5 Spring Data Access Framework

In order to simplify accessing relational databases from Java, JDBC (Java Database Connectivity) technology was introduced to define a set of standard APIs for all common SQL operations (SELECT, INSERT, UPDATE, DELETE, and so on) in a vendor-independent fashion. Spring JDBC Framework further simplifies this kind of operations by defining an abstract layer on top of the standard JDBC APIs. The core of the Spring JDBC Framework is that it provides a template based methods for different types of JDBC operations. In this chapter, I'll give you a brief introduction to Spring JDBC Framework by demonstrating how it is applied to SOBA. I will also show you how Spring Data Access Framework supports Hibernate in addition to JDBC.

5.1 Defining Spring DataSources

Using Spring JDBC Framework starts with defining a JDBC data source. As an example, SOBA has its JDBC data source defined in the soba-services.xml file, which is shown in Listing 5.1. First, note the datasource bean with its properties defined with entries like ${jdbc.driverClassName}, and so on. The values for these entries are read in from the jdbc.properties file located at /Web-INF/classes. Also note that every manager bean depends on one or more DAO (Data Access Object) beans, and every DAO bean depends on the datasource bean defined based on the external jdbc.properties file.

Listing 5.1 soba-services.xml

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
   http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
   <bean id="sobaConfig" class="com.perfmath.spring.soba.service.SobaConfig">
```
<property name="databaseVendor" value="mysql" />
</bean>
<bean id="perfBasicUtil" class="com.perfmath.spring.soba.util.PerfBasicUtil"
  init-method="createApfWriter" destroy-method="closeApfWriter">
  <property name="apfFileName" value="webapps/soba/logs/perfLogData.apf" />
  <property name="profilingOn" value="yes" />
</bean>
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
<bean id="activityManager"
  class="com.perfmath.spring.soba.service.SimpleActivityManager">
  <property name="activityDao" ref="activityDao" />
</bean>
<bean id="transferManager"
  class="com.perfmath.spring.soba.service.SimpleTransferManager">
  <property name="transferDao" ref="transferDao" />
  <property name="bankingTxDao" ref="bankingTxDao" />
</bean>
<bean id="activityDao"
  class="com.perfmath.spring.soba.model.dao.JdbcActivityDao">
  <property name="dataSource" ref="dataSource" />
</bean>
<bean id="customerManager"
  class="com.perfmath.spring.soba.service.SimpleCustomerManager">
  <property name="customerDao" ref="customerDao" />
</bean>
<bean id="customerDao"
  class="com.perfmath.spring.soba.model.dao.JdbcCustomerDao">
  <property name="dataSource" ref="dataSource" />
</bean>
<bean id="transferDao"
  class="com.perfmath.spring.soba.model.dao.JdbcTransferDao">
  <property name="dataSource" ref="dataSource" />
</bean>
<bean id="aclTest" class="com.perfmath.spring.soba.test.ACLTest">
  <property name="dataSource" ref="dataSource" />
  <property name="mutableAclService" ref="aclService" />
</bean>
CHAPTER 5: SPRING DATA ACCESS FRAMEWORK

```xml
<property name="platformTransactionManager" ref="transactionManager"/>
</bean>

<bean id="accountManager"
    class="com.perfmath.spring.soba.service.SimpleAccountManager">
    <property name="accountDao" ref="accountDao"/>
</bean>

<bean id="accountDao"
    class="com.perfmath.spring.soba.model.dao.JdbcAccountDao">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="loginUserManager"
    class="com.perfmath.spring.soba.service.SimpleLoginUserManager">
    <property name="loginUserDao" ref="loginUserDao"/>
</bean>

<bean id="loginUserDao"
    class="com.perfmath.spring.soba.model.dao.JdbcLoginUserDao">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="txManager"
    class="com.perfmath.spring.soba.service.SimpleTxManager">
    <property name="bankingTxDao" ref="bankingTxDao"/>
    <property name="accountManager" ref="accountManager"/>
    <!-- property name="mutableAclService" ref="aclService" -->
</bean>

<bean id="billPayManager"
    class="com.perfmath.spring.soba.service.SimpleBillPayManager">
    <property name="billPaymentDao" ref="billPaymentDao"/>
    <property name="aclTxManager" ref="aclTxManager"/>
    <!-- property name="mutableAclService" ref="aclService" -->
</bean>

<bean id="billPaymentDao"
    class="com.perfmath.spring.soba.model.dao.HibernateBillPaymentDao">
</bean>

<bean id="aclTxManager"
    class="com.perfmath.spring.soba.service.SimpleAclTxManager">
    <property name="aclBankingTxDao" ref="aclBankingTxDao"/>
    <property name="loginUserDao" ref="loginUserDao"/>
    <property name="accountDao" ref="accountDao"/>
    <property name="accountManager" ref="accountManager"/>
    <property name="mutableAclService" ref="aclService"/>
</bean>

<bean id="userDetailsService"
    class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="bankingTxDao"
    class="com.perfmath.spring.soba.model.dao.JdbcBankingTxDao">
</bean>
```
5.2 IMPLEMENTING DAO WITH JDBC

JDBC itself is a broad topic. In order to help you get familiar with it quickly, let’s use a real DAO class to illustrate it. My favorite DAO class with SOBA is BankingTxDao, and I’ll tell you why later.

For a particular domain object, for example, the BankingTx domain object, you need two Java classes: one for defining the interface, and the other for implementing the interface. For the BankingTx domain object, we have the BankingTxDao.java class and JDBCBankingTxDao.java class. Let’s first review the BankingTx domain object next.

5.2.1 Defining Domain Objects

The BankingTx domain object code in Java is shown in Listing 5.2 (it was originally named Transaction instead of BankingTx, which was okay with Oracle and MySQL, but I had to rename it because the word Transaction is a reserved keyword in SQL Server and SQL Server would not allow me to create a table named TRANSACTION). Notice those @XmlElement annotations? These are JAXB related and you can ignore them for now. JAXB stands for Java Architecture for XML Binding, which is a JAVA standard defining how Java objects are converted from and to XML. We’ll talk about it a little bit more later.

Listing 5.2 BankingTx.java
package com.perfmath.spring.soba.model.domain;

import java.io.Serializable;
import java.sql.Timestamp;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
import com.perfmath.spring.soba.model.domain.TimestampAdapter;

public class BankingTx implements Serializable {

    private long transactionId;
    private Timestamp transDate;
    private String type;
    private String initiator;
    private String description;
    private double amount;
    private double balance;
    private String accountId;
    private String status;

    @XmlElement(name="transactionId")
    public long getTransactionId() {
        return transactionId;
    }

    public long getId() {
        return transactionId;
    }

    public void setTransactionId(long transactionId) {
        this.transactionId = transactionId;
    }

    @XmlElement(name="transDate")
    @XmlJavaTypeAdapter(TimestampAdapter.class)
    public Timestamp getTransDate() {
        return transDate;
    }

    public void setTransDate(Timestamp transDate) {
        this.transDate = transDate;
    }

    @XmlElement(name="type")
    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    @XmlElement(name="initiator")

public String getInitiator() {
    return initiator;
}
public void setInitiator(String initiator) {
    this.initiator = initiator;
}

@XmlElement(name="description")
public String getDescription() {
    return description;
}
public void setDescription(String description) {
    this.description = description;
}

@XmlElement(name="amount")
public double getAmount() {
    return amount;
}
public void setAmount(double amount) {
    this.amount = amount;
}

@XmlElement(name="balance")
public double getBalance() {
    return balance;
}
public void setBalance(double balance) {
    this.balance = balance;
}

@XmlElement(name="accountId")
public String getAccountId() {
    return accountId;
}
public void setAccountId(String accountId) {
    this.accountId = accountId;
}

@XmlElement(name="status")
public String getStatus() {
    return status;
}
public void setStatus(String status) {
    this.status = status;
}

public String toString() {
    StringBuffer buffer = new StringBuffer();
    buffer.append("transactionId: " + transactionId + ";");
    buffer.append("accountId: " + accountId + ";");
    buffer.append("amount: " + amount);
}
This is a simple domain object despite those visually noisy XML annotations. So let’s move to the BankingTxDAO interface next.

### 5.2.2 Defining DAO Interface

The DAO interface corresponding to the domain object of BankingTx is exhibited in Listing 5.3. This is a very simple Java interface, and it’s very self-explanatory. After you take a quick look at it, let’s move to the next section which explains in detail how these interfaces are implemented in JDBC, or specifically the Spring JDBC Framework.

**Listing 5.3 BankingTxDao.java**

```java
package com.perfmath.spring.soba.model.dao;

import java.util.List;
import java.util.Map;
import com.perfmath.spring.soba.model.domain.BankingTx;

public interface BankingTxDao {
    public List<BankingTx> getTransactionList();
    public List<BankingTx> getTransactionList(String accountId);
    public void insert(BankingTx transaction);
    public void update(BankingTx transaction);
    public void delete(String transactionId);
    public BankingTx findByTransactionID(String transactionId);
    public void insertBatch(List<BankingTx> transactions);
    public List<Map<String, Object>> findAll();
    public String getAccountId(String transactionId);
    public int countAll();
}
```

### 5.3 IMPLEMENTING DAO WITH SPRING JDBC TEMPLATE

This section explains in detail how the BankingTxDAO interface defined in the preceding section is implemented with the two major Spring JDBC classes: JdbcTemplate and NamedParameterJdbcTemplate, against the domain object BankingTx defined in Listing 5.2. There are many variations with Spring JDBC, but I feel that these two classes should be sufficient most of the time. You should refer to Chapter 13 Data Access with JDBC of the Spring Reference Documentation 3.1 if you want to know every detail about this subject. Here, I’d like to present a succinct example to show how it works.

First, let’s show how the JDBCBankingTxDao.java code is implemented in Listing 5.4. Note the highlighted parts that we explain next following the code listing.
Listing 5.4 JDBCBankingTxDao.java

```java
package com.perfmath.spring.soba.model.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import java.util.Map;
import javax.sql.DataSource;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcDaoSupport;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSourceUtils;
import org.springframework.jdbc.core.simple.ParameterizedRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import com.perfmath.spring.soba.model.domain.BankingTx;

public class JdbcBankingTxDao implements BankingTxDao {
    private JdbcTemplate jdbcTemplate;
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    public List<BankingTx> getTransactionList() {
        List<BankingTx> txs = this.jdbcTemplate
                .query("SELECT TRANSACTION_ID, TRANS_DATE, TYPE, " + "INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS FROM BANKINGTX",
                    new TransactionMapper());
        return txs;
    }

    public List<BankingTx> getTransactionList(String accountId) {
        List<BankingTx> txs = this.jdbcTemplate
                .query("SELECT TRANSACTION_ID, TRANS_DATE, TYPE, " + "INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS FROM BANKINGTX WHERE ACCOUNT_ID = ?",
                    " ORDER BY TRANS_DATE DESC", new TransactionMapper(), accountId);
        return txs;
    }
}
```
public void insert(BankingTx transaction) {
    String sql = "INSERT INTO BANKINGTX (TRANSACTION_ID, TRANS_DATE, TYPE, "
    + "INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS)"
    + "VALUES (:transactionId, :transDate, :type, :initiator, "
    + ":description, :amount, :balance, :accountId, :status);"
    SqlParameterSource namedParameters = new
    BeanPropertySqlParameterSource( transaction);

    int count = this.namedParameterJdbcTemplate
    .update(sql, namedParameters);
}

public void insertBatch(List<BankingTx> transactions) {
    String sql = "INSERT INTO BANKINGTX (TRANSACTION_ID, TRANS_DATE, TYPE, "
    + "INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS )"
    + "VALUES (:transactionId, :transDate, :type, :initiator, "
    + ":description, :amount, :balance, :accountId, :status);"

    SqlParameterSource[] parameterSource =
    SqlParameterSourceUtils.createBatch(transactions.toArray());

    int count[] = this.namedParameterJdbcTemplate
    .batchUpdate(sql, parameterSource);
}

public BankingTx findByTransactionID(String transID) {
    String sql = "SELECT TRANSACTION_ID, TRANS_DATE, TYPE, "
    + " INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS "
    + " FROM BANKINGTX WHERE TRANSACTION_ID = '" + transID + "'";
    BankingTx trans = this.jdbcTemplate
    .queryForObject(sql,
    new TransactionMapper());

    return trans;
}

public void update(BankingTx tx) {
}

public void delete(String txId) {
    String sql = "DELETE BANKINGTX WHERE TRANSACTION_ID = ?";
int count = this.jdbcTemplate.update(sql, txId);
}

public List<Map<String, Object>> findAll() {
    String sql = "SELECT * FROM BANKINGTX";
    List<Map<String, Object>> trans = this.jdbcTemplate.queryForList(sql,
            new TransactionMapper());
    return trans;
}

public String getAccountId(String transID) {
    String sql = "SELECT ACCOUNT_ID FROM BANKINGTX WHERE TRANSACTION_ID = ?";
    String accountId = this.jdbcTemplate.queryForObject(sql, String.class,
            transID);
    return accountId;
}

public int countAll() {
    String sql = "SELECT COUNT(*) FROM BANKINGTX";
    int count = this.jdbcTemplate.queryForInt(sql);
    return count;
}

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

private static class TransactionMapper implements
    ParameterizedRowMapper<BankingTx> {
    public BankingTx mapRow(ResultSet rs, int rowNum) throws SQLException {
        BankingTx tx = new BankingTx();
        tx.setTransactionId(rs.getLong("TRANSACTION_ID"));
        tx.setTransDate(rs.getTimestamp("TRANS_DATE"));
        tx.setType(rs.getString("TYPE"));
        tx.setInitiator(rs.getString("INITIATOR"));
        tx.setDescription(rs.getString("DESCRIPTION"));
        tx.setAmount(rs.getDouble("AMOUNT"));
        tx.setBalance(rs.getDouble("BALANCE"));
        tx.setAccountId(rs.getString("ACCOUNT_ID"));
    }
}
Here is a generic pattern adopted in implementing the JDBCBankingTxDao class show in the above class:

- Add two variables of jdbcTemplate and namedParameterJdbcTemplate follows:
  ```java
  private JdbcTemplate jdbcTemplate;
  private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
  ```

- Add a setDataSource method as shown in the above listing. The datasource object is configured externally in soba-services.xml and you don’t have to worry about it here.

- Define a TransactionMapper class as shown in the above listing.

The rule of thumb is that you use the SQL operations from the JdbcTemplate class with SQLs that do not use named parameters, but you need to use NamedParameterJdbcTemplate class with SQLs that use named parameters. Otherwise, if use something like this.jdbcTemplate.update (sql, namedParameters) (for example, with the insert method), it would compile fine, but you would get the following error when it is actually executed, for example, with createTransaction calls in SOBA:

```java
java.sql.SQLException: Invalid argument value: java.io.NotSerializableException
com.mysql.jdbc.SQLException.createSQLException(SQLException.java:1055)
com.mysql.jdbc.SQLException.createSQLException(SQLException.java:956)
org.springframework.jdbc.core.StatementCreatorUtils.setValue(StatementCreatorUtils.java:351)
org.springframework.jdbc.core.StatementCreatorUtils.setParameterValueInternal(StatementCreatorUtils.java:216)
org.springframework.jdbc.core.StatementCreatorUtils.setParameterValue(StatementCreatorUtils.java:144)
org.springframework.jdbc.core.JdbcTemplate$2.doInPreparedStatement(JdbcTemplate.java:816)
org.springframework.jdbc.core.JdbcTemplate.execute(JdbcTemplate.java:587)
```
Therefore, it’s important to know when to use JdbcTemplate and when to use NamedParameterJdbcTemplate, as explained next.

5.3.1 JdbcTemplate.query

This method is for returning a result set of all or partial rows in your database from the table corresponding to your domain object. It is used to implement the two getTransactionList methods, one for all transactions, and the other for a specific account. Note that in the second method, a question mark “?” is used in the SQL query to hold the ACCOUNT_ID variable. If you have more variables, then you can use more question marks, but you need to make sure the sequence of the values provided after the second argument new TransactionMapper() matches. The return type is a Java List in this case.

5.3.2 JdbcTemplate.queryForObject

This method is applied to findByTransactionID and getAccountId. In the first case, the returned object is a BankingTx object, and in the second case, the returned object is a String representing an account id.

5.3.3 JdbcTemplate.queryForList

This method is applied to findAll, which returns all banking transactions as a Java List. Note that you need to specify new TransactionMapper() for the second argument of this queryForList call, which used to be BankingTx.class with the deprecated getSimpleJdbcTemplate method.

5.3.4 JdbcTemplate.queryForInt

This method is applied to countAll, which returns the number of rows of the domain object in the database.

5.3.5 NamedParameterJdbcTemplate.update

This method covers all SQL statements of INSERT, UPDATE and DELETE. JDBC does not have a method for each of these SQL operations. Instead, the update method applies to all these cases, and the concrete operation is resolved internally.

Note in the insert method, you see items preceded with “:” in the SQL statement instead of question marks. These are called named parameters, which is a feature provided with the Spring JDBC Framework. It is used in conjunction with SqlParameterSource, which automatically matches the passed-in object with the named parameters. Once again, if you use this.jdbcTemplate instead of namedParameterJdbcTemplate, you would get the
5.3.6 NamedParameterJdbcTemplate.batchUpdate

This method enables batch operations, for example, inserting/deleting/updating multiple rows in one batch to the database. Note that you need to create a SqlParameterSource array with SqlParameterSourceUtils.createBatch for the second argument of the batchUpdate method. Array processing is a common practice to improve the performance and scalability of a database-centric enterprise application.

Next, let's see how these JDBC DAOs are used by service beans to query or persist the model from and to the database.

5.4 SERVICE BEANS

In the beginning of this chapter, you might have noticed that many beans defined in the soba-services.xml file have the word Manager in their names. These are service beans that mostly depend on the DAO beans. In this section, we explore one of such examples.

The example we use is the BankingTx Manager bean. Its interface is defined in TxManager.java and its implementation is defined in SimpleTxManager.java. These two Java classes are exhibited in Listings 5.5 (a) and (b). The TxManager interface is self-explanatory and there is not much to explain. However, a few things worth to notice with the implementation class of SimpleTxManager.java:

- First, notice those annotations like @Transactional and @Secured. The annotation @Transactional indicates that all operations within that method should either all succeed or none, while the annotation @Secured indicates only the user who has one of these designated roles is allowed to execute that method.
- From each of the methods, it's easy to see how the BankingTxDao is used to carry out various database operations.
- At last, notice the createTransaction (BankingTx tx) method. There is a requirement that when a banking transaction is persisted into the BankingTx table in the SOBA database, that user’s account balance needs to be updated and so does the balance value of that banking transaction. This is a non-issue with Oracle and MySQL, both of which support BEFORE-TRIGGER (refer to Listing 2.12 about this trigger), but since SQL Server does not support BEFORE-TRIGGER, I could not find an easy workaround on SQL Server. Eventually, I have to make it work here by updating the account balance and then setting the new balance for the banking transaction before it is inserted into the database. Because of this interesting twist, I chose the banking transaction domain object as the example for this chapter.

Listing 5.5 (a) TxManager.java

```java
package com.perfmath.spring.soba.service;

import java.io.Serializable;
```
import java.util.List;
import com.perfmath.spring.soba.model.domain.BankingTx;

    public interface TxManager extends Serializable{
    public void createTransaction(BankingTx tx);
    public List<BankingTx> getTransactions();
    public List<BankingTx> getTransactions(String accountId);
    public void updateTransaction(BankingTx tx);
    public BankingTx findByTransactionID(String txId);
    public void deleteTransaction(String txId);
    }

Listing 5.5 (b) SimpleTxManager.java

    package com.perfmath.spring.soba.service;

    import java.util.List;
    import com.perfmath.spring.soba.model.dao.BankingTxDao;
    import com.perfmath.spring.soba.model.domain.BankingTx;
    import java.util.ArrayList;
    import java.util.LinkedHashMap;
    import java.util.List;
    import java.util.Map;
    import org.springframework.transaction.annotation.Transactional;
    import org.springframework.security.annotation.Secured;
    public class SimpleTxManager implements TxManager {
        private BankingTxDao bankingTxDao;
        private AccountManager accountManager;

        public List<BankingTx> getTransactions() {
            return bankingTxDao.getTransactionList();
        }

        public List<BankingTx> getTransactions(String accountId) {
            return bankingTxDao.getTransactionList(accountId);
        }

        public void updateTransaction(BankingTx tx) {
            bankingTxDao.update(tx);
        }

        @Transactional
        @Secured("ROLE_REP")
        public void deleteTransaction(String txId) {
            bankingTxDao.delete(txId);
        }
    }
@Transactional
@Secured("ROLE_CUST, ROLE_REP")
public void createTransaction(BankingTx tx) {
    // added due to no BEFORE-TRIGGER in sql server
    if (SobaConfig.getDatabaseVendor().equalsIgnoreCase("SQLServer")) {
        double balance = accountManager.updateAccountBalance
            (tx.getAmount(), tx.getAccountId());
        tx.setBalance(balance);
    }
    bankingTxDao.insert(tx);
}

public BankingTx findByTransactionID(String txId) {
    return bankingTxDao.findByTransactionID(txId);
}

public void setBankingTxDao(BankingTxDao bankingTxDao) {
    this.bankingTxDao = bankingTxDao;
}

public void setAccountManager(AccountManager accountManager) {
    this.accountManager = accountManager;
}

Next, let’s take a look at how Spring Data Access Framework works with Hibernate.

5.5 Hibernate Data Access Method

First, there is no barrier with using an ORM framework like Hibernate with the Spring
MVC framework, thanks to the architectural pluggability of Spring. Secondly, in general,
an ORM framework is favored over JDBC for accessing data stored in a database. Let’s
first explore what benefits an ORM framework like Hibernate provides over JDBC.

5.5.1 Benefits of Using Hibernate

As one of the most popular ORM frameworks, Hibernate provides the following benefits:

- Hibernate works by persisting data according to the mapping metadata defined in a
  mapping file. A mapping file defines mappings between a Java class and a database
  table by specifying how the attributes of the Java class and the columns of the table
  map to each other. This provides a natural bridge between the object-oriented
  programming model and the relational database model.

- Hibernate generates SQL statements at runtime, and therefore its implementation is
  database vendor neutral. This is especially important if your application has to
  support multiple database platforms from all major vendors.

- Hibernate goes beyond the basic ORM functionality. It supports additional features
  like caching, cascading, and lazy loading which may help enhance the performance
  and scalability of your application.
- Hibernate defines a powerful query language of its own which is called HQL (Hibernate Query Language). It’s very easy to write queries with the HQL.

Next, let’s see how metadata mapping works with Hibernate.

### 5.5.2 Hibernate Mapping (HBM)

*Hibernate Mapping* (HBM) between a Java object and a database table is straightforward. For example, with the `BillPayment.java` class shown in Listing 5.6, metadata mapping is achieved with a mapping file as shown in Listing 5.7. Note that this mapping file needs to be placed in the class target directory of the java class it is mapped to, or in this case, together with the `BillPayment.class` file rather than the source file.

One needs to pay attention to how the ID field is mapped in Listing 5.7. First, this field is mandatory that both the Java class and the database table must have. Secondly, its type must be `long`. Finally, note how a generator is defined with a property of `class = "..."`. The values of this property can have include: `assigned`, `increment`, `identity`, `sequence`, `hilo`, `seqhilo`, `uuid`, `native`, `select`, `foreign`, etc. Detailed discussion on what each setting means is beyond the scope of this text, except that the setting of `assigned` chosen in SOBA means that it’s assigned by the application.

#### Listing 5.6 BillPayment.java

```java
package com.perfmath.spring.soba.model.domain;
import java.io.Serializable;
import java.sql.Timestamp;
public class BillPayment implements Serializable {
    private Long id = null;
    private String accountId;
    private String description;
    private double amount;
    private String fromAccount;
    private String biller;
    private String address;
    private String city;
    private String state;
    private String zipcode;
    private String status;
    private Timestamp scheduleDate;
    private Timestamp sendDate;
    public BillPayment() {
    }
    // All getters and setter methods have been removed to save space
    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append(" Id: " + id + ";");
        buffer.append(" accountId: " + accountId + ";");
        buffer.append(" description: " + description);
    }
```

buffer.append(" amount: "+ amount);
buffer.append(" fromAccount: "+ fromAccount);
buffer.append(" biller: "+ biller);
buffer.append(" address: "+ address);
buffer.append(" city: "+ city);
buffer.append(" state: "+ state);
buffer.append(" zipcode: "+ zipcode);
buffer.append(" status: "+ status);
buffer.append(" scheduleDate: "+ scheduleDate);
buffer.append(" sendDate: "+ sendDate);
return buffer.toString();
}
}

Listing 5.7 BillPayment.hbm.xml

```xml
<hibernate-mapping package="com.perfmath.spring.soba.model.domain">
  <class name="BillPayment" table="BILL_PAYMENT">
    <id name="id" type="long" column="ID">
      <generator class="assigned">
        <!-- param name="sequence">BLL_PYMNT_SEQ</param -->
      </generator>
    </id>
    <property name="fromAccount" type="string">
      <column name="FROM_ACCOUNT" length="9" not-null="true" />
    </property>
    <property name="accountId" type="string">
      <column name="ACCOUNT_ID" length="9" not-null="true" />
    </property>
    <property name="description" type="string">
      <column name="DESCRIPTION" length="500" not-null="true" />
    </property>
    <property name="biller" type="string">
      <column name="BILLER" length="25" not-null="true" />
    </property>
    <property name="address" type="string">
      <column name="ADDRESS" length="50" not-null="true" />
    </property>
    <property name="city" type="string">
      <column name="CITY" length="25" not-null="true" />
    </property>
    <property name="state" type="string">
      <column name="STATE" length="2" not-null="true" />
    </property>
  </class>
</hibernate-mapping>
```
<property name="zipcode" type="string">
  <column name="ZIPCODE" length="10" not-null="true" />
</property>
<property name="status" type="string" column="STATUS" />
<property name="amount" type="double" column="AMOUNT" />
<property name="scheduleDate" type="date" column="SCHEDULE_DATE" />
<property name="sendDate" type="date" column="SEND_DATE" />
</class>
</hibernate-mapping>

Next, let’s take a look at how Hibernate is configured to work with a database system.

5.5.3 Hibernate Connectivity Configuration

Hibernate connectivity with a database platform is defined in a file named hibernate.cfg.xml file as shown in Listing 5.8 below. Hibernate looks for this file by default to establish connections with the configured database. Note that although Listing 5.8 is for MySQL, you can find the versions for Oracle and SQL Server from the SOBA download.

In this Hibernate configuration file, note the last line referring to the metadata mapping file described in the previous section. It tells a Hibernate engine where to find the metadata mapping file, which points to where the mapped object resides at the lowest level. The hibernate configuration file hibernate.cfg.xml must be placed in the classpath root directory, which is WEB-INF/classes directory.

Given what we have covered in Chapter 2 and Appendices B through D, the database connectivity configuration part in Listing 5.8 should be self-explanatory. The last few lines specific to Hibernate itself are denoted as follows:

- **The property hibernate.show_sql = true.** This entry specifies that all SQLs generated by Hibernate at runtime will be output to the console. This is a convenient feature for debugging purposes. However, it should be set to false in production due to performance concerns.

- **The property hibernate.hbm2ddl.auto = update.** This entry specifies the intention for creating the database schema on deploy if it doesn’t exist. It has many other possible settings and which one is most appropriate should be determined with the tests conducted with your application.

- **The property current_session_context_class = thread.** This entry specifies the scope of the current session. Valid values include jta, thread, managed, and so on. The setting of thread specified here limits the session context to the thread level.

Note that prior to Hibernate 3.6, the dtd file specified in the beginning of Listing 5.8 was http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd, which must be upgraded to the new dtd http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd for Hibernate 3.6. However, do not make this change in Listing 5.6 for BillPayment.hbm.xml; otherwise, you would get an error of “The content of element type “property” must match “null”” with each property element.
Since this is not a text about Hibernate, please consult more in-depth documentations to learn more about various Hibernate settings. Next, we discuss how a Hibernate DAO class is implemented.

### 5.5.4 Data Access Methods and Transaction Management for Hibernate

In general, data can be persisted to the backend database with one of the three mechanisms: (1) JDBC, (2) Hibernate, and (3) JPA (Java Persistence Architecture). Table 5.1 lists the resource, resource factory and exception implementations with each mechanism. It is seen that Hibernate uses the concepts of `Session` and `SessionFactory` to persist data, JDBC uses `Connection` and `DataSource` to persist data, whereas JPA uses `EntityManager` and `EntityManagerFactory` to persist data. The ramifications implied here are that there might be a variety of methods for Hibernate to persist data to the backend data store, as we will see next.

Table 5.1 Data Persistence Mechanisms

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Resource</th>
<th>ResourceFactory</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>Connection</td>
<td>DataSource</td>
<td>SQLException</td>
</tr>
<tr>
<td>Hibernate</td>
<td>Session</td>
<td>SessionFactory</td>
<td>HibernateException</td>
</tr>
<tr>
<td>JPA</td>
<td>EntityManager</td>
<td>EntityManagerFactory</td>
<td>PersistenceException</td>
</tr>
</tbody>
</table>

The other variable associated with persisting data is how a transaction is managed. While EJB supports bean-managed transaction (BMT) and container-managed transaction (CMT).
(CMT), Spring supports *programmatic transaction management (PTM)* and *declarative transaction management (DTM)*. The choice between PTM and DTM is leveraged by a trade-off between flexibility and complexity. DTM is simpler but less flexible than PTM.

Next, let’s see a Hibernate data persisting implementation method based on a POJO domain object with HBM.

**POJO Domain Object with HBM**

Listings 5.9 (a) and (b) illustrate how a Hibernate DAO is implemented for the *BillPayment* domain object with PTM. The Java class *HibernateBillPaymentDao* implements the interface defined in *BillPaymentDao.java*. Some unique features with this Hibernate DAO method include:

- The Hibernate DAO depends on a Hibernate *SessionFactory* object for all its operations. A *session* needs to be opened and closed within each operation.
- Each operation is managed as a transaction by calling the transactional begin and commit methods directly. This is what we called *PTM*, which allows a programmer to decide when to begin, commit or roll back a transaction precisely.
- Note the use of the session method `saveOrUpdate` with the *store* operation. The rule with this method is that it would perform an INSERT if the row to be inserted does not exist in the database or an UPDATE if the row exists.
- This Hibernate DAO implementation uses a POJO domain object (*BillPayment.java* defined in Listing 5.6), an *HBM* xml file (*BillPayment.hbm.xml* defined in Listing 5.7), a Hibernate configuration file (*hibernate.cfg.xml* defined in Listing 5.8), and PTM identifiable in *HibernateBillPaymentDao.java* defined in Listing 5.9 (b). Next, we’ll see how HBM can be eliminated by annotating the domain object class *BillPayment.java*.

**Listing 5.9 (a) BillPaymentDao.java**

```java
package com.perfmath.spring.soba.model.dao;
import java.util.List;
import com.perfmath.spring.soba.model.domain.BillPayment;
public interface BillPaymentDao {
    public void store(BillPayment billPayment);
    public void delete(String id);
    public BillPayment findById(String id);
    public List<BillPayment> findAll();
}
```

**Listing 5.9 (b) HibernateBillPaymentDao.java**

```java
package com.perfmath.spring.soba.model.dao;
import com.perfmath.spring.soba.model.BillPaymentDao;
import com.perfmath.spring.soba.model.domain.BillPayment;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
```
```java
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import java.util.List;

public class HibernateBillPaymentDao implements BillPaymentDao {
    private SessionFactory sessionFactory;
    public HibernateBillPaymentDao() {
        Configuration configuration = new Configuration().configure();
        sessionFactory = configuration.buildSessionFactory();
    }

    public void store(BillPayment billPayment) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.getTransaction();
        try {
            tx.begin();
            session.saveOrUpdate(billPayment);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        } finally {
            session.close();
        }
    }

    public void delete(String id) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.getTransaction();
        try {
            tx.begin();
            BillPayment billPayment = (BillPayment) sessionFactory
                    .getCurrentSession().get(BillPayment.class, id);
            session.delete(billPayment);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        } finally {
            session.close();
        }
    }

    public BillPayment findById(String id) {
        Session session = sessionFactory.openSession();
        try {
            return (BillPayment) sessionFactory.getCurrentSession().get(
                    BillPayment.class, id);
        } finally {
            session.close();
        }
    }
}
```
public List<BillPayment> findAll() {
    Session session = sessionFactory.openSession();
    try {
        Query query = sessionFactory.getCurrentSession().createQuery(
            "from Bill_Payment");
        return query.list();
    } finally {
        session.close();
    }
}

**JPA Annotated Domain Object without HBM**

Listing 5.10 illustrates the Java BillPayment class with JPA annotations (note that you need to manually add the hibernate-jpa-2.0-api-0.1.1.Final.jar file to your project library path if you use ANT to build SOBA). To save space, all setter methods have been excluded. This mechanism eliminates the need for an HBM file as was used in the previous case. In addition, in the hibernate.cfg.xml file, one change is required: The item in the <mapping> element should be changed from resource="com/perfmath/spring/soba/model/domain/BillPayment.hbm.xml" (see Listing 5.8) to class="com.perfmath.spring.soba.model.domain.BillPayment". How each attribute is annotated is self-explanatory so that we would not explain further.

**Listing 5.10 BillPayment.java (JPA-annotated)**

```java
package com.perfmath.spring.soba.model.domain;
import java.io.Serializable;
import java.sql.Timestamp;
import java.util.Date;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
@Entity
@Table(name = "BILL_PAYMENT")
public class BillPayment implements Serializable {
    private Long id = null;
    private String accountId;
    private String description;
    private double amount;
    private String fromAccount;
```
private String biller;
private String address;
private String city;
private String state;
private String zipcode;
private String status;
private Date scheduleDate;
private Date sendDate;
public BillPayment() {
}
@Id
@Column(name = "ID", unique = true, nullable = false, precision = 10, scale = 0)
public Long getId() {
    return id;
}
@Column(name = "DESCRIPTION", nullable = false, length = 500)
public String getDescription() {
    return description;
}
@Column(name = "AMOUNT", nullable = false)
public double getAmount() {
    return amount;
}
@Column(name = "ACCOUNT_ID", nullable = false, length = 9)
public String getAccountId() {
    return accountId;
}
@Column(name = "FROM_ACCOUNT", nullable = false, length = 25)
public String getFromAccount() {
    return fromAccount;
}
@Column(name = "BILLER", nullable = false, length = 25)
public String getBiller() {
    return biller;
}
@Column(name = "ADDRESS", nullable = false, length = 50)
public String getAddress() {
    return address;
}
@Column(name = "CITY", nullable = false, length = 25)
public String getCity() {
    return city;
}
@Column(name = "STATE", nullable = false, length = 2)
public String getState() {
    return state;
}
Note that whether a DAO is implemented in JDBC or Hibernate, it’s used the same way in a service bean, as the service depends on the DAO interface rather than the implementation. Because of this transparency, we could have just stopped here without showing how the service layer of SOBA performs bill payment tasks with HibernateBillPaymentDao working behind the scene. However, this actually is a good place for us to get a little deeper on Spring validation framework by picking up where we left off with the bill payment use scenario demonstrated in Section 2.7. This is the subject of the next section.

5.6 SPRING DATA VALIDATION FRAMEWORK

To get a complete picture about how Spring validation works with the bill pay service, let’s begin with the programmatic logic flow with the bill pay service implemented in SOBA.

5.6.1 Programmatic Logic Flow with the Bill Pay Service

Since we have covered so much about how Spring MVC works, let us capture the essence of the programmatic logic flow associated with the bill pay service by looking at what classes get involved at each layer. Table 5.2 lists the Java classes associated with this service. It should be clear what each Java class does based on its name and in which layer it is placed.
Table 5.2 Java classes at each layer for the bill pay service

<table>
<thead>
<tr>
<th>Layer</th>
<th>Java Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain object</td>
<td>BillPayment.java</td>
</tr>
<tr>
<td>DAO</td>
<td>BillPaymentDao.java, HibernateBillPaymentDao.java</td>
</tr>
<tr>
<td>Service</td>
<td>BillPayManager.java/SimpleBillPayManager.java, CreateBillPayValidator.java</td>
</tr>
<tr>
<td>Web</td>
<td>CreateBillPayFormController.java, CreateBillPaySuccessController.java</td>
</tr>
<tr>
<td>View</td>
<td>createBillPayForm.jsp, createBillPaySuccess.jsp</td>
</tr>
</tbody>
</table>

Now let us take a look at how the bill pay service flows programatically based on the Java classes associated with it as listed in Table 5.2. This service is initiated when a user clicks on the Bill Payment tab on the home page as shown in Figure 2.18. That home page was generated with the activityList.jsp, which has an embedded link as shown below:

```
<a href="<c:url value="createBillPayForm/customerId/${customerId}/
accountid/${accountid}"/>">Bill Payment</a>
```

Note the url value hard-coded in the above HTML element. It is similar to the RequestMapping we introduced in Listing 4.5 CreateCustomerFormController.java. Not surprisingly, it is mapped to the CreateBillPayFormController.java class as shown in Listing 5.11. When control is directed to this class, the setupForm method is executed first, which creates a billPayment object with some of the attributes pre-populated. Most of the pre-populated attributes here are purely for our convenience except the fromAccount attribute so that we don’t have to type them every time when we test this service. Then, the setupForm method returns control to the createBillPayForm, namely, the createBillPayment.jsp. The form is then presented to the user for entering all required information for a bill payment transaction.Refer back to Figure 2.19 for an actual instance of this form.

Listing 5.11 CreateBillPayFormController.java

```java
package com.perfmath.spring.soba.web;
import java.sql.Timestamp;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
```
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;
import com.perfmath.spring.soba.model.domain.BillPayment;
import com.perfmath.spring.soba.service.BillPayManager;
import com.perfmath.spring.soba.service.CreateBillPayValidator;
import com.perfmath.spring.soba.util.RandomID;

@Controller
@RequestMapping("/createBillPayForm/customerId/{customerId}/accountId/{accountId}")
@SessionAttributes("billPay")
public class CreateBillPayFormController {

    private CreateBillPayValidator validator;
    private BillPayManager billPayManager;
    @Autowired
    public CreateBillPayFormController(BillPayManager billPayManager,
            CreateBillPayValidator validator) {
        this.billPayManager = billPayManager;
        this.validator = validator;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(@PathVariable String accountId, Model model) {
        BillPayment billPayment = new BillPayment();
        billPayment.setFromAccount(accountId);
        billPayment.setDescription("bill pay test");
        billPayment.setAddress("One Way");
        billPayment.setCity("Any City");
        billPayment.setState("CA");
        billPayment.setZipcode("95999");
        model.addAttribute("billPayment", billPayment);
        return "createBillPayForm";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(@PathVariable String customerId,
            @ModelAttribute("billPayment") BillPayment billPayment,
            BindingResult result, SessionStatus status) {
        validator.validate(billPayManager, result);
        if (result.hasErrors()) {
            return "createBillPayForm";
        }
    }
}
After a user fills in the bill payment form and hits Submit button, control is redirected back to the class `CreateBillPayFormController.java`, and the `submitForm` is initiated. As is seen from Listing 5.11, the `submitForm` method validates the `billPayment` object using the `validate` method of its validator. This is where validation gets invoked, as is discussed next. If validation is successful without errors, control is directed to the `CreateBillPaySuccessController`, which presents the responses back to the user via the `createBillPaySuccessForm.jsp` as listed in Table 5.2. If errors occurred during validation, control would be directed back to the bill pay form to display the errors to the user so that the user can correct the errors and resubmit the bill pay again.

Next, we focus on understanding how Spring Validation Interface works in the specific context of this bill pay service example.

5.6.2 Spring Validation Interface

Refer back to Figure 2.19, which shows an actual instance of a bill pay form. Since the user's own account ID has been pre-populated, we don't have to worry about it at all. Of course, in reality, a user might have an option to decide from which account the fund should be used to pay the bill, but that is not very important for our example here. Our concern is how to validate the data entered on this form by the user.

We are particularly concerned about how the bill pay amount is validated. As is shown in Listing 5.12 `CreateBillPayValidator.java`, we use a `Validator` interface in Spring's validation package, which is very basic and usable. This `BillPayment` validator has two methods: `supports(…)` and `validate(…)`. The `supports` method checks whether the passed-in type supports validation, whereas the `validate` method does the actual validation if this validator supports validation. The `validate` method uses a method of `rejectIfEmptyOrWhitespace (errors, "amount", "required.amount", "amount is required.")` on the `Spring ValidationUtils` class to validate the attribute `amount` of type `double`. This entry is rejected if a user enters an empty or whitespace string or an invalid item in the `amount` field on the `createBillPayForm` form defined in `createBillPayForm.jsp` file. If a type mismatch occurs, an error message of "invalid data" would be displayed near the `amount` field entry, according to the `typeMismatch` entry defined in the `messages.properties` file located at the root class path of SOBA.

Of course, you can introduce additional validation based on your business context, after form data format is validated. For example, the `validate` method for this example further validates that the bill pay amount must be larger than zero, after a user enters an amount that is syntactically correct. If a less than zero bill pay amount is entered, it would pass the `rejectIfEmptyOrWhitespace` validation, but not the following validation as
shown in Listing 5.12, which returns control to the bill pay form with an error message of “bill pay amount must be > 0” displayed along with the amount field:

```java
if (billPayment.getAmount() <= 0.0) {
    errors.rejectValue("amount", "invalid.billPayAmount",
    "bill pay amount must be > 0");
}
```

Table 5.3 lists some interesting test cases about how this validation works. Note that the “invalid data” errors like “1.a0” are caught by the Spring form validation APIs before they get to this validator. The “invalid data” message is specified in the messages.properties file.

As you can see, this is a very simple, yet very powerful validation framework. You can consult [http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/validation/](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/validation/) to learn more about how this framework works and what other Spring validation APIs are available to meet your specific needs.

**Listing 5.12 CreateBillPayValidator.java**

```java
package com.perfmath.spring.soba.service;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;
import org.springframework.validation.Errors;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import com.perfmath.spring.soba.model.domain.BillPayment;
import com.perfmath.spring.soba.util.RandomID;
@Component
public class CreateBillPayValidator implements Validator {
    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());
    public boolean supports(Class clazz) {
        return BillPayment.class.isAssignableFrom(clazz);
    }
    public void validate(Object target, Errors errors) {
        BillPayment billPayment = (BillPayment) target;
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "amount",
        "required.amount", "amount is required.");
        if (billPayment.getAmount() <= 0.0) {
            errors.rejectValue("amount", "invalid.billPayAmount",
            "bill pay amount must be > 0");
        }
        billPayment.setId(Long.parseLong(new RandomID(10).getId()));
    }
}
```
Chapter 5: Spring Data Access Framework

billPayment.setScheduleDate(new Timestamp(System.currentTimeMillis()));
billPayment.setSendDate(new Timestamp(System.currentTimeMillis()));
billPayment.setStatus("complete");
}
}

Table 5.3 BillPayment form data validation on the amount attribute

<table>
<thead>
<tr>
<th>Input</th>
<th>Comment</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.a0</td>
<td>Letter/number mixed</td>
<td>Error: invalid data</td>
</tr>
<tr>
<td>abc</td>
<td>A string</td>
<td>Error: invalid data</td>
</tr>
<tr>
<td>1</td>
<td>An integer</td>
<td>Ok ($1.0 was paid)</td>
</tr>
<tr>
<td>empty</td>
<td>An empty string</td>
<td>Error: invalid data/amount is required</td>
</tr>
<tr>
<td>&quot; &quot;</td>
<td>A two-space string</td>
<td>Error: invalid data/amount is required</td>
</tr>
<tr>
<td>1.0 1</td>
<td>A space b/t “0” and “1”</td>
<td>Ok ($1.01 was paid with “ “ ignored)</td>
</tr>
<tr>
<td>1 0.1</td>
<td>A space b/t “1” and “0”</td>
<td>Ok ($10.1 was paid with “ “ ignored)</td>
</tr>
<tr>
<td>10.0.1</td>
<td>Two dots</td>
<td>Error: invalid data</td>
</tr>
<tr>
<td>0.0</td>
<td>Zero amount</td>
<td>Error: bill pay amount must be &gt; 0</td>
</tr>
<tr>
<td>-10.0</td>
<td>negative amount</td>
<td>Error: bill pay amount must be &gt; 0</td>
</tr>
</tbody>
</table>

5.6.3 JSR-303 Bean Validation

JSR-303 bean validation is a spec about validating domain objects using Java annotations under the package of `javax.validation.constraints`. For example, with the Payment domain object shown in Listing 5.10, we could have added the following annotations of `@NotNull` and `@Size` to add JSR-303 based validation to limit the length of a required attribute of `description`:

```java
...  
Import javax.validation.constraints.NotNull;  
Import javax.validation.constraints.Size;  
...
@NotNull  
@Size (min=2, max=50)  
Private String description;  
...
```

Both Spring and Hibernate validators support JSR-303 Bean Validation. However, make a careful decision when choosing which validation mechanism to use with your application. Your application will be less performing and scalable if you have double or even triple validating implemented at all layers for the same validation. Also, keep in mind that it’s hard to achieve the same finer granularity with PSR-303 as with Spring...
validation interface. For example, it might be challenging to use JSR-303 to specify that an entry like “10.0.1” is an invalid input for the amount attribute of the bill payment domain object as shown in Table 5.3. Therefore, most of the time, Spring validation interface is a cleaner, more efficient validation mechanism. In a word, try to avoid using JSR-303 bean validation unless you can’t do without it.

Before concluding this chapter, we describe how to upgrade the deprecated Spring SimpleJdbcDAOSupport used prior to Spring 3.1 next.

5.7 Upgrading Spring Deprecated SimpleJdbcDAOSupport to 3.1 JDBC Support

If your Spring-based application uses SimpleJdbcDAOSupport prior to 3.1, you will get a warning as shown in Figure 5.1 on an Eclipse IDE, indicating that it’s deprecated. This is easy to fix, as all you need to do is to remove “extends SimpleJdbcDaoSupport” and replace all getSimpleJdbcTemplate calls properly. See the upgraded version of this class, Listing 5.4, for more details. Note that this class JdbcBankingTxDaoDeprecated.java is included both in the Ant-based SOBA build (soba3.1) and in the Maven-based SOBA build (soba3.2). Although it is deprecated, it still works.

Figure 5.1 Deprecated Spring SimpleJdbcDaoSupport

Listing 5.13 JdbcBankingTxDaoDeprecated.java

```java
package com.perfmath.spring.soba.model.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;

import com.perfmath.spring.soba.model.domain.BankingTx;

public class JdbcBankingTxDaoDeprecated extends SimpleJdbcTemplate implements BankingTxDao {

// Method implementations...
}
```

```java
package com.perfmath.spring.soba.model.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;

import com.perfmath.spring.soba.model.domain.BankingTx;

public class JdbcBankingTxDaoDeprecated extends SimpleJdbcTemplate implements BankingTxDao {

// Method implementations...
}
```
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.ParameterizedRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

import com.perfmath.spring.soba.model.domain.BankingTx;

public class JdbcBankingTxDaoDeprecated extends SimpleJdbcDaoSupport implements BankingTxDao {
    public List<BankingTx> getTransactionList() {
        logger.info("Getting transactions!");
        List<BankingTx> txs = getSimpleJdbcTemplate().query("SELECT TRANSACTION_ID, TRANS_DATE, TYPE, "
                + " INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS FROM BANKINGTX",
                new TransactionMapper());
        return txs;
    }

    public List<BankingTx> getTransactionList(String accountId) {
        logger.info("Getting transactions!");
        List<BankingTx> txs = getSimpleJdbcTemplate().query("SELECT TRANSACTION_ID, TRANS_DATE, TYPE, "
                + " INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS FROM BANKINGTX WHERE ACCOUNT_ID = ?"
                + " ORDER BY TRANS_DATE DESC",
                new TransactionMapper(), accountId);
        return txs;
    }

    public void insert(BankingTx transaction) {
        String sql = "INSERT INTO BANKINGTX (TRANSACTION_ID, TRANS_DATE, TYPE,"
                + " INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS ) "
                + "VALUES (:transactionId, :transDate, :type, :initiator, "
                + ":description, :amount, :balance, :accountId, :status)"
                + ";

        SqlParameterSource parameterSource =
            new BeanPropertySqlParameterSource(transaction);

        int count = getSimpleJdbcTemplate().update(sql, parameterSource);
    }

    public void insertBatch(List<BankingTx> trans) {
        String sql = "INSERT INTO BANKINGTX (TRANSACTION_ID, TRANS_DATE, TYPE,"
                + " INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS ) "
                + "VALUES (:transactionId, :transDate, :type, :initiator, "
                + ":description, :amount, :balance, :accountId, :status)"
                + ";

        SqlParameterSource parameterSource =
            new BeanPropertySqlParameterSource(transaction);

        int count = getSimpleJdbcTemplate().update(sql, parameterSource);
    }
}
List<SqlParameterSource> parameters = new ArrayList<SqlParameterSource>();
for (BankingTx Transaction : trans) {
    parameters.add(new BeanPropertySqlParameterSource(Transaction));
}

getSimpleJdbcTemplate().batchUpdate(sql,
    parameters.toArray(new SqlParameterSource[0]));
}

public BankingTx findByTransactionID(String transID) {

    String sql = "SELECT TRANSACTION_ID, TRANS_DATE, TYPE, " + " INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS " + " FROM BANKINGTX WHERE TRANSACTION_ID = " + transID + ";"
    BankingTx trans = getSimpleJdbcTemplate().queryForObject(sql,
        new TransactionMapper());

    return trans;
}

public void update(BankingTx tx) {
}

public void delete(String txId) {
    String sql = "DELETE BANKINGTX WHERE TRANSACTION_ID = ?";
    int count = getSimpleJdbcTemplate().update(sql, txId);
}

public List<Map<String, Object>> findAll() {
    String sql = "SELECT * FROM BANKINGTX";

    List<Map<String, Object>> trans = getSimpleJdbcTemplate().queryForList(sql,
        BankingTx.class);
    return trans;
}

public String getAccountId(String transID) {

    String sql = "SELECT ACCOUNT_ID FROM BANKINGTX WHERE TRANSACTION_ID = ?";
    String accountId = getSimpleJdbcTemplate().queryForObject(sql,
        String.class, transID);
    return accountId;
}
public int countAll() {
    String sql = "SELECT COUNT(*) FROM BANKINGTX";
    int count = getJdbcTemplate().queryForInt(sql);
    return count;
}

private static class TransactionMapper implements ParameterizedRowMapper<BankingTx> {
    public BankingTx mapRow(ResultSet rs, int rowNum) throws SQLException {
        BankingTx tx = new BankingTx();
        tx.setTransactionId(rs.getInt("TRANSACTION_ID"));
        tx.setTransDate(rs.getTimestamp("TRANS_DATE"));
        tx.setType(rs.getString("TYPE"));
        tx.setInitiator(rs.getString("INITIATOR"));
        tx.setDescription(rs.getString("DESCRIPTION"));
        tx.setAmount(rs.getDouble("AMOUNT"));
        tx.setBalance(rs.getDouble("BALANCE"));
        tx.setAccountId(rs.getString("ACCOUNT_ID"));
        tx.setStatus(rs.getString("STATUS"));
        return tx;
    }
}

5.8 SUMMARY

In this chapter, we explained how Spring Data Access Framework supports JDBC and Hibernate data access methods. Real SOBA code examples were used to demonstrate the key concepts and technologies associated with JDBC and Hibernate. We also covered the Spring validation interface using the bill pay service. Since SOBA is a fully functioning, integrated sample, you can explore more JDBC and Hibernate features as well as Spring data validations using SOBA as your experimental platform.

The next chapter covers how RESTful Web Services is supported by Spring and applied to SOBA. This is an interesting subject, since RESTful Web Services has become more and more popular for building enterprise applications. Since the JDBC and Hibernate parts are re-usable, they will not be repeated.

RECOMMENDED READING

Study Chapter 13 Data Access with JDBC in the Spring Reference Documentation 3.1 to have a more thorough understanding on how Spring supports JDBC. Here is a list of the subjects regarding Spring JDBC support introduced there (highlighted subjects are covered in this chapter):

Spring Reference Documentation 3.1/ Chapter 13 Data Access with JDBC:

13.1 Introduction to Spring Framework JDBC
13.2 Using the JDBC core classes to control basic JDBC processing and error handling

13.3 Controlling database connections

13.4 JDBC batch operations

13.5 Simplifying JDBC operations with the SimpleJdbc classes

13.6 Modeling JDBC operations as Java objects

13.7 Common problems with parameter and data value handling

13.8 Embedded database support

13.9 Initializing a Datasource

You may also want to review Chapter 14 Object Relational Mapping (ORM) Data Access. The contents of this chapter are listed below, with those we covered highlighted to help you decide how deep you want to go yourself.

Spring Reference Documentation 3.1/ Chapter 14 Object Relational Mapping (ORM) Data Access:

14.1 Introduction to ORM with Spring

14.2 General ORM Integration Considerations

14.3 Hibernate

14.4 JDO

14.5 JPA

14.6 iBATIS SQL Maps

SELF-REVIEW EXERCISES

5.1 What’s the difference between a Spring JDBC data source and a JDBC connection pool?

5.2 How do you use Spring’s JdbcTemplate and NamedParameterJdbcTemplate with different types of SQLs?

5.3 What are the benefits of using Hibernate rather than more conventional JDBC?

5.4 Discuss the differences among three different type of data persistence mechanisms (JDBC, Hibernate, and JPA).

5.5 Discuss when you need or don’t need an HBM file?

5.6 For the input “1.0a” in Table 5.3, is it caught in the validator or somewhere else?

5.7 How do you validate a Java BigDecimal type with Spring’s validation framework?