForInt("select count(*) from t_actor");

// A simple query using a bind variable:
int countOfActorsNamedJoe = this.jdbcTemplate.queryForInt(
    "select count(*) from t_actor where first_name = ?", "Joe");

// Querying for a String
String lastName = this.jdbcTemplate.queryForObject(
    "select last_name from t_actor where id = ?", // SQL query to execute
    new Object[]{1212L}, // arguments to bind to the query
    String.class); // requiredType - the type that the result object is expected to match
```

JdbcTemplate Query Examples

```
// Querying and populating a single domain object
Actor actor = this.jdbcTemplate.queryForObject(
    "select first_name, last_name from t_actor where id = ?",
    new Object[]{1212L},
    // RowMapper interface is used by JdbcTemplate for mapping rows of a ResultSet on a
    // per-row basis. Implementations of this interface perform the actual work of mapping
    // each row to a result object, but don't need to worry about exception handling.
    // SQLExceptions will be caught and handled by the calling JdbcTemplate.
    new RowMapper<Actor>() {
        public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setFirstName(rs.getString("first_name"));
            actor.setLastName(rs.getString("last_name"));
            return actor;
        }
    });
```

JdbcTemplate Update Examples

```
// You use the update(..) method to perform insert, update and delete
// operations. Parameter values are usually provided as var args or
// alternatively as an object array.
```

```
// Insert operation
this.jdbcTemplate.update(
    "insert into t_actor (first_name, last_name) values (?, ?)",
    "Leonor", "Watling");
```

```
// Update operation
this.jdbcTemplate.update(
    "update t_actor set = ? where id = ?",
    "Banjo", 5276L);
```

```
// Delete operation
this.jdbcTemplate.update(
    "delete from actor where id = ?",
    Long.valueOf(actorId));
```

JdbcTemplate Execute Examples

```
// You can use the execute(..) method to execute any arbitrary SQL, and  
// as such the method is often used for DDL statements. It is heavily  
// overloaded with variants taking callback interfaces, binding variable  
// arrays, and so on.
```

```
// Execute an arbitrary SQL  
this.jdbcTemplate.execute("create table mytable (id integer,  
                                                                    name varchar(100))");
```

```
// Invokes a simple stored procedure  
this.jdbcTemplate.update(  
    "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",  
    Long.valueOf(unionId));
```

NamedParameterJdbcTemplate Class

NamedParameterJdbcTemplate

- The *NamedParameterJdbcTemplate* class adds support for programming JDBC statements using named parameters, as opposed to programming JDBC statements using only classic placeholder ('?') arguments
 - > More flexible (since you don't have to worry about the order of parameters)
- The *NamedParameterJdbcTemplate* class wraps a *JdbcTemplate*, and delegates to the wrapped *JdbcTemplate* to do much of its work

SqlParameterSource Interface

- Source of named parameter values to a *NamedParameterJdbcTemplate*
- Implementations
 - > *MapSqlParameterSource*
 - > *BeanPropertySqlParameterSource* -wraps an arbitrary JavaBean, and uses the properties of the wrapped JavaBean as the source of named parameter values

NamedParameterJdbcTemplate

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;  
  
public void setDataSource(DataSource dataSource) {  
    this.namedParameterJdbcTemplate =  
        new NamedParameterJdbcTemplate(dataSource);  
}  
  
public int countOfActorsByFirstName(String firstName) {  
    String sql = "select count(*) from T_ACTOR where first_name = :first_name";  
  
    SqlParameterSource namedParameters =  
        new MapSqlParameterSource("first_name", firstName);  
  
    return namedParameterJdbcTemplate.queryForInt(sql, namedParameters);  
}
```

SimpleJdbcTemplate & SimpleJdbcDaoSupport

SimpleJdbcTemplate

- The *SimpleJdbcTemplate* class wraps the classic *JdbcTemplate* and leverages Java 5 language features such as varargs and autoboxing

Classic JdbcTemplate Style

```
// Classic JdbcTemplate-style is shown here as a comparison to the SimpleJdbcTemplate
private JdbcTemplate jdbcTemplate;
```

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

```
public Actor findActor(String specialty, int age) {
```

```
    String sql = "select id, first_name, last_name from T_ACTOR" +
        " where specialty = ? and age = ?";
```

```
    RowMapper<Actor> mapper = new RowMapper<Actor>() {
        public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setId(rs.getLong("id"));
            actor.setFirstName(rs.getString("first_name"));
            actor.setLastName(rs.getString("last_name"));
            return actor;
        }
    };
```

```
// notice the wrapping up of the arguments in an array
```

```
return (Actor) jdbcTemplate.queryForObject(sql, new Object[] {specialty, age}, mapper);
}
```

SimpleJdbcTemplate (with Varargs)

```
// SimpleJdbcTemplate-style...
private SimpleJdbcTemplate simpleJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.simpleJdbcTemplate = new SimpleJdbcTemplate(dataSource);
}

public Actor findActor(String specialty, int age) {

    String sql = "select id, first_name, last_name from T_ACTOR" +
        " where specialty = ? and age = ?";
    RowMapper<Actor> mapper = new RowMapper<Actor>() {
        public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setId(rs.getLong("id"));
            actor.setFirstName(rs.getString("first_name"));
            actor.setLastName(rs.getString("last_name"));
            return actor;
        }
    };

    // notice the use of varargs since the parameter values now come
    // after the RowMapper parameter
    return this.simpleJdbcTemplate.queryForObject(sql, mapper, specialty, age);
}
```

SimpleJdbcDaoSupport

- Extension of *JdbcDaoSupport* that exposes a *SimpleJdbcTemplate* as well.
- Only usable on Java 5 and above

SimpleJdbcDaoSupport

```
public class JdbcStudentDao extends SimpleJdbcDaoSupport implements StudentDao {

    public void insert(Student student) {

        String sql = "INSERT INTO STUDENT (STUDENT_NO, NAME, BIRTHDAY, GRADE) " +
            "VALUES (?, ?, ?, ?)";
        getSimpleJdbcTemplate().update(sql,
            student.getStudentNo(), student.getName(),
            student.getBirthday(), student.getGrade());
    }

    // Use named parameters - more flexible
    public void update(Student student) {
        String sql = "UPDATE STUDENT SET NAME = :name, GRADE = :grade,
            BIRTHDAY = :birthday WHERE STUDENT_NO = :studentNo";
        SqlParameterSource parameterSource =
            new BeanPropertySqlParameterSource(student);
        getSimpleJdbcTemplate().update(sql, parameterSource);
    }
}
```

Demo:

[database_jdbc_SimpleJdbcDaoSupport](#)

[database_jdbc_SimpleJdbcDaoSupport_errorhandling](#)

[database_jdbc_SimpleJdbcDaoSupport_mysql](#)

[4941_spring3_database.zip](#)



ORM Support in Spring

ORM Support in Spring

- Spring supports resource management, data access object (DAO) implementations, and transaction strategies
 - > Hibernate
 - > JPA (Java Persistence API)
 - > JDO
 - > iBATIS SQL Maps

Benefits of using Spring ORM DAO's

- Easier testing
- Common data access exceptions
- General resource management
 - > Location and configuration of Hibernate SessionFactory instances, JPA EntityManagerFactory instances are easily handled through Spring application contexts
- 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

Hibernate

Two Options For Mapping Domain Class

- Option #1: Use a XML mapping file for describing the mapping between a domain class and a database table
 - > *Student.hbm.xml*
- Option #2: Use JPA annotation in the domain class
 - > Use *@Entity* annotation in the domain class itself
 - > No need to have XML mapping file

Option #1: Hibernate Context Configuration

```
<beans>
```

```
...
```

```
<!-- Hibernate SessionFactory -->
```

```
<bean id="sessionFactory"
```

```
  class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
```

```
  <property name="dataSource" ref="dataSource" />
```

```
  <property name="mappingResources">
```

```
    <list>
```

```
      <value>Student.hbm.xml</value>
```

```
    </list>
```

```
  </property>
```

```
  <property name="hibernateProperties">
```

```
    <props>
```

```
      <prop key="hibernate.dialect">${hibernate.dialect}</prop>
```

```
      <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
```

```
      <prop key="hibernate.generate_statistics">${hibernate.generate_statistics}</prop>
```

```
    </props>
```

```
  </property>
```

```
</bean>
```

```
</beans>
```

Option #1: Mapping file - Student.hbm.xml

```
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.javapassion.examples.student.domain">
  <class name="Student" table="STUDENT">
    <id name="id" type="int" column="ID">
      <generator class="identity"/>
    </id>
    <property name="name" type="string">
      <column name="NAME" length="20" not-null="true"/>
    </property>
    <property name="birthday" type="date" column="BIRTHDAY"/>
    <property name="grade" type="int" column="GRADE"/>
  </class>
</hibernate-mapping>
```

Option #2: Use @Entity Annotation in Domain class

```
<beans>
```

```
...
```

```
<!-- Hibernate SessionFactory -->
```

```
<bean id="sessionFactory"
```

```
  class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
```

```
  <property name="dataSource" ref="dataSource" />
```

```
  <property name="annotatedClasses">
```

```
    <list>
```

```
      <value>com.javapassion.examples.student.domain.Student</value>
```

```
    </list>
```

```
  </property>
```

```
  <property name="hibernateProperties">
```

```
    <props>
```

```
      <prop key="hibernate.dialect">${hibernate.dialect}</prop>
```

```
      <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
```

```
      <prop key="hibernate.generate_statistics">${hibernate.generate_statistics}</prop>
```

```
    </props>
```

```
  </property>
```

```
</bean>
```

```
</beans>
```

Option #2: Use @Entity Annotation in Domain class

```
@Entity
@Table(name = "STUDENT")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID")
    private Integer id;
    @Column(name = "NAME", length = 20, nullable = false)
    private String name;
    @Column(name = "BIRTHDAY")
    private Date birthday;
    @Column(name = "GRADE")
    private int grade;

    public Student() {
    }

    public Student(String name, Date birthday, int grade) {
        this.name = name;
        this.birthday = birthday;
        this.grade = grade;
    }

    ....
}
```

Two Options For Programming Model

- Option #1: Use *SessionFactory*
 - > SessionFactory is from Hibernate and expose Hibernate specific exceptions
- Option #2: Use *HibernateTemplate*
 - > HibernateTemplate is a helper class that simplifies Hibernate data access code.
 - > Automatically converts HibernateExceptions into DataAccessExceptions, following the org.springframework.dao exception hierarchy
 - > Recommended over Option #1

Option #1: Using SessionFactory

```
@Repository("studentDao")
public class HibernateStudentDao implements StudentDao {

    @Autowired
    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    @Transactional
    public void store(Student student) {
        sessionFactory.getCurrentSession().saveOrUpdate(student);
    }

    @Transactional
    public void delete(Integer studentId) {
        Student student =
            (Student) sessionFactory.getCurrentSession().get(Student.class, studentId);
        sessionFactory.getCurrentSession().delete(student);
    }

    @Transactional(readOnly = true)
    public Student findById(Integer studentId) {
        return (Student) sessionFactory.getCurrentSession().get(Student.class, studentId);
    }
}
```

Option #2: Using HibernateTemplate

```
@Repository("studentDao")
public class HibernateStudentDao implements StudentDao {

    @Autowired
    private HibernateTemplate hibernateTemplate;

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

    @Transactional
    public void store(Student student) {
        hibernateTemplate.saveOrUpdate(student);
    }

    @Transactional
    public void delete(Integer studentId) {
        Student student = (Student) hibernateTemplate.get(Student.class, studentId);
        hibernateTemplate.delete(student);
    }

    @Transactional(readOnly = true)
    public Student findById(Integer studentId) {
        return (Student) hibernateTemplate.get(Student.class, studentId);
    }
}
```

Demo:

database_hibernate_xmlmapping_SessionFactory
database_hibernate_annotation_SessionFactory
database_hibernate_annotation_HibernateTemplate
4941_spring3_database.zip



JPA

JPA Engine Selection

- In JPA, a JPA engine (persistence provider) can be selected through configuration
- Possible JPA engines you can use
 - > Hibernate
 - > OpenJPA
 - > EclipseLink

JPA Engine Selection Configuration

```
<!-- JPA EntityManagerFactory -->
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="jpaVendorAdapter">

    <!-- Use Hibernate as JPA engine -->
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
          p:database="{jpa.database}" p:showSql="{jpa.showSql}" />

    <!-- Use OpenJPA as JPA engine -->
    <!-- <bean class="org.springframework.orm.jpa.vendor.OpenJpaVendorAdapter"
          p:database="{jpa.database}" p:showSql="{jpa.showSql}" /> -->

  </property>
  <property name="persistenceXmlLocation"
            value="classpath:META-INF/persistence.xml" />
</bean>
```

Three options for JPA setup

- The Spring JPA support offers three ways of setting up the JPA *EntityManagerFactory* that will be used by the application to obtain an entity manager.
 - > *LocalEntityManagerFactoryBean*
 - > Obtaining an *EntityManagerFactory* from JNDI
 - > *LocalContainerEntityManagerFactoryBean*

LocalEntityManagerFactoryBean

- Only use this option in simple deployment environments such as stand-alone applications and integration tests
 - > This form of JPA deployment is the simplest and the most limited. You cannot refer to an existing JDBC DataSource bean definition and no support for global transactions exists.

```
<beans>
```

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

```
  </bean>
```

```
</beans>
```

Obtaining an EntityManagerFactory from JNDI

- Use this option when deploying to a Java EE 5 server

```
<beans>
```

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

```
</beans>
```

LocalEntityManagerFactoryBean

- Use this option for full JPA capabilities in a Spring-based application environment.
 - > Allows fine-grained customization
 - > Selection of JPA engine

```
<bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="jpaVendorAdapter">

    <!-- Use Hibernate as JPA engine -->
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"
      p:database="{jpa.database}" p:showSql="{jpa.showSql}" />

    <!-- <bean class="org.springframework.orm.jpa.vendor.OpenJpaVendorAdapter"
      p:database="{jpa.database}" p:showSql="{jpa.showSql}"/> -->

  </property>
  <property name="persistenceXmlLocation"
    value="classpath:META-INF/persistence.xml" />
</bean>
```

DAO Implementation using JPA

```
// The DAO has no dependency on Spring and still fits nicely into a Spring application
// context. Moreover, the DAO takes advantage of annotations to require the injection of the
// default EntityManagerFactory
public class ProductDaoImpl implements ProductDao {
```

```
    private EntityManagerFactory emf;
```

```
    @PersistenceUnit
```

```
    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.emf = emf;
    }
```

```
    public Collection loadProductsByCategory(String category) {
        EntityManager em = this.emf.createEntityManager();
        try {
            Query query = em.createQuery("from Product as p where p.category = ?1");
            query.setParameter(1, category);
            return query.getResultList();
        }
        finally {
            if (em != null) {
                em.close();
            }
        }
    }
}
```

Demo:

**database_jpa_hibernate_engine
4941_spring3_database.zip**



Thank you!

Check JavaPassion.com Codecamps!
<http://www.javapassion.com/codecamps>
“Learn with Passion!”

