Spring MVC Tutorial – Contact List Application

Step 15: Creating the Database DAO

SOFTWARE GUILD

The Learning House

427 S 4th Street #300

Louisville KY 40202

Step 15: Creating the Database DAO

Overview

In Step 4 of this tutorial, we created an implementation of our DAO interface that kept all Contacts in memory but lost all of our changes when our application stopped running. In Step 15, we will create an implementation of the DAO interface that will persist all changes to the database that we created in Step 14. Our implementation uses a Spring JdbcTemplate to help communicate with the database server. This step in the tutorial assumes that you are familiar with Spring JdbcTemplates and the configuration of the MySQL datasource in Spring.

Database DAO Implementation

This implementation of the DAO interface will use a JdbcTemplate to persist Contact data to the database. The in-memory version of the DAO simulated the auto_increment functionality of the database by incrementing a counter to assign contactId values. This version of the DAO will use the MySQL auto_increment feature to assign contactId values.

Notes:

1. All SQL code must be in the form of Prepared Statements. This helps prevent SQL injection attacks against our application.
2. Spring JdbcTemplate:
    a. Declare a reference to a JdbcTemplate.
    b. Create setter for JdbcTemplate so we can use Spring Setter Injection to set the reference (this is configured in the spring-persistence.xml file in the following section).
3. Wrap addContact in a transaction so that we are guaranteed that the insert statement and the LAST_INSERT_ID call use the same connection. If different connections are used for these queries in a high-concurrency environment (i.e. many users using the database at the same time), it is possible to get the wrong value for the last inserted id.
4. Any time we get values back from the database, we must provide an instance of a class that knows how to map database rows into model objects. In our case, we pass an instance of the ContactMapper class.
5. The ContactMapper class knows how to map rows from the contacts table to properties on the Contact model object.

```java
package com.swcguild.contactlistmvc.dao;

import com.swcguild.contactlistmvc.model.Contact;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.simple.RowMapper;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class ContactListDaoDbImpl implements ContactListDao {

    // #1 - All SQL code is in the form of Prepared Statements
    private static final String SQL_INSERT_CONTACT
            = "insert into contacts (first_name, last_name, company, phone, email) values (?, ?, ?, ?, ?)";
    private static final String SQL_DELETE_CONTACT
            = "delete from contacts where contact_id = ?";
    private static final String SQL_SELECT_CONTACT
            = "select * from contacts where contact_id = ?";
    private static final String SQL_UPDATE_CONTACT
            = "update contacts set first_name = ?, last_name = ?, company = ?, phone = ?, email = ? where contact_id = ?";
    private static final String SQL_SELECT_ALL_CONTACTS
            = "select * from contacts";
    private static final String SQL_SELECT_CONTACTS_BY_LAST_NAME
            = "select * from contacts where last_name = ?";
    private static final String SQL_SELECT_CONTACTS_BY_COMPANY
            = "select * from contacts where company = ?";

    // #2a - Declare JdbcTemplate reference - the instance will be handed to us by Spring
    private JdbcTemplate jdbcTemplate;

    // #2b - We are using Setter Injection to direct Spring to hand us an instance of
    //       the JdbcTemplate (see the Spring Configuration section below for configuration
    //       details).
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override
    // #3 - Wrap addContact in a transaction so the we are guaranteed to get the
    //      correct LAST_INSERT_ID for our row.
    @Transactional(propagation = Propagation.REQUIRED, readOnly = false)
    public Contact addContact(Contact contact) {
        jdbcTemplate.update(SQL_INSERT_CONTACT,
                contact.getFirstName(),
                contact.getLastName(),
                contact.getCompany(),
                contact.getPhone(),
                contact.getEmail());
        contact.setContactId(jdbcTemplate.queryForObject("select LAST_INSERT_ID()",
                Integer.class));
        return contact;
    }

    @Override
    public void removeContact(int contactId) {
        jdbcTemplate.update(SQL_DELETE_CONTACT, contactId);
    }
```

```
    @Override
    public void updateContact(Contact contact) {
        jdbcTemplate.update(SQL_UPDATE_CONTACT,
                contact.getFirstName(),
                contact.getLastName(),
                contact.getCompany(),
                contact.getPhone(),
                contact.getEmail(),
                contact.getContactId());
    }

    @Override
    // #4 - getAllContacts, getContactById, searchContactsByLastName, and
    //      searchContactsByCompany require us to pass in mapper
    //      class to map the rows from the database into Contact objects
    public List<Contact> getAllContacts() {
        return jdbcTemplate.query(SQL_SELECT_ALL_CONTACTS, new ContactMapper());
    }

    @Override
    public Contact getContactById(int contactId) {
        try {
            return jdbcTemplate.queryForObject(SQL_SELECT_CONTACT,
                    new ContactMapper(), contactId);
        } catch (EmptyResultDataAccessException ex) {
            // there were no results for the given contact id - we just want to
            // return null in this case
            return null;
        }
    }

    @Override
    public List<Contact> searchContactsByLastName(String lastName) {
        return jdbcTemplate.query(SQL_SELECT_CONTACTS_BY_LAST_NAME, new ContactMapper());
    }

    @Override
    public List<Contact> searchContactsByCompany(String company) {
        return jdbcTemplate.query(SQL_SELECT_CONTACTS_BY_COMPANY, new ContactMapper());
    }

    // #5 - This class maps the columns in the 'contacts' table into properties on the
    //      Contact object
    private static final class ContactMapper implements RowMapper<Contact> {

        public Contact mapRow(ResultSet rs, int rowNum) throws SQLException {
            Contact contact = new Contact();
            contact.setContactId(rs.getInt("contact_id"));
            contact.setFirstName(rs.getString("first_name"));
            contact.setLastName(rs.getString("last_name"));
            contact.setCompany(rs.getString("company"));
            contact.setPhone(rs.getString("phone"));
            contact.setEmail(rs.getString("email"));
            return contact;
        }
    }
}
```

Spring Configuration

Connecting to the database using a JdbcTemplate in our DAO requires some additional Spring configuration. We must do the following:

1. Enable Spring database transactions and configure transactions to be annotation-driven
2. Define a data source — this bean definition contains all of the information needed to connect with the database including:
    a. The bean class to instantiate
    b. The class to be used to connect to the database (i.e. the database driver class)
    c. The URL with which to connect to the database server (i.e. host, port, and name of the database)
    d. The username and password for the database
    e. Connection pool settings
3. Define a bean for the Transaction Manager. The Transaction Manager requires a data source, so we use Setter Injection to inject the data source defined in step 1 above.
4. Define a bean for the JdbcTemplate that will be injected into our DAO. The JdbcTemplate requires a data source, so we use Setter Injection to inject the data source defined in step 1 above.
5. Update our DAO bean:
    a. Change the class attribute to that of the new database implementation
    b. Have Spring inject the JdbcTemplate (defined in step 2) via Setter Injection

Both your spring-persistence.xml and test-applicationContext.xml should look like this with one exception — **test-applicationContext.xml must point to contact_list_test in the Data Source bean definition**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
              xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/mvc
  http://www.springframework.org/schema/mvc/spring-mvc.xsd
  http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop-3.2.xsd">

    <!-- Bean definitions go here -->
    <!-- #1 Use annotation-driven transactions -->
    <tx:annotation-driven />

    <!-- Bean definitions go here →
    <!-- #2 Define Data Source -->
    <bean id="dataSource"
          class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url"
        <!-- test-applicationContext.xml should point to contact_list_test -->
        <!-- value="jdbc:mysql://localhost:3306/contact_list_test" /> -->
                  value="jdbc:mysql://localhost:3306/contact_list" />
        <property name="username" value="root" />
        <property name="password" value="root" />
        <property name="initialSize" value="5" />
        <property name="maxTotal" value="10" />
    </bean>

    <!-- #3 Define Transaction Manager -->
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <!-- #3 Define JdbcTemplate -->
    <bean id="jdbcTemplate"
          class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <!-- #3 Define the databae DAO -->
    <bean id="contactListDao"
          class="com.swcguild.contactlistmvc.dao.ContactListDaoDbImpl">
        <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>
</beans>
```

Unit Tests

Test methods from Step 4 should work with one small modification to the setUp method. Because our DAO now writes data to the database that persists between test runs, we must put the database in a known state before each test is run. We do this using a JdbcTemplate and SQL statements that clean up our table before each test. Make changes to your test file so that it looks like this:

```
package com.swcguild.contactlistmvc;

import com.swcguild.contactlistmvc.dao.ContactListDao;
import com.swcguild.contactlistmvc.model.Contact;
import static junit.framework.TestCase.assertEquals;
import static junit.framework.TestCase.assertNull;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;

public class ContactListDaoTest {

    private ContactListDao dao;

    public ContactListDaoTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }
```

```
@Before
public void setUp() {
// NEW CODE START- this cleans up the database table before each test
    // Ask Spring for my DAO
    ApplicationContext ctx =
                    new ClassPathXmlApplicationContext("test-applicationContext.xml");
    dao = (ContactListDao) ctx.getBean("contactListDao");

    // Grab a JdbcTemplate to use for cleaning up
    JdbcTemplate cleaner = (JdbcTemplate) ctx.getBean("jdbcTemplate");
    cleaner.execute("delete from contacts");
// NEW CODE STOP
}

@After
public void tearDown() {
}

@Test
public void addGetDeleteContact() {
    // create new contact
    Contact nc = new Contact();
    nc.setName("John");
    nc.setEmail("john@john.com");
    nc.setPhone("1234445678");

    dao.addContact(nc);

    Contact fromDb = dao.getContactById(nc.getContactId());

    assertEquals(fromDb.getContactId(), nc.getContactId());
    assertEquals(fromDb.getName(), nc.getName());
    assertEquals(fromDb.getPhone(), nc.getPhone());
    assertEquals(fromDb.getEmail(), nc.getEmail());

    dao.removeContact(nc.getContactId());

    assertNull(dao.getContactById(nc.getContactId()));
}
```

```
    @Test
    public void addUpdateContact() {
        // create new contact
        Contact nc = new Contact();
        nc.setName("Jimmy Smith");
        nc.setEmail("jimmy@smith.com");
        nc.setPhone("1112223333");

        dao.addContact(nc);

        nc.setPhone("9999999999");

        dao.updateContact(nc);

        Contact fromDb = dao.getContactById(nc.getContactId());

        assertEquals(fromDb.getContactId(), nc.getContactId());
        assertEquals(fromDb.getName(), nc.getName());
        assertEquals(fromDb.getPhone(), nc.getPhone());
        assertEquals(fromDb.getEmail(), nc.getEmail());
    }

    @Test
    public void getAllContacts() {
        // create new contact
        Contact nc = new Contact();
        nc.setName("Jimmy Smith");
        nc.setEmail("jimmy@smith.com");
        nc.setPhone("1112223333");

        dao.addContact(nc);

        // create new contact
        Contact nc2 = new Contact();
        nc2.setName("John Jones");
        nc2.setEmail("john@jones.com");
        nc2.setPhone("5556667777");

        dao.addContact(nc);

        Contact[] cArr = dao.getAllContacts();
        assertEquals(cArr.length, 2);
    }
}
```

Wrap-up

That concludes Step 15 of the tutorial. You now have a fully-functional web application that reads and writes data to a database. In the next step, we will see how to integrate Bootstrap into our Spring MVC application.