Spring DAO Support

Agenda

1. Spring DAO Support
2. Data Access using JDBC
3. Spring ORM

Objectives

- Introduction
- Exception hierarchy
- Dao support packages
- Dao Support classes

Spring DAO Support

Introduction

Introduction

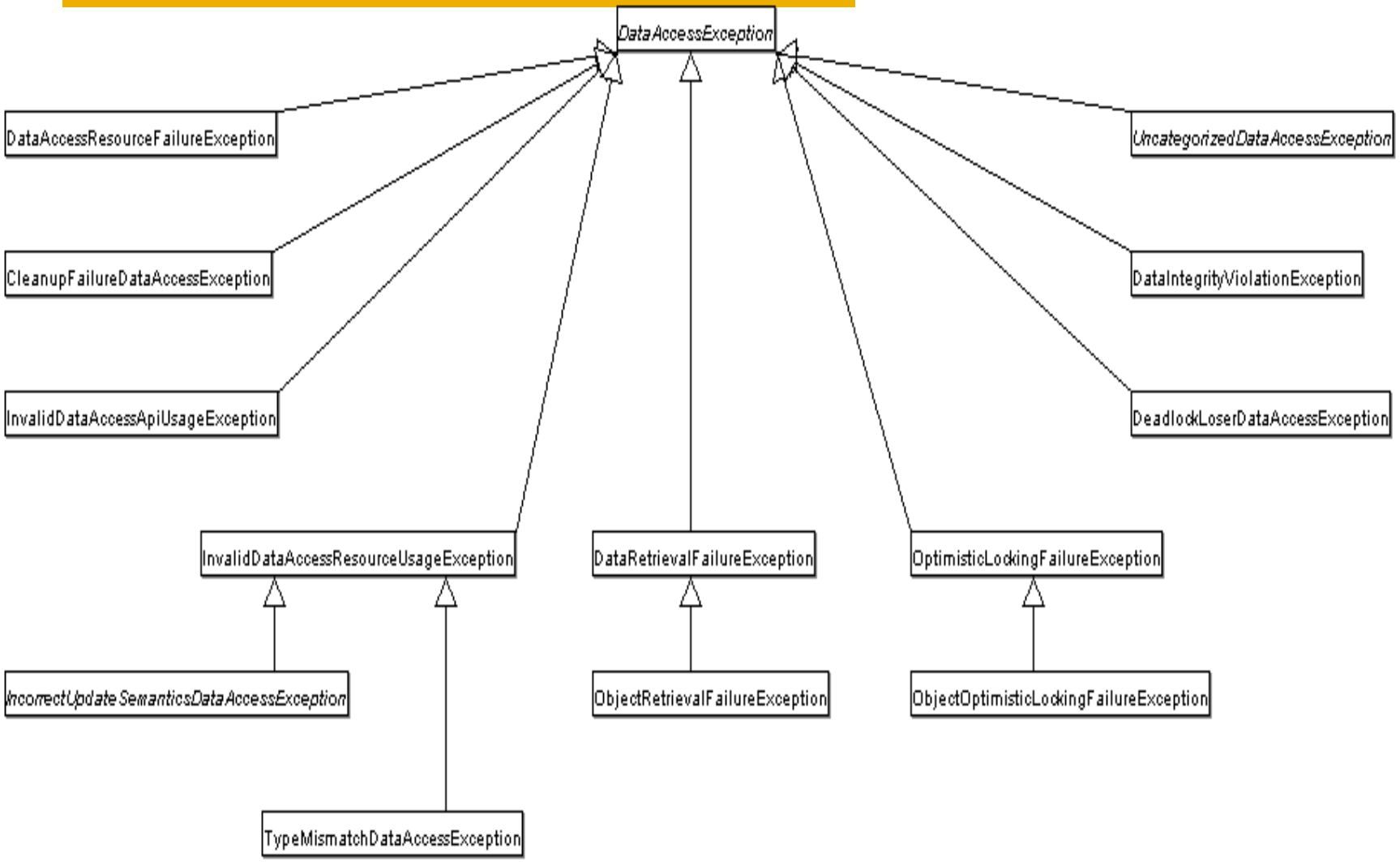
- The Spring Data Access Object (DAO) support makes it easy to work with data access technologies like JDBC, Hibernate or JDO in a standardized way

- Makes switching between databases easy and simple

- Code without Worrying about catching exceptions
  - Spring will do it for you!!!

Exception Hierarchy

Consistent exception hierarchy



Dao Support classes

Consistent abstract classes for DAO support

- Spring provides a set of abstract DAO classes that one can extended.
- These classes make data access easier with technologies like JDBC, Hibernate and JDO in consistent way.
- These abstract classes have methods for providing the data source and any other configuration settings that are specific to the technology one currently is using

Dao support classes

- JdbcDaoSupport
- HibernateDaoSupport
- JdoDaoSupport
- JpaDaoSupport

JdbcDaoSupport

- super class for JDBC data access objects.
- Requires a DataSource to be provided
- This class provides a JdbcTemplate instance initialized from the supplied DataSource to subclasses.

HibernateDaoSupport

- super class for Hibernate data access objects
- Requires a SessionFactory to be provided;
- This class provides a HibernateTemplate instance initialized from the supplied SessionFactory to subclasses
- Can alternatively be initialized directly via a HibernateTemplate, to reuse the latter's settings like SessionFactory, flush mode, exception translator, etc.

JdoDaoSupport

- super class for JDBC data access objects
- Requires a PersistenceManagerFactory to be provided
- This class provides a JdoTemplate instance initialized from the supplied PersistenceManagerFactory to subclasses.

JpaDaoSupport

- Super class for JPA data access objects
- Requires a EntityManagerFactory to be provided
- This class provides a JpaTemplate instance initialized from the supplied EntityManagerFactory to subclasses

Data access using JDBC

- In accessing the database Normally we write the code in the following way
  - Define connection parameters
  - Open the connection
  - *Specify the statement*
  - 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

Data access using JDBC (2)

- Spring Framework relaxes a developer from writing numerous jdbc codes lines
- A developer needs to write only the code to
  - Specify the statement
  - Do the work for each iteration
- Spring takes care of all the grungy, low-level details that can make JDBC such a tedious API to develop against

Using the JDBC Core classes to control basic JDBC processing and error handling

JdbcTemplate

- Central class in the JDBC core package
- It simplifies the use of JDBC since it handles the creation and release of resources
  - This helps to avoid common errors such as forgetting to always close the connection
- It executes the core JDBC workflow
  - like statement creation and execution
  - leaving application code to provide SQL
  - and extract results

JdbcTemplate

- This class executes
  - SQL queries
  - update statements or stored procedure calls, imitating iteration over ResultSets and extraction of returned parameter values.
- It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the *org.springframework.dao* package

Jdbctemplate

- The code using this class only need to implement callback interfaces
  - PreparedStatementCreator
    - Creates a PreparedStatement given a connection
  - CallableStatementCreator interface
    - which creates callable statement
  - RowCallbackHandler interface
    - extracts values from each row of a ResultSet

NamedParameterJdbcTemplate

- Adds support for programming JDBC statements using named parameters
  - as opposed to programming JDBC statements using only classic placeholder ('?') arguments
- wraps a vanilla JdbcTemplate, and delegates to the wrapped JdbcTemplate to do much of its work

DataSource

Executing statements

- We need a little code alog with a Datasource and JdbcTemplate

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAStatement {

    private JdbcTemplate jt;
    private DataSource dataSource;

    public void doExecute() {
        jt = new JdbcTemplate(dataSource);
        jt.execute("create table mytable (id integer, name varchar(100))");
    }

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

Running Queries

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class RunAQuery {

    private JdbcTemplate jt;
    private DataSource dataSource;

    public int getCount() {
        jt = new JdbcTemplate(dataSource);
        int count = jt.queryForInt("select count(*) from mytable");
        return count;
    }

    public String getName() {
        jt = new JdbcTemplate(dataSource);
        String name = (String) jt.queryForObject("select name from mytable", String.class);
        return name;
    }

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

Updating the database

```
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {

    private JdbcTemplate template;

    public void setName(int id, String name) {
        template.update("update mytable set name = ? where id = ?", new Object[]
        {name, new Integer(id)});
    }

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

Controlling database connections

DataSourceUtils

- The DataSourceUtils class is a convenient and powerful helper class that provides static methods to obtain connections from JNDI and close connections if necessary
- It has support for thread-bound connections, for example for use with *DataSourceTransactionManager*

SmartDataSource

- The SmartDataSource interface is to be implemented by classes that can provide a connection to a relational database
- Extends the DataSource interface to allow classes using it to query whether or not the connection should be closed after a given operation
- This can sometimes be useful for efficiency, in the cases where one knows that one wants to reuse a connection.

AbstractDataSource

- This is an abstract base class for Spring's DataSource implementations
- This is the class one would extend if one was writing one's own DataSource implementation

SingleConnectionDataSource

- an implementation of the SmartDataSource interface that wraps a *single* Connection that is *not* closed after use
- This is not multi-threading capable
- This is primarily a test class, it enables easy testing of code outside an application server, in conjunction with a simple JNDI environment
- In contrast to DriverManagerDataSource, it reuses the same connection all the time, avoiding excessive creation of physical connections

DriverManagerDataSource

- an implementation of the SmartDataSource interface that configures a plain old JDBC Driver via bean properties, and returns a new connection every time

TransactionAwareDataSourceProxy

- TransactionAwareDataSourceProxy is a proxy for a target DataSource, which wraps that target DataSource to add awareness of Spring-managed transactions
- In this respect it is similar to a transactional JNDI DataSource as provided by a J2EE server

Modeling JDBC operations as Java objects

- Classes of org.springframework.jdbc.object package allows one to access the database in a more object-oriented manner
- one can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object
- One can also execute stored procedures and run update, delete and insert statements.

SqlQuery

- A reusable, threadsafe class that encapsulates an SQL query
- The SqlQuery class is rarely used directly
- The MappingSqlQuery subclass provides a much more convenient implementation for mapping rows to Java classes

MappingSqlQuery

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

MappingSqlQuery

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

An Example

```
private class CustomerMappingQuery extends MappingSqlQuery {

    public CustomerMappingQuery(DataSource ds) {
        super(ds, "SELECT id, name FROM customer WHERE id = ?");
        super.declareParameter(new SqlParameter("id", Types.INTEGER));
        compile();
    }

    public Object mapRow(ResultSet rs, int rowNumber) throws SQLException {
        Customer cust = new Customer();
        cust.setId((Integer) rs.getObject("id"));
        cust.setName(rs.getString("name"));
        return cust;
    }
}
```

An Example

```
public Customer getCustomer(Integer id) {
    CustomerMappingQuery custQry = new
      CustomerMappingQuery(dataSource);
    Object[] parms = new Object[1];
    parms[0] = id;
    List customers = custQry.execute(parms);
    if (customers.size() > 0) {
        return (Customer) customers.get(0);
    }
    else {
        return null;
    }
}
```

SqlUpdate

Spring ORM

Spring - Hibernate

SessionFactory setup in Spring Container

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

 <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost:3306/hibernate"/>
  <property name="username" value=""/>
  <property name="password" value=""/>
 </bean>

 <bean id="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="myDataSource"/>
  <property name="mappingResources">
   <list>
    <value>emp.hbm.xml</value>
   </list>
  </property>
  <property name="hibernateProperties">
   <props>
    <prop key="dialect">org.hibernate.dialect.MySQLDialect</prop>
    <prop key="hibernate.hbm2ddl.auto">update</prop>
   </props>

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

Configuring the DataSource

```
<beans>


  <bean id="myDataSource"
          class="org.apache.commons.dbcp.BasicDataSource" >

    <property name="driverClassName"
          value="com.mysql.jdbc.Driver"/>

    <property name="url"     value="jdbc:mysql://localhost:3306/hibernate"/>

    <property name="username" value=""/>

    <property name="password" value=""/>

  </bean>

  ..........

  </beans>
```

Configuring the SessionFactory

```
<bean id="mySessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
   <property name="dataSource" ref="myDataSource"/>
   <property name="mappingResources">
     <list>
       <value>emp.hbm.xml</value>
     </list>
   </property>
   <property name="hibernateProperties">
     <props>
       <prop key="dialect">org.hibernate.dialect.MySQLDialect</prop>
       <prop key="hibernate.hbm2ddl.auto">update</prop>
     </props>

   </property>
 </bean>
```

Configuring the HibernateTemplate

```
<bean id="hibernateTemplate"
      class="org.springframework.orm.hibernate3.HibernateTemplate">

      <property name="sessionFactory"><ref
      local="mySessionFactory"/></property>

      </bean>
```

DataSource from JNDI

- If the Datasource is located in JNDI then the Datasource Configuration is

```
<beans>

 <bean id="myDataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">

   <property name="jndiName"
         value="java:comp/env/jdbc/myds"/>

 </bean>


</beans>
```