

Spring Data

Trainer: Ly Quy Duong



Course Objectives

- At the end of the course, you will have acquired sufficient knowledge to:
 - perform objective 1
 - perform objective 2



Agenda

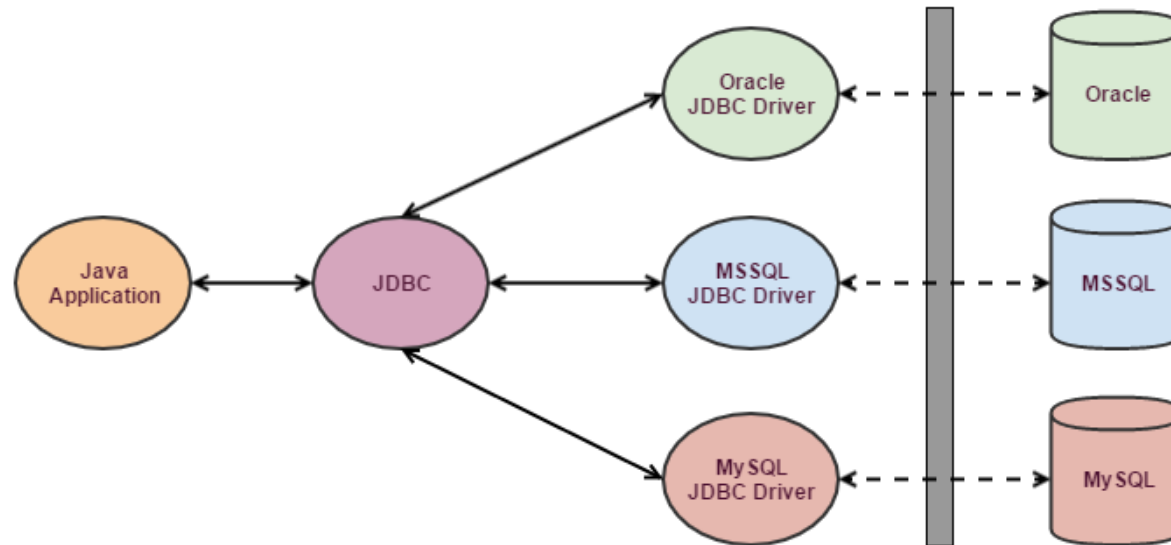
- JDBC
- Hibernate
- Spring Data JDBC
- Spring Data Hibernate

August 15, 2017

JDBC

What is JDBC?

- JDBC (Java Database Connectivity) is a java API to connect and execute query with the database.
- JDBC API uses JDBC drivers to connect with the Relational Database.





Core JDBC Components

- DriverManager
 - This class manages a list of database drivers
- Connection
 - It uses a username, password, and a JDBC URL to establish a connection to the database and returns a connection object
 - A JDBC Connection represents a session/connection with a specific database



Core JDBC Components

- Statement
 - Use to execute queries and updates against the database.
- ResultSet
 - Perform a query against the database to get back a ResultSet. Then traverse this ResultSet to read the result of the query.
- SQLException
 - This class handles any errors that occur in a database application.



Kinds of Statements

- Statement
 - Execute simple SQL queries without parameters
- Prepared Statement
 - Execute precompiled SQL queries with or without parameters
- Callable Statement
 - Execute a call to a database stored procedure

JDBC: Work With MySQL

- Loading the JDBC Driver

```
Class.forName("com.mysql.jdbc.Driver");
```

- Opening the Connection

```
Connection conn = null;  
conn = DriverManager.getConnection(  
    "jdbc:mysql://hostname:port/dbname",  
    "username",  
    "password");
```

- Closing the Connection

```
conn.close();
```

Query the Database

```
CREATE TABLE students (  
    student_id INT NOT NULL AUTO_INCREMENT,  
    fullname CHAR(30) NOT NULL,  
    sex CHAR(1) NOT NULL,  
    address varchar(100),  
    PRIMARY KEY (student_id)  
);
```

```
String sql = "SELECT * FROM students";
```

```
Statement statement = connection.createStatement();  
ResultSet result = statement.executeQuery(sql);
```

```
while (result.next()){  
    String fullname = result.getString("fullname");  
    String sex = result.getString("sex");  
    String address = result.getString("address");  
}
```

Update the Database

- Update records

```
String sql = "UPDATE students SET fullname = 'Nguyen Van A' WHERE student_id = 1";
```

```
Statement statement = null;  
statement = connection.createStatement();  
int rowAffected = statement.executeUpdate(sql);
```

- Delete records

```
String sql = "DELETE FROM students WHERE student_id = 1";
```

```
Statement statement = null;  
statement = connection.createStatement();  
int rowAffected = statement.executeUpdate(sql);
```

Using PreparedStatement

```
String sql = "INSERT INTO students (fullname, sex, address) VALUES (?, ?, ?)";

PreparedStatement preparedStatement = connection.prepareStatement(sql);

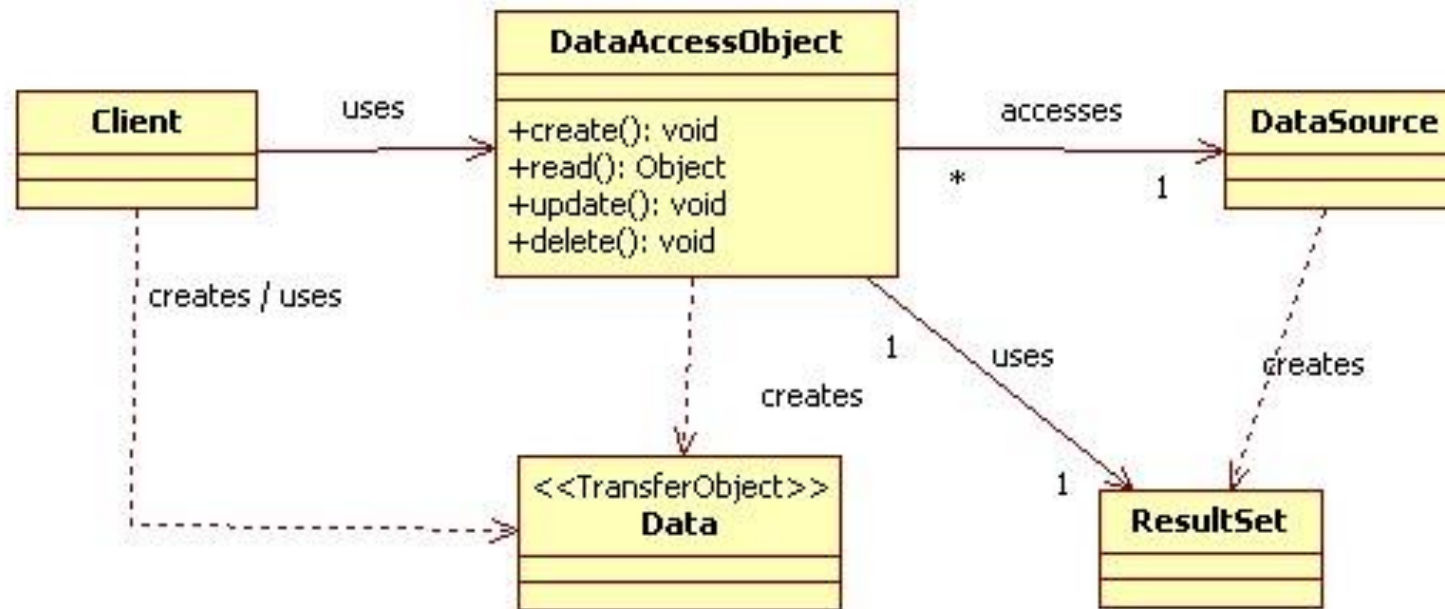
preparedStatement.setString(1, "Tran Minh");
preparedStatement.setString(2, "M");
preparedStatement.setString(3, "Cong Hoa, Tan Binh");

int rowsInserted = preparedStatement.executeUpdate();
```

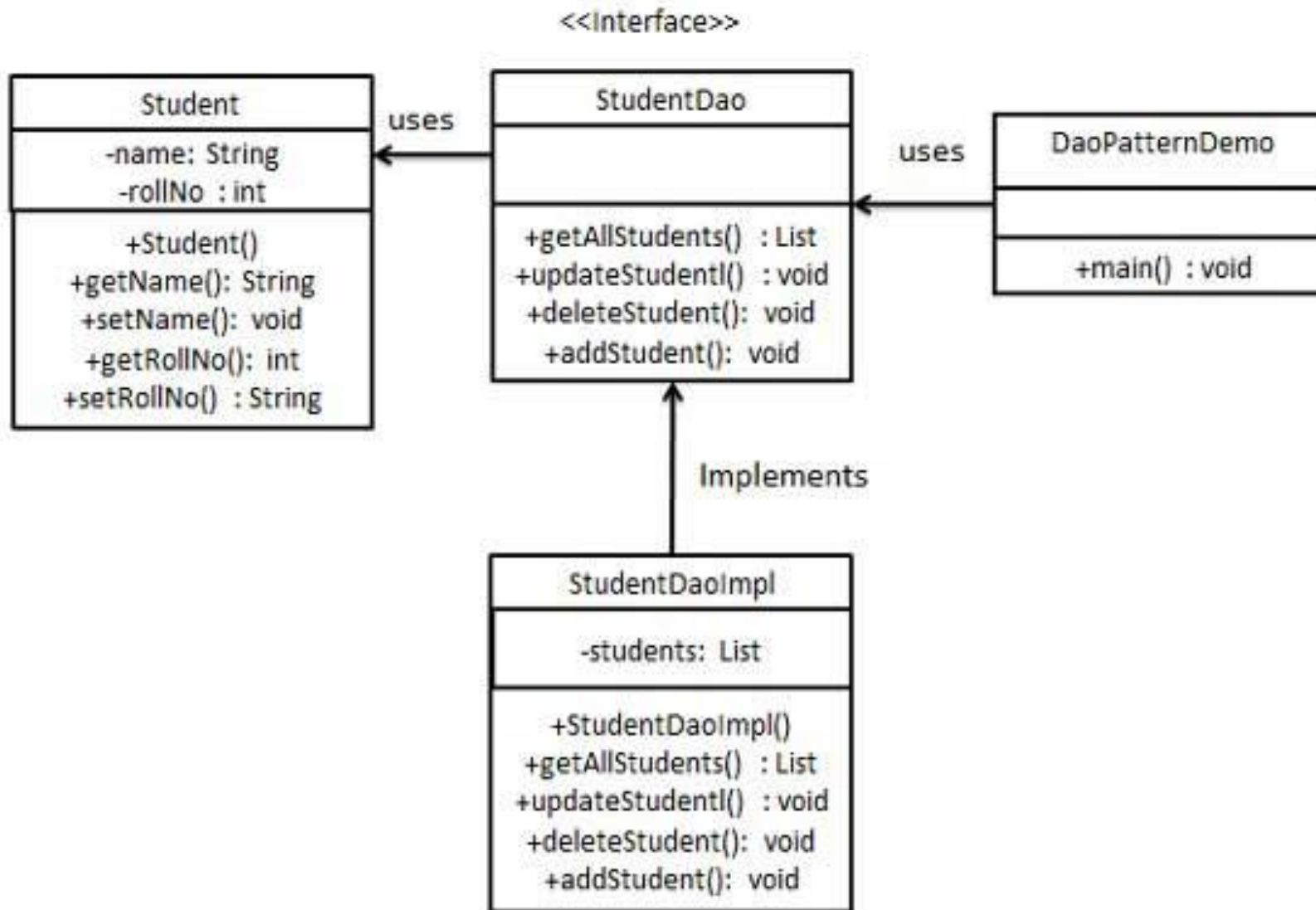
Data Access Object

Data Access Object Pattern

- DAO pattern is used to separate low level data accessing API or operations from high level business services



Sample





The participants in Data Access Object Pattern

- Data Access Object Interface
 - This interface defines the standard operations to be performed on a model object(s)
- Data Access Object concrete class
 - This class implements above interface.
 - This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
- Model Object or Value Object
 - This object is simple POJO containing get/set methods to store data retrieved using DAO class

Implementation

- Student

```
public class Student {  
  
    private int id;  
    private String fullName;  
    private String sex;  
    private String address;  
  
    public Student() {  
        super();  
    }  
}
```

Implementation

- StudentDAO

```
public interface StudentDAO {  
    /**  
     * This is the method to be used to create a record in the Student table.  
     */  
    public void create(String fullName, String sex, String address);  
  
    /**  
     * This is the method to be used to list down all the records from the  
     * Student table.  
     */  
    public List<Student> listStudents();  
}
```

Implementation

- StudentDAOImpl

```
public class StudentDAOImpl implements StudentDAO {  
  
    @Override  
    public void create(String fullName, String sex, String address) {  
        PreparedStatement preparedStatement = null;  
        Connection connection = ConnectionUtil.getCurrentConnection();  
  
        try {  
            String sql = "INSERT INTO students (fullname, sex, address) VALUES (?, ?, ?)";  
  
            preparedStatement = connection.prepareStatement(sql);  
  
            preparedStatement.setString(1, fullName);  
            preparedStatement.setString(2, sex);  
            preparedStatement.setString(3, address);  
  
            int rowsInserted = preparedStatement.executeUpdate();  
            if (rowsInserted > 0) {  
                System.out.println("A new student was inserted successfully!");  
            }  
        } catch (SQLException e) {  
            System.out.println("Connection Failed! Check output console");  
            e.printStackTrace();  
        } finally {  
            ConnectionUtil.cleanup(preparedStatement, connection);  
        }  
    }  
  
    public List<Student> listStudents() {  
          
    }  
}
```

Implementation

- StudentDAODemo

```
public static void main(String[] args) {  
  
    StudentDAO dao = new StudentDAOImpl();  
    dao.create("Tran Hao", "M", "Cong Hoa");  
  
    List<Student> list = dao.listStudents();  
    for (Student student : list) {  
        System.out.println(student.toString());  
    }  
}
```

Spring Data JDBC



Problems of JDBC API

- Need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
- Need to perform exception handling code on the database logic
- Need to handle transaction
- Repetition of all these codes from one to another database logic is a time consuming task



What is Spring JDBC?

- Spring provides a simplification in handling database access with the Spring JDBC Template.
- The Spring JDBC template allows to clean-up the resources automatically, e.g. release the database connections.



Choosing an approach for JDBC database access

- JdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcInsert and SimpleJdbcCall
- RDBMS Objects including MappingSqlQuery, SqlUpdate and StoredProcedure



JdbcTemplate

- The classic Spring JDBC approach and the most popular
- Performs some tasks
 - Executes SQL queries, update statements and stored procedure calls
 - Performs iteration over ResultSets and extraction of returned parameter values

JdbcTemplate

- Querying (SELECT)

```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor", Integer.class);
```

```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(  
    "select count(*) from t_actor where first_name = ?", Integer.class, "Joe");
```

```
String lastName = this.jdbcTemplate.queryForObject(  
    "select last_name from t_actor where id = ?",  
    new Object[]{1212L}, String.class);
```

JdbcTemplate

- RowMapper
 - Map the ResultSet data to bean object in queryForObject() method

```
public List<Actor> findAllActors() {  
    return this.jdbcTemplate.query( "select first_name, last_name from t_actor", new ActorMapper());  
}  
  
private static final class ActorMapper implements 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

- Updating (INSERT/UPDATE/DELETE) with jdbcTemplate

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

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

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

JdbcTemplate

- Other jdbcTemplate operations

```
this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");
```

```
this.jdbcTemplate.update(  
    "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",  
    Long.valueOf(unionId));
```

NamedParameterJdbcTemplate

- Support for programming JDBC statements using named parameters

```
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 this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);  
}
```

SimpleJdbcInsert

- Provides a simplified configuration by taking advantage of database metadata that can be retrieved through the JDBC driver

```
public class JdbcActorDao implements ActorDao {  
  
    private JdbcTemplate jdbcTemplate;  
    private SimpleJdbcInsert insertActor;  
  
    public void setDataSource(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
        this.insertActor = new SimpleJdbcInsert(dataSource).withTableName("t_actor");  
    }  
  
    public void add(Actor actor) {  
        Map<String, Object> parameters = new HashMap<String, Object>(3);  
        parameters.put("id", actor.getId());  
        parameters.put("first_name", actor.getFirstName());  
        parameters.put("last_name", actor.getLastName());  
        insertActor.execute(parameters);  
    }  
  
    // ... additional methods  
}
```

SimpleJdbcCall

- Calling a stored procedure with SimpleJdbcCall

```
CREATE PROCEDURE read_actor (  
    IN in_id INTEGER,  
    OUT out_first_name VARCHAR(100),  
    OUT out_last_name VARCHAR(100),  
    OUT out_birth_date DATE)  
BEGIN  
    SELECT first_name, last_name, birth_date  
    INTO out_first_name, out_last_name, out_birth_date  
    FROM t_actor where id = in_id;  
END;
```


SimpleJdbcCall

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.procReadActor = new SimpleJdbcCall(dataSource)
            .withProcedureName("read_actor");
    }

    public Actor readActor(Long id) {
        SqlParameterSource in = new MapSqlParameterSource()
            .addValue("in_id", id);
        Map out = procReadActor.execute(in);
        Actor actor = new Actor();
        actor.setId(id);
        actor.setFirstName((String) out.get("out_first_name"));
        actor.setLastName((String) out.get("out_last_name"));
        actor.setBirthDate((Date) out.get("out_birth_date"));
        return actor;
    }
}
```

MappingSqlQuery

- MappingSqlQuery is a reusable query in which concrete subclasses must implement the abstract mapRow(..) method to convert each row of the supplied ResultSet into an object of the type specified.

```
public class ActorMappingQuery extends MappingSqlQuery<Actor> {  
  
    public ActorMappingQuery(DataSource ds) {  
        super(ds, "select id, first_name, last_name from t_actor where id = ?");  
        super.declareParameter(new SqlParameter("id", Types.INTEGER));  
        compile();  
    }  
  
    @Override  
    protected 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;  
    }  
}
```

MappingSqlQuery

```
private ActorMappingQuery actorMappingQuery;

@Autowired
public void setDataSource(DataSource dataSource) {
    this.actorMappingQuery = new ActorMappingQuery(dataSource);
}

public Customer getCustomer(Long id) {
    return actorMappingQuery.findObject(id);
}
```

SqlUpdate

- The SqlUpdate class encapsulates an SQL update.
- Like a query, an update object is reusable

```
public class UpdateCreditRating extends SqlUpdate {

    public UpdateCreditRating(DataSource ds) {
        setDataSource(ds);
        setSql("update customer set credit_rating = ? where id = ?");
        declareParameter(new SqlParameter("creditRating", Types.NUMERIC));
        declareParameter(new SqlParameter("id", Types.NUMERIC));
        compile();
    }

    /**
     * @param id for the Customer to be updated
     * @param rating the new value for credit rating
     * @return number of rows updated
     */
    public int execute(int id, int rating) {
        return update(rating, id);
    }
}
```

StoredProcedure

- The StoredProcedure class is a superclass for object abstractions of RDBMS stored procedures

```
private class GetSysdateProcedure extends StoredProcedure {  
  
    private static final String SQL = "sysdate";  
  
    public GetSysdateProcedure(DataSource dataSource) {  
        setDataSource(dataSource);  
        setFunction(true);  
        setSql(SQL);  
        declareParameter(new SqlOutParameter("date", Types.DATE));  
        compile();  
    }  
  
    public Date execute() {  
        // the 'sysdate' sproc has no input parameters, so an empty Map is supplied...  
        Map<String, Object> results = execute(new HashMap<String, Object>());  
        Date sysdate = (Date) results.get("date");  
        return sysdate;  
    }  
}
```

StoredProcedure

```
public class StoredProcedureDao {  
  
    private GetSysdateProcedure getSysdate;  
  
    @Autowired  
    public void init(DataSource dataSource) {  
        this.getSysdate = new GetSysdateProcedure(dataSource);  
    }  
  
    public Date getSysdate() {  
        return getSysdate.execute();  
    }  
}
```

August 15, 2017

Hibernate

What and why is Hibernate?

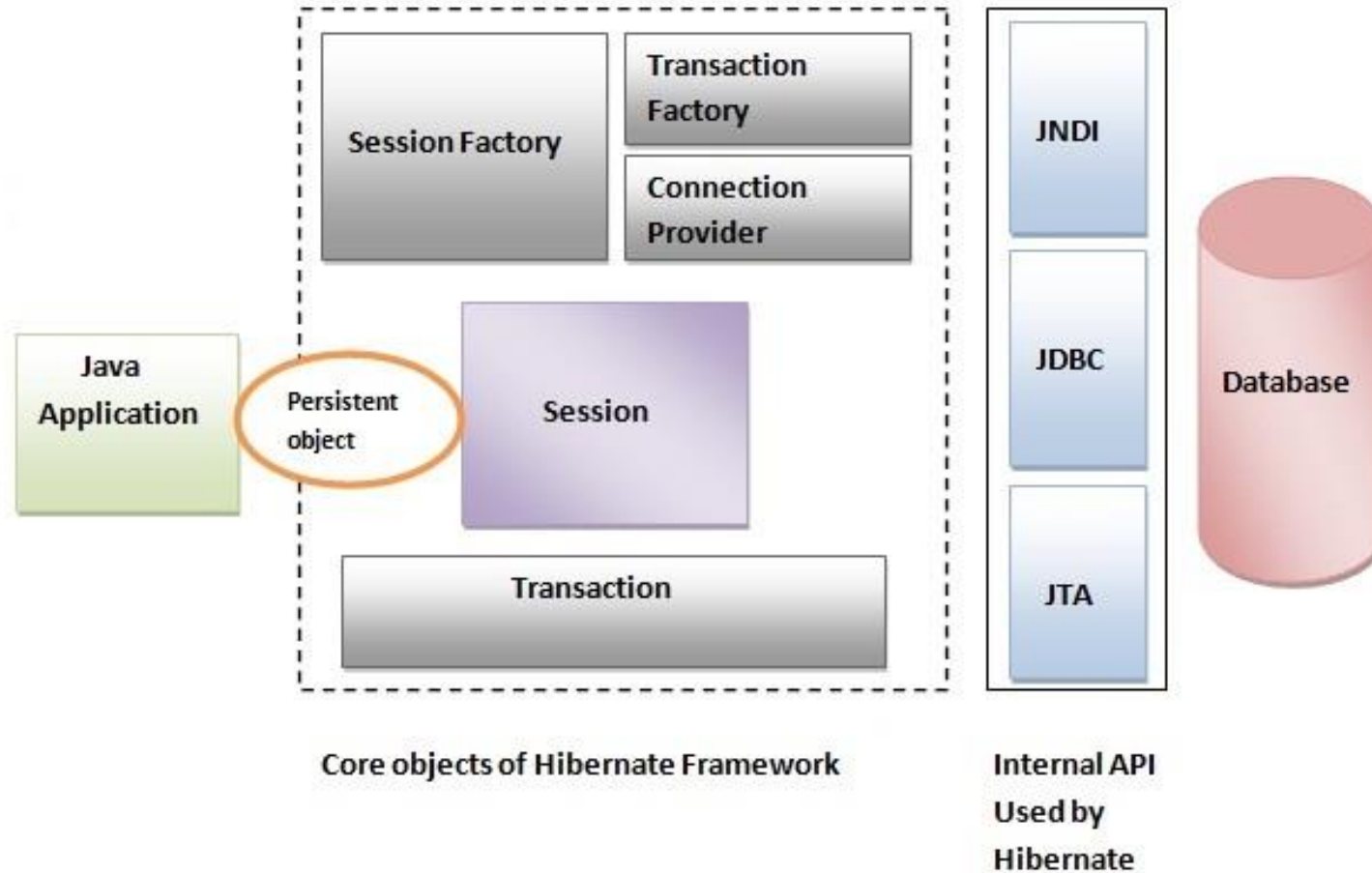
- Hibernate ORM (object-relational mapping) is an object-relational mapping framework for the Java language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions.
- Hibernate is database independent
- Object-relational mapping, you will map a database table with java object called "Entity".
- Caching mechanism
- Supports **Lazy loading**

Advantages of hibernates

- Supports Inheritance, Collections (List,Set,Map)
- Supports relationships like One-To-Many,One-To-One, Many-To-Many, Many-To-One
- HQL (Hibernate Query Language) - database independent commands
- Simplifies complex join
- Supports caching mechanism - Fast performance



Hibernate Architecture





Hibernate architecture overview

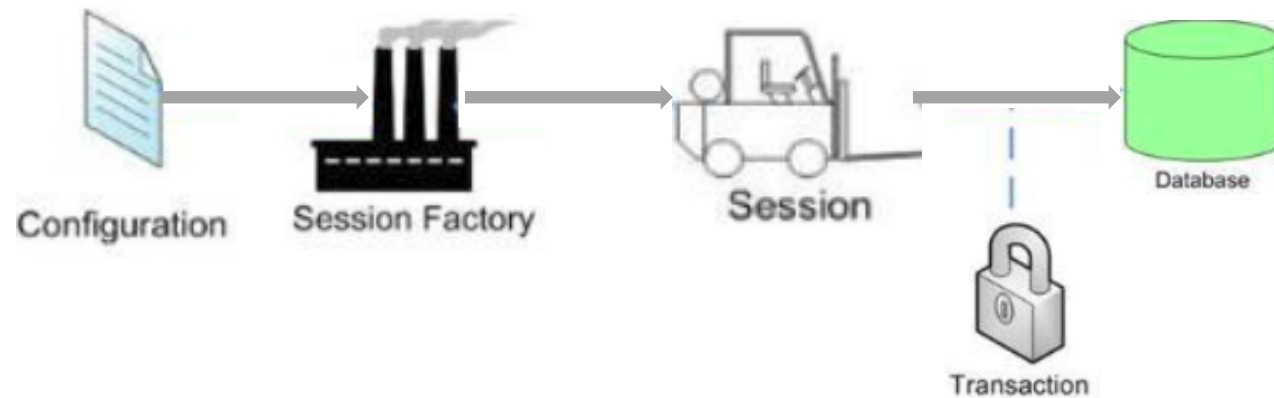
- Hibernate architecture has three main components:
 - **Connection Management**
 - Provides efficient management of the database connections.
 - **Transaction management**
 - Provides the ability to the user to execute more than one database statements at a time.
 - **Object relational mapping**
 - Is a technique of mapping the data representation from an object model to a relational data model.

Hibernate main classes and interface API

- The main Hibernate API are given below:
 - org.hibernate.Hibernate
 - org.hibernate.cfg.Configuration
 - org.hibernate.SessionFactory
 - org.hibernate.Session
 - org.hibernate.Transaction
 - org.hibernate.Criteria
 - org.hibernate.ScrollableResults
 - org.hibernate.expression.Expression
 - org.hibernate.Query
 - org.hibernate.expression.Order

Hibernate main classes and interface API

- Flow of Hibernate application

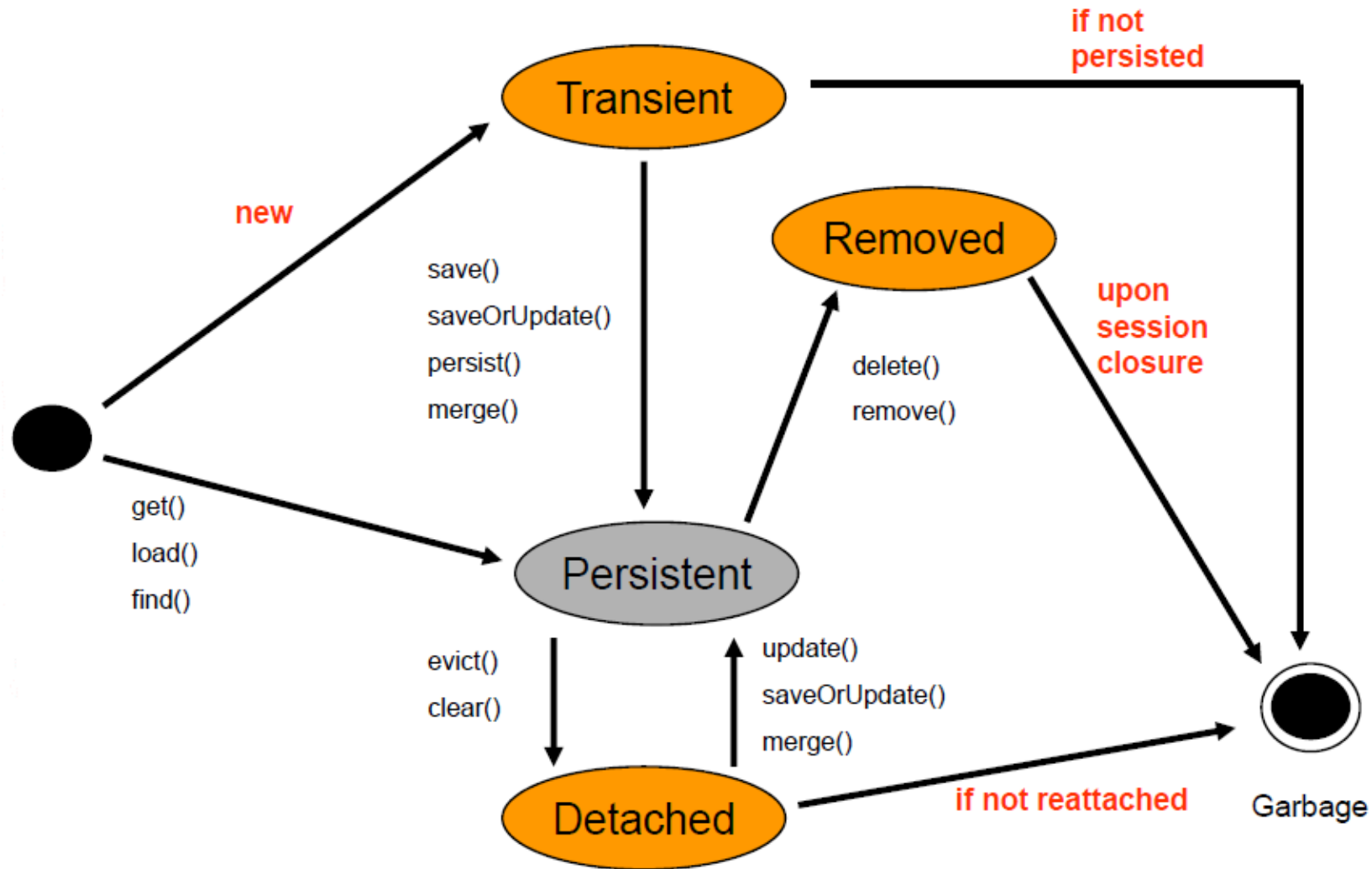


Working with Session interface

- Some important methods of Session interface:

Method	Description
connection()	Get the JDBC connection of this Session
contains(Object object)	Check if the instance is associated with the Session
merge(Object object)	Copy the state of the given object to the persistent object with the same identifier
save(Object object)	Persist the give transient instance, first assigning generated identifier
update(Object object)	Update the persistent instance with the given detached instance
delete(Object object)	Remove the persistent instance from data store
createSQLQuery(String query)	Create a new instance of SQLQuery for the given SQL query string
disconnect()	Disconnect the Session from the current JDBC connection
getTransaction()	Get the Transaction instance associated with this Session

Entity states





Hibernate Object Relational mapping

- Why Object Relational Mapping?
- Ways to map
- Types of mappings
- Inheritance
- Annotation mappings
- Hibernate data types



Hibernate data types

- Support all Java primitives and many JDK classes.
- Hibernate supports user-defined custom types.
 - int, long, String, java.io.Serializable, java.util.Calendar,..... (Java Types)
 - Collection, List, ArrayList,.... (Java Collection)
 - Personel, PlayerInfo,.... (Custom Types)



Querying in Hibernate

- Hibernate Query Language
- The Criteria Query API
- Native SQL

Why to Integrate Hibernate with Spring

- Spring Integrates very well with Hibernate. If someone asks why do we need to integrate hibernate in Spring? Yes, there are benefits.
- The very first benefit is the Spring framework itself. The IoC container makes configuring data sources, transaction managers, and DAOs easy.
- It manages the Hibernate SessionFactory as a singleton – a small but surprisingly annoying task that must be implemented manually when using Hibernate alone.
- It offers a transaction system of its own, which is aspectoriented and thus configurable, either through Spring AOP or Java-5 annotations. Either of these are generally much easier than working with Hibernate's transaction API.
- Transaction management becomes nearly invisible for many applications, and where it's visible, it's still pretty easy.
- You integrate more easily with other standards and frameworks.



Integrating Hibernate with Spring

- HibernateTemplate (no prefer)
- HibernateDaoSupport
- SessionFactory Injection



Integrating Hibernate with Spring - HibernateTemplate

- Step 1: Set Hibernate Libraries in classpath.
- Step 2: Declare a bean in Spring Config file for Hibernate Session Factory.
- Step 3: Inject session factory into Hibernate template.
- Step 4: Inject hibernate template into DAO classes.
- Step 5: Define the property HibernateTemplate in each DAO classes.
- Step 6: Use hibernate Queries through Hibernate Template object in DAO.

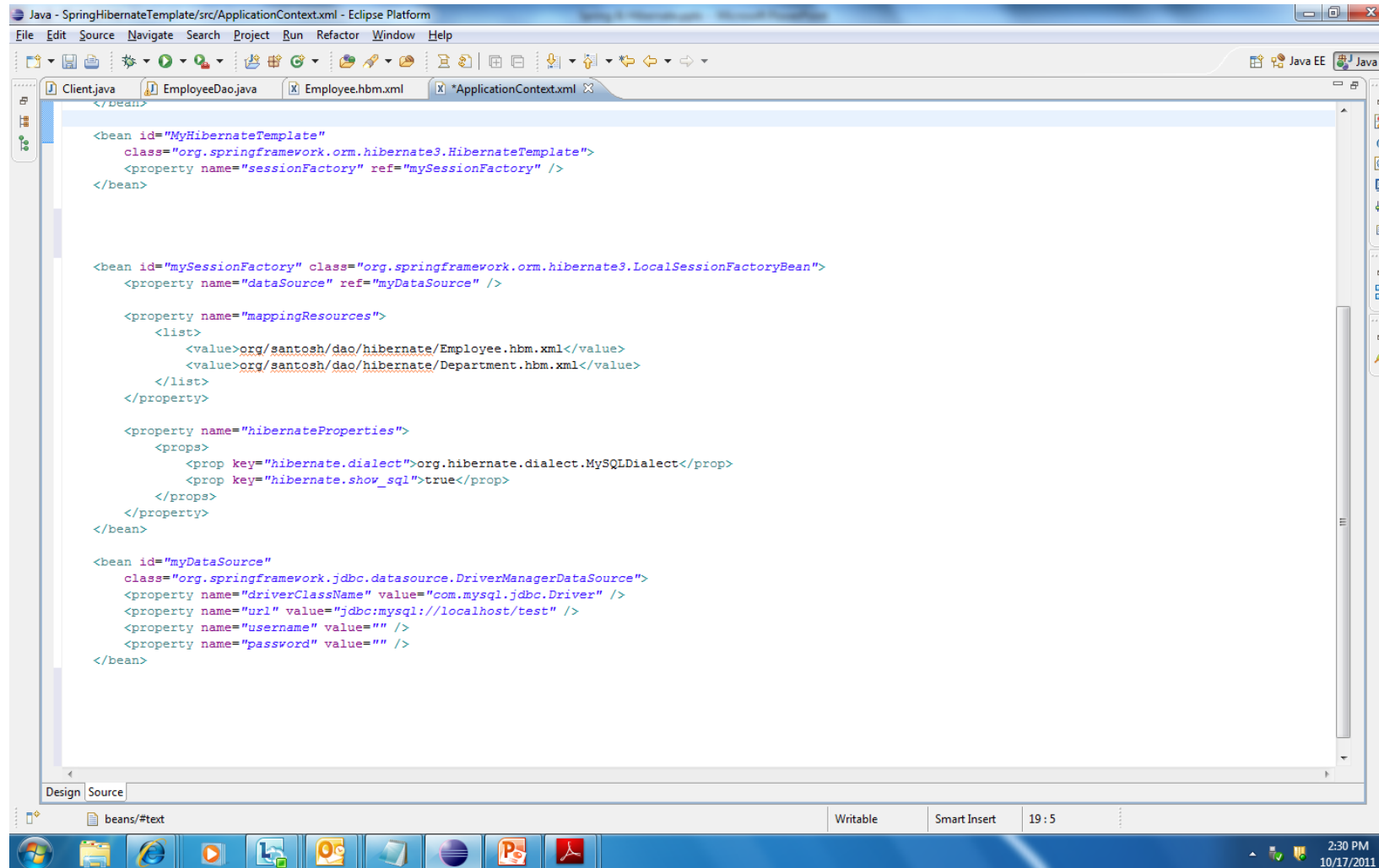
Integrating Hibernate with Spring – HibernateTemplate

- Step 1: Set Hibernate Libraries in classpath.
- To integrate Hibernate with Spring, you need the hibernate libraries along with Spring.
- So the first step should be downloading all the necessary library jars for Hibernate and set those jars into the project classpath just like you already have set the Spring libraries..

Integrating Hibernate with Spring – HibernateTemplate

- Step 2: Declare a bean in Spring Config file for Hibernate Session Factory
- A typical Hibernate application uses the Hibernate Libraries, configures its SessionFactory using a properties file or an XML file, use hibernate session etc. Now let's discuss the steps we must follow while integrating Hibernate with Spring.
- A typical Hibernate application configures its SessionFactory using a properties file or an XML file.
- First, we start treating that session factory as a Spring bean.
- Declare it as a Spring <bean> and instantiate it using a Spring ApplicationContext.
- Configure it using Spring <property>s, and this removes the need for a hibernate.cfg.xml or hibernate.properties file.
- Spring dependency injection – and possibly autowiring – make short work of this sort of configuration task.
- Hibernate object/relational mapping files are included as usual.

Integrating Hibernate with Spring – HibernateTemplate



```
</bean>

<bean id="MyHibernateTemplate"
      class="org.springframework.orm.hibernate3.HibernateTemplate">
  <property name="sessionFactory" ref="mySessionFactory" />
</bean>

<bean id="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="myDataSource" />

  <property name="mappingResources">
    <list>
      <value>org/santosh/dao/hibernate/Employee.hbm.xml</value>
      <value>org/santosh/dao/hibernate/Department.hbm.xml</value>
    </list>
  </property>

  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
      <prop key="hibernate.show_sql">true</prop>
    </props>
  </property>
</bean>

<bean id="myDataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost/test" />
  <property name="username" value="" />
  <property name="password" value="" />
</bean>
```


Integrating Hibernate with Spring – HibernateTemplate

- Step 3: Inject session factory into Hibernate template

```
<bean id="MyHibernateTemplate"  
      class="org.springframework.orm.hibernate3.HibernateTemplate">  
    <property name="sessionFactory" ref="mySessionFactory" />  
</bean>
```

- Step 4: Inject hibernate template into DAO classes

```
<bean id="employeeDao" class="org.santosh.dao.EmployeeDao">  
    <property name="hibernateTemplate" ref="MyHibernateTemplate"></property>  
</bean>  
<bean id="departmentDao" class="org.santosh.dao.DepartmentDao">  
    <property name="hibernateTemplate" ref="MyHibernateTemplate"></property>  
</bean>
```

Integrating Hibernate with Spring – HibernateTemplate

- Step 5: Define the property HibernateTemplate in each DAO classes

```
public class EmployeeDao {

    private HibernateTemplate hibernateTemplate;

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

    public HibernateTemplate getHibernateTemplate() {
        return hibernateTemplate;
    }

    public Employee getEmployeeById(long id) {
        return getHibernateTemplate().get(Employee.class, id);
    }

    public void addEmployee(Employee emp) {
        getHibernateTemplate().save(emp);
        System.out.println("1 record inserted...");
    }

    public void deleteEmployee(int employeeId) {
        getHibernateTemplate().delete(new Employee(employeeId));
        System.out.println("1 record(s) deleted...");
    }

}
```

Integrating Hibernate with Spring – HibernateTemplate

- Step 6: Use hibernate Queries through Hibernate Template object in DAO.

```
private HibernateTemplate hibernateTemplate;

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

public HibernateTemplate getHibernateTemplate() {
    return hibernateTemplate;
}

public Employee getEmployeeById(long id) {
    return getHibernateTemplate().get(Employee.class, id);
}

public void addEmployee(Employee emp) {
    getHibernateTemplate().save(emp);
    System.out.println("1 record inserted...");
}

public void deleteEmployee(int employeeId) {
    getHibernateTemplate().delete(new Employee(employeeId));
    System.out.println("1 record(s) deleted...");
}
}
```

Integrating Hibernate with Spring – HibernateDaoSupport

- So far, the configuration of DAO class (e.g. EmployeeDao or DepartmentDao) involves four beans.
- The data source is wired into the session factory bean through LocalSessionFactoryBean
- The session factory bean is wired into the HibernateTemplate.
- Finally, the HibernateTemplate is wired into DAO (e.g. EmployeeDao or DepartmentDao), where it is used to access the database.

Integrating Hibernate with Spring – HibernateDaoSupport

- To simplify things slightly, 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.
- So the only thing you
- Extend the class HibernateDaoSupport in each of the DAO class.
- Remove the property HibernateTemplate as it is already provided in HibernateDaoSupport class.

Integrating Hibernate with Spring – HibernateDaoSupport

```
package org.santosh.dao;

import org.santosh.vo.Employee;

public class EmployeeDao extends HibernateDaoSupport{

    /*private HibernateTemplate hibernateTemplate;

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

    public HibernateTemplate getHibernateTemplate() {
        return hibernateTemplate;
    }*/

    public Employee getEmployeeById(long id) {
        return getHibernateTemplate().get(Employee.class, id);
    }

    public void addEmployee(Employee emp) {
        getHibernateTemplate().save(emp);
        System.out.println("1 record inserted...");
    }

    public void deleteEmployee(int employeeId) {
        getHibernateTemplate().delete(new Employee(employeeId));
        System.out.println("1 record(s) deleted...");
    }

}
```

Integrating Hibernate with Spring – contextual sessions

- To implement the contextual session in Hibernate 3, you need to inject sessionFactory in place of HibernateTemplate in Spring configuration file.

```
<bean id="employeeDao" class="org.santosh.dao.EmployeeDao">
    <property name="sessionFactory" ref="mySessionFactory"></property>
</bean>
<bean id="departmentDao" class="org.santosh.dao.DepartmentDao">
    <property name="sessionFactory" ref="mySessionFactory"></property>
</bean>

<bean id="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource" />

    <property name="mappingResources">
        <list>
            <value>org/santosh/dao/hibernate/Employee.hbm.xml</value>
            <value>org/santosh/dao/hibernate/Department.hbm.xml</value>
        </list>
    </property>
```

Integrating Hibernate with Spring – contextual sessions

```
package org.santosh.dao;

import org.hibernate.Session;

public class EmployeeDao{

    private SessionFactory sessionFactory;

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

    public SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public Employee getEmployeeById(long id) {
        return (Employee) sessionFactory.openSession().get(Employee.class, id);
    }

    public void addEmployee(Employee emp) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();
        session.save(emp);
        tx.commit();
        System.out.println("1 record inserted...");
    }

    public void deleteEmployee(int employeeId) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();
        session.delete(new Employee(employeeId));
        tx.commit();
        System.out.println("1 record(s) deleted...");
    }
}
```


Questions & Answer



Thank You!

Revision History

Date	Version	Description	Updated by	Reviewed and Approved By
11/11/2015	1.0	Initial Document	Kien Tran	



BUSINESS SOLUTIONS
TECHNOLOGY
OUTSOURCING