Java and Databases Unit

Lesson 1: Introduction to JdbcTemplates - Notes

Lesson 1: Introduction to JdbcTemplates

Overview

This document is a step-by-step tutorial that covers moving from an in-memory DAO to a DAO that is connected to a MySQL database. We will start with already developed code (you can pull from the class GitHub repository - the project is called StudentRosterDB) and go from there.

The Database

The database for this tutorial is very simple. It consists of just one table called **students** with the following structure:

Column	Type	Null	Default	Extra
id	int(10)	No		auto_increment
first_name	varchar(30)	No		
last_name	varchar(30)	No		
major	varchar(25)	No		
street	varchar(25)	No		
city	varchar(50)	No		
state	varchar(2)	No		
zipcode	varchar(5)	No		

SQL Creation Script

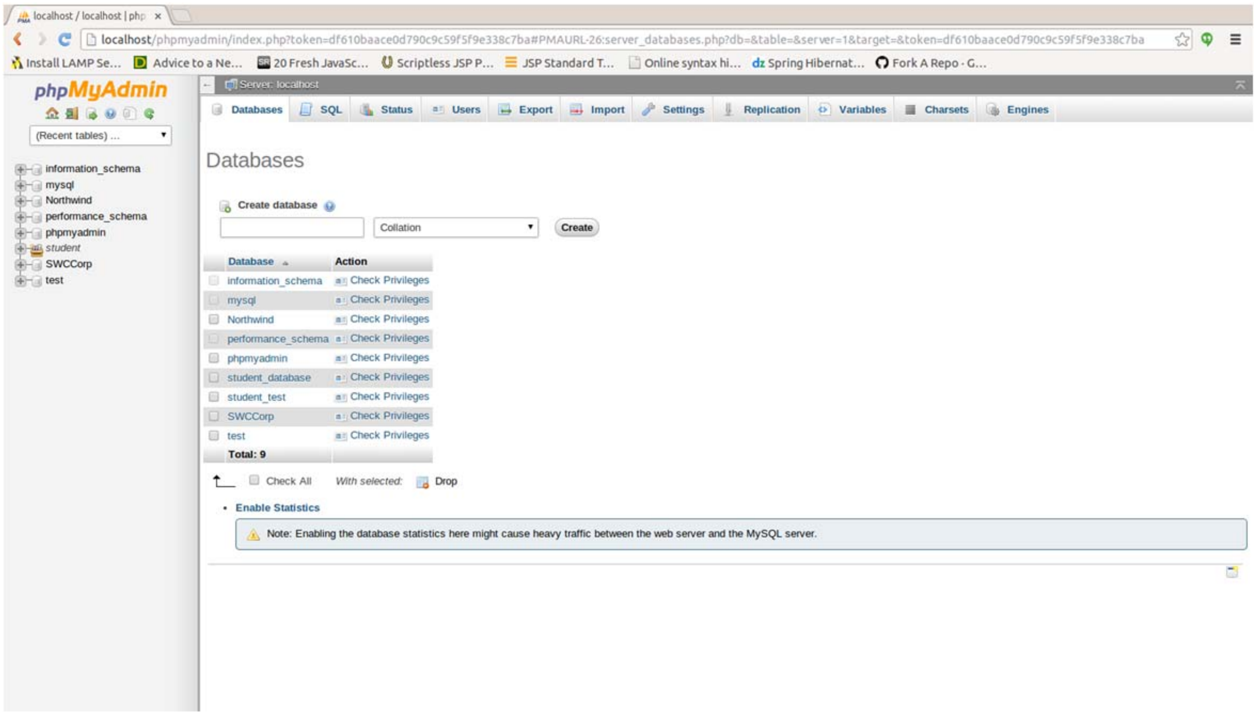
The following script creates the **students** table:

```
-- phpMyAdmin SQL Dump
-- version 4.0.10deb1
-- http://www.phpmyadmin.net
--
-- Host: localhost
-- Generation Time: Oct 12, 2014 at 12:24 PM
-- Server version: 5.5.38-0ubuntu0.14.04.1
-- PHP Version: 5.5.9-1ubuntu4.4

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
SET time_zone = "+00:00";

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;

--
-- Database: `StudentRoster`
--
-- --------------------------------------------------------
--
-- Table structure for table `students`
--

CREATE TABLE IF NOT EXISTS `students` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(30) NOT NULL,
  `last_name` varchar(30) NOT NULL,
  `major` varchar(25) NOT NULL,
  `street` varchar(25) NOT NULL,
  `city` varchar(25) NOT NULL,
  `state` varchar(2) NOT NULL,
  `zipcode` varchar(5) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
```

Creating the Database

Now we need to create two databases (one for dev and one for test) and then run the creation script against each.

1. Log into phpmyadmin and click on the **Databases** tab.



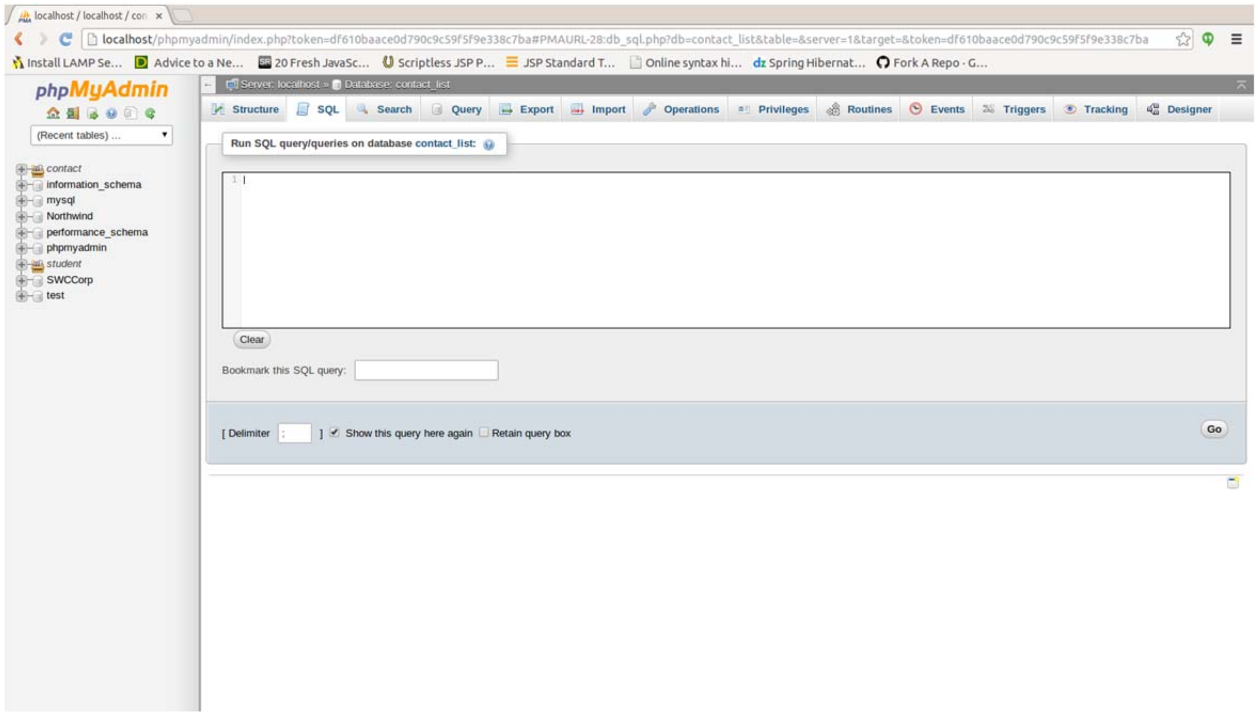
Databases

Database	Action
information_schema	Check Privileges
mysql	Check Privileges
Northwind	Check Privileges
performance_schema	Check Privileges
phpmyadmin	Check Privileges
student_database	Check Privileges
student_test	Check Privileges
SWCCorp	Check Privileges
test	Check Privileges
Total: 9	

Note: Enabling the database statistics here might cause heavy traffic between the web server and the MySQL server.

2. Type StudentRoster in the **Create database** textbox and click the **Create** button.
3. Type StudentRoster_test in the **Create database** textbox and click the **Create** button. Both StudentRoster and StudentRoster_test should now appear in the list of databases.

4. Click on StudentRoster and then click the **SQL** tab:


localhost / localhost / con
localhost/phpmyadmin/index.php?token=df610baace0d790c9c59f5f9e338c7ba#PMAURL-28:db_sql.php?db=contact_list&table=&server=1&target=&token=df610baace0d790c9c59f5f9e338c7ba
phpMyAdmin
(Recent tables) ...
contact
information_schema
mysql
Northwind
performance_schema
phpmyadmin
student
SWCCorp
test
Server: localhost » Database: contact_list
Structure
SQL
Search
Query
Export
Import
Operations
Privileges
Routines
Events
Triggers
Tracking
Designer
Run SQL query/queries on database contact_list:
Clear
Bookmark this SQL query:
[Delimiter ;] Show this query here again Retain query box
Go

5. Now copy the creation script (above) and paste it into the SQL window. Click the **Go** button. This will create the contacts table in the StudentRoster database.
6. Now click on the **Home** icon in the upper left of the screen and then click on the **Databases** tab. This will list all of the databases on the server.
7. Click StudentRoster_test and then click the **SQL** tab.
8. Now copy the creation script (above) and paste it into the SQL window and click the **Go** button (just as you did for contact_list). This will create the contacts table in the StudentRoster_test database.

Maven Dependencies

We have to add three dependencies to our POM file - one for Spring JDBC support, one for code that allows us to communicate with the MySQL database, and one that allows us to use what is known as **connection pooling**. Connection pooling creates a group, or pool, of connections to the database when the application starts up. The application then uses the connections in the pool when talking to the database. Using connections from the pool is more efficient than creating a new connection from scratch each time we need to talk to the database.

Please add the following dependencies to your POM file:

```
        <!-- Spring JDBC Support Library -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>4.1.0.RELEASE</version>
        </dependency>
        <!-- Apache Commons Database Connection Pooling Library -->
        <dependency>
            <groupId>commons-dbcp</groupId>
            <artifactId>commons-dbcp</artifactId>
            <version>1.4</version>
        </dependency>
        <!-- MySQL Java Database Driver - allows us to connect to our database -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.5</version>
        </dependency>
```

The DAO

Now we will create an implementation of our StudentDao interface that communicates with the database and table that we created in the previous step. Our implementation will use Spring's JdbcTemplate object and will rely on the database to assign the unique ids to our Student objects. Please see the comments in the code below for details as to what is going on in the code:

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

import com.swcguild.studentrostermaven.domain.Student;
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.RowMapper;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

/**
 *
 * @author apprentice
 */
public class StudentDaoDbImpl implements StudentDao {

    // All SQL statements are in the form of Prepared Statements
    // This prevents SQL Injection Attacks
    private static final String SQL_INSERT_STUDENT
            = "insert into students (first_name, last_name, major, street, city, state,
zipcode) "
            + "values (?, ?, ?, ?, ?, ?, ?)";

    private static final String SQL_SELECT_STUDENT_BY_ID
            = "select * from students where id = ?";

    private static final String SQL_SELECT_STUDENT_BY_LAST_NAME
            = "select * from students where last_name = ?";

    private static final String SQL_UPDATE_STUDENT
            = "update students set first_name = ?, last_name = ?, major = ?, "
            + "street = ?, city = ?, state = ?, zipcode = ? where id = ?";

    private static final String SQL_DELETE_STUDENT
            = "delete from students where id = ?";

    // reference to JdbcTemplate
    private JdbcTemplate jdbcTemplate;

    // We'll use setter injection for the JdbcTemplate
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

        this.jdbcTemplate = jdbcTemplate;
    }

    // For us, inserting a student into the database is a two step process.
    // First we insert the data into the row, then we ask for the auto incremented
    // id of the newly inserted student so we can set it on the Student object.
    // Since many process can access the database at the same time, we need to
    // ensure that we use the connection for both the insert and when we check
    // for the id - this is only way to make sure we get the id of the row that
    // we inserted rather than the id of a row some other process inserted.
    // Wrapping the method in a transaction annotation accomplishes this for
    // us.
    // Deletes, updates and inserts all use the JdbcTemplate 'update' method
    @Override
    @Transactional(propagation = Propagation.REQUIRED, readOnly = false)
    public void addStudent(Student student) {
        jdbcTemplate.update(SQL_INSERT_STUDENT,
                student.getFirstName(),
                student.getLastName(),
                student.getMajor(),
                student.getStreet(),
                student.getCity(),
                student.getState(),
                student.getZipcode());
        // now set the newly generated student id value on our object
        student.setStudentId(jdbcTemplate.queryForObject("select LAST_INSERT_ID()",
                              Integer.class));
    }

    // Our API semantics state that queries for students that don't exist should return
    // null.  The Spring JdbcTemplate API semantics are that querying for an object
    // that doesn't exist throws an exception.  So...we'll put our operation in a
    // try/catch block and just return null if we encounter an exception.
    @Override
    public Student getStudentById(int id) {
        try {
            return jdbcTemplate.queryForObject(SQL_SELECT_STUDENT_BY_ID,
                                              new StudentMapper(), id);
        } catch (EmptyResultDataAccessException e) {
            // no results for this id - just return null
            return null;
        }
    }


// We have to use the 'query' method instead of 'queryForObject' when we expect
     // to get 0 or more objects back.
     // Note that we use the same RowMapper for single Students and List of Students
    @Override
    public List<Student> getStudentsByLastName(String lastName) {
        return jdbcTemplate.query(SQL_SELECT_STUDENT_BY_LAST_NAME, new StudentMapper(),
                            lastName);
    }

    // Deletes, updates and inserts all use the JdbcTemplate 'update' method
    @Override
    public void removeStudent(int studentId) {
        jdbcTemplate.update(SQL_DELETE_STUDENT, studentId);
    }

    // Deletes, updates and inserts all use the JdbcTemplate 'update' method
    @Override
    public void updateStudent(Student student) {
        jdbcTemplate.update(SQL_UPDATE_STUDENT,
                student.getFirstName(),
                student.getLastName(),
                student.getMajor(),
                student.getStreet(),
                student.getCity(),
                student.getState(),
                student.getZipcode(),
                student.getStudentId());
    }
```

```
// This nested class maps a row from the database into a Student object
    private static final class StudentMapper implements RowMapper<Student> {

        @Override
        public Student mapRow(ResultSet rs, int i) throws SQLException {
            Student student = new Student();
            student.setStudentId(rs.getInt("id"));
            student.setFirstName(rs.getString("first_name"));
            student.setLastName(rs.getString("last_name"));
            student.setMajor(rs.getString("major"));
            student.setStreet(rs.getString("street"));
            student.setCity(rs.getString("city"));
            student.setState(rs.getString("state"));
            student.setZipcode(rs.getString("zipcode"));
            return student;
        }
    }
}
```

Spring Configuration

Our new components and our desire to communicate with our MySQL database require changes to our applicationContext.xml Spring configuration files (app and test). Add the following bean definitions to both your applicationContext.xml files. After these entries are added, remove the existing bean definition for the dao from each file.

```
<tx:annotation-driven />

<!-- Define Data Source - this defines the connection to the database -->
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
     <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
     <!-- test-applicationContext.xml should point to StudentRoster_test -->
     <!-- value="jdbc:mysql://localhost:3306/StudentRoster_test" /> -->
     <property name="url"
               value="jdbc:mysql://localhost:3306/StudentRoster_test" />
     <property name="username" value="root" />
     <property name="password" value="root" />
     <property name="initialSize" value="5" />
     <property name="maxActive" value="10" />
</bean>

<!-- Define Transaction Manager - just need to define it so we can inject it -->
<!-- The Transaction Manager needs a Data Source - it uses setter injection  -->
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property name="dataSource" ref="dataSource" />
</bean>
```

```xml
<!-- Define JdbcTemplate - just need to define it so we can inject it -->
<!-- The JdbcTemplate needs a Data Source - it uses setting injection -->
<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.JdbcTemplate">
      <property name="dataSource" ref="dataSource" />
</bean>

<!-- Define the database DAO - it needs a JdbcTemplate via setting injection -->
<bean id="dao"
      class="com.swcguild.studentrostermaven.dao.StudentDaoDbImpl">
      <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

Unit Tests

Updating the unit tests for the DAO is left as an in-class exercise. There are several things that change when going against an actual database. Make sure you note the problems that you encounter as the unit tests are updated - it is very likely that you will run into these issues in future projects.