Spring MVC Tutorial – Contact List Application

Step 04: Creating the In-Memory DAO

SOFTWARE GUILD

The Learning House

427 S 4th Street #300

Louisville KY 40202

Step 04: Creating the In-Memory DAO

Overview

In this step, we will create an in-memory version of our DAO. This version of the DAO will hold our Contact objects in a HashMap in memory as long as the application is running. All of our changes will be lost when the application is stopped. This step assumes that you have successfully completed all previous steps in the tutorial.

In-Memory DAO Implementation

This version of the DAO will hold all Contact objects in a HashMap. We'll use a static counter to assign contact ids to each Contact as it is added to the DAO. This will simulate the auto increment features for primary keys in a database.

Notes:

1. We use a HashMap to hold our Contacts so we can enforce the uniqueness of the contact id.
2. We use Java 8 aggregate operations, lambdas, and predicates to search the HashMap values.

Create the following class in the com.swcguild.contactlistmvc.dao package:

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

import com.swcguild.contactlistmvc.model.Contact;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.function.Predicate;
import java.util.stream.Collectors;
```

```
public class ContactListDaoInMemImpl implements ContactListDao {

    // holds all of our Contact objects - simulates the database
    private Map<Integer, Contact> contactMap = new HashMap<>();
    // used to assign ids to Contacts - simulates the auto increment
    // primary key for Contacts in a database
    private static int contactIdCounter = 0;

    @Override
    public Contact addContact(Contact contact) {
        // assign the current counter values as the contactid
        contact.setContactId(contactIdCounter);
        // increment the counter so it is ready for use for the next contact
        contactIdCounter++;
        contactMap.put(contact.getContactId(), contact);
        return contact;
    }

    @Override
    public void removeContact(int contactId) {
        contactMap.remove(contactId);
    }

    @Override
    public void updateContact(Contact contact) {
        contactMap.put(contact.getContactId(), contact);
    }

    @Override
    public List<Contact> getAllContacts() {
        Collection<Contact> c = contactMap.values();
        return new ArrayList(c);
    }

    @Override
    public Contact getContactById(int contactId) {
        return contactMap.get(contactId);
    }

    @Override
    public List<Contact> searchContacts(Map<SearchTerm, String> criteria) {
        // Get all the search terms from the map
        String firstNameCriteria = criteria.get(SearchTerm.FIRST_NAME);
        String lastNameCriteria = criteria.get(SearchTerm.LAST_NAME);
        String companyCriteria = criteria.get(SearchTerm.COMPANY);
        String phoneCriteria = criteria.get(SearchTerm.PHONE);
        String emailCriteria = criteria.get(SearchTerm.EMAIL);

        // Declare all the predicate conditions
        Predicate<Contact> firstNameMatches;
        Predicate<Contact> lastNameMatches;
        Predicate<Contact> companyMatches;
        Predicate<Contact> phoneMatches;
        Predicate<Contact> emailMatches;
```

```
        // Placeholder predicate - always returns true.  Used for search terms
        // that are empty
        Predicate<Contact> truePredicate = (c) -> {return true;};

        // Assign values to predicates.  If a given search term is empty, just
        // assign the default truePredicate, otherwise assign the predicate that
        // properly filters for the given term.
        firstNameMatches = (firstNameCriteria == null || firstNameCriteria.isEmpty())
                ? truePredicate
                : (c) -> c.getFirstName().equals(firstNameCriteria);

        lastNameMatches = (lastNameCriteria == null || lastNameCriteria.isEmpty())
                ? truePredicate
                :(c) -> c.getLastName().equals(lastNameCriteria);

        companyMatches = (companyCriteria == null || companyCriteria.isEmpty())
                ? truePredicate
                :(c) -> c.getCompany().equals(companyCriteria);

        phoneMatches = (phoneCriteria == null || phoneCriteria.isEmpty())
                ? truePredicate
                :(c) -> c.getPhone().equals(phoneCriteria);

        emailMatches = (emailCriteria == null || emailCriteria.isEmpty())
                ? truePredicate
                :(c) -> c.getEmail().equals(emailCriteria);

        // Return the list of Contacts that match the given criteria.  To do this we
        // just AND all the predicates together in a filter operation.
        return contactMap.values().stream()
                .filter(firstNameMatches
                            .and(lastNameMatches)
                            .and(companyMatches)
                            .and(phoneMatches)
                            .and(emailMatches))
                .collect(Collectors.toList());
    }
}
```

Spring Configuration

As described in the Tutorial Overview, we are using the Spring Dependency Injection (DI) container for this project. This means that we must create a bean definition in our Spring application context files for this implementation of the DAO. We need to put this definition in both the spring-persistence.xml and the test-applicationContext.xml files (test-applicationContext.xml should be in the **Other Test Sources** folder in the Project view). Add the following entry to each of these files:

```
<bean id="contactListDao"
          class="com.swcguild.contactlistmvc.dao.ContactListDaoInMemImpl">
</bean>
```

Unit Tests

Now that we have a concrete implementation of our DAO interface, we must create some unit tests. The following unit test has some very basic test cases to run against our implementation.

As we saw in Step 01 of the tutorial, we have configured our web application so that Tomcat will automatically load the Spring context files when our web application starts up. For our unit test, we must create and load the application context by hand. This happens in the setUp method of our test class. The setUp method runs before each of our test methods, meaning that the application context is loaded fresh for each test.

Create the following class in the com.swcguild.contactlistmvc.dao package in the **Test Packages** section in the Project view in NetBeans:

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

import com.swcguild.contactlistmvc.model.Contact;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import static junit.framework.TestCase.assertEquals;
import static junit.framework.TestCase.assertNull;
import static junit.framework.TestCase.assertTrue;
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;

public class ContactListDaoTest {

    private ContactListDao dao;

    public ContactListDaoTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
        ApplicationContext ctx
                = new ClassPathXmlApplicationContext("test-applicationContext.xml");
        dao = ctx.getBean("contactListDao", ContactListDao.class);
    }
```

```
@After
public void tearDown() {
}

@Test
public void addGetDeleteContact() {
    // Create new contact
    Contact nc = new Contact();
    nc.setFirstName("John");
    nc.setLastName("Doe");
    nc.setCompany("Oracle");
    nc.setEmail("john@doe.com");
    nc.setPhone("1234445678");

    dao.addContact(nc);

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

    assertEquals(fromDb, nc);

    dao.removeContact(nc.getContactId());

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

@Test
public void addUpdateContact() {
    // Create new contact
    Contact nc = new Contact();
    nc.setFirstName("Jimmy");
    nc.setLastName("Smith");
    nc.setCompany("Sun");
    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, nc);
}
```

```
@Test
public void getAllContacts() {
    // Create new contact
    Contact nc = new Contact();
    nc.setFirstName("Jimmy");
    nc.setLastName("Smith");
    nc.setCompany("Sun");
    nc.setEmail("jimmy@smith.com");
    nc.setPhone("1112223333");

    dao.addContact(nc);

    // Create new contact
    Contact nc2 = new Contact();
    nc2.setFirstName("John");
    nc2.setLastName("Jones");
    nc2.setCompany("Apple");
    nc2.setEmail("john@jones.com");
    nc2.setPhone("5556667777");

    dao.addContact(nc2);

    List<Contact> cList = dao.getAllContacts();
    assertEquals(cList.size(), 2);
}

@Test
public void searchContacts() {
    // Create new contact
    Contact nc = new Contact();
    nc.setFirstName("Jimmy");
    nc.setLastName("Smith");
    nc.setCompany("Sun");
    nc.setEmail("jimmy@smith.com");
    nc.setPhone("1112223333");

    dao.addContact(nc);

    // Create new contact
    Contact nc2 = new Contact();
    nc2.setFirstName("John");
    nc2.setLastName("Jones");
    nc2.setCompany("Apple");
    nc2.setEmail("john@jones.com");
    nc2.setPhone("5556667777");

    dao.addContact(nc2);
```

```
        // Create new contact - same last name as first contact but different
        // company
        Contact nc3 = new Contact();
        nc3.setFirstName("Steve");
        nc3.setLastName("Smith");
        nc3.setCompany("Microsoft");
        nc3.setEmail("steve@msft.com");
        nc3.setPhone("5552221234");

        dao.addContact(nc3);

        // Create search criteria
        Map<SearchTerm, String> criteria = new HashMap<>();
        criteria.put(SearchTerm.LAST_NAME, "Jones");
        List<Contact> cList = dao.searchContacts(criteria);
        assertEquals(1, cList.size());
        assertEquals(nc2, cList.get(0));

        // New search criteria - look for Smith
        criteria.put(SearchTerm.LAST_NAME, "Smith");
        cList = dao.searchContacts(criteria);
        assertEquals(2, cList.size());

        // Add company to search criteria
        criteria.put(SearchTerm.COMPANY, "Sun");
        cList = dao.searchContacts(criteria);
        assertEquals(1, cList.size());
        assertEquals(nc, cList.get(0));

        // Change company to Microsoft, should get back nc3
        criteria.put(SearchTerm.COMPANY, "Microsoft");
        cList = dao.searchContacts(criteria);
        assertEquals(1, cList.size());
        assertEquals(nc3, cList.get(0));

        // Change company to Apple, should get back nothing
        criteria.put(SearchTerm.COMPANY, "Apple");
        cList = dao.searchContacts(criteria);
        assertEquals(0, cList.size());
    }
}
```

If everything is properly implemented, you should be able to right-click on the ContactListDaoTest file and select **Test File**. If these tests pass, you have successfully completed this part of the tutorial.

Wrap-up

That does it for this step. In this part of the tutorial, we did the following:

1. Created an in-memory version of our DAO — this implementation is designed to mimic the operations of a database, including auto-generation of IDs for Contact objects upon insertion
2. Used aggregate operations, lambdas, and predicates to implement the search functionality
3. Created a rudimentary set of unit tests for our DAO (these tests will be reused to test the database implementation of our DAO in a later step, as well)
4. Added bean entries into our Spring application context configuration files so that the Spring DI container can control the lifecycle of our DAO

In the next step, we will add REST endpoints for Contact CRUD operations to the Home controller.