SENG2050 – Lab Week 7

This laboratory will cover how we can achieve persistence in Java web applications. In this workshop we will:

- Revise creating SQL Server Databases;
- Configure a JDBC data source for web applications;
- Utilise a database in a web application.

Note: This laboratory assumes you have Microsoft SQL Server installed on your local machine. Please see the associated videos for help.

1 Persistence in Web Applications

Java specifies a standard method of interacting with databases through JDBC: Java DataBase Connectivity. JDBC allows us to interact with any SQL database regardless of vendor. So far, we have used the Microsoft SQL server, however other vendors include MySQL, Postgresql or Oracle. Each of these are distinct products, with different methods for interacting with them in code.

JDBC simplifies this by specifying a simple set of abstractions for communicating with a database server. However, every database server 'speaks its own language'. Because of this, we need a JDBC driver to translate from the JDBC method of database interaction, to a database's native method of interaction; essentially acting as a translator.

There is a standard 3-step process for using a database in a web applications:

1. Create your schema.
2. Define your datasource.
3. Query the database.

2 Database Schema

The schema describes the structure of our data. In this lab we will have a simple database which stores a list of movies. Each movie will have a list of reviews associated with it.

First, create your database:

```
CREATE DATABASE [seng2050_lab06];
USE [seng2050_lab06];
```

Second, create your *movies* table. The syntax to create a table is:

```
CREATE TABLE table_name (attributes);
```

Where attributes is a comma-separated list of pairs: attribute name attribute type, attribute name attribute type, ...

Our movies table will store a: title, year, and url.

```
CREATE TABLE [movies] (
    [title] VARCHAR(80),
    [year] INT,
    [url] VARCHAR(256)
);
```

If the table already exists, or if you make a mistake and want to try again, you may want to delete the old table. The command for this is:

```
DROP TABLE table_name;
```

Third, we need to insert some data. The command to insert a new row is:

```
INSERT INTO table_name VALUES ( values );
```

Where values is a comma-separated list of values – one value for each attribute of the table, in the same order as they were specified by the CREATE TABLE command. String values should be enclosed in single quotes ('value'). Numeric values should be given without quotes.

Try inserting a movie into the table:

```
INSERT INTO movies
VALUES ('Akira', 1988, 'http://www.imdb.com/Title?Akira+(1988)');
```

If successful, MSSQL should respond saying that 1 row was affected. To check if the movie data is correct in the table, execute:

```
SELECT * FROM movie;
```

MSSQL should display the table with the one row of data.

Next, try inserting some movies of your own. Just copy the above template, but change the values. I have uploaded a movies.sql file to Blackboard in case you don't watch many movies.

Fourth, we need a second table to hold the movie reviews. This table (call it reviews) should contain 5 rows:

- title: A string of up to 80 characters (VARCHAR(80))
- year: INT
- reviewer: A string of up to 20 characters
- score: INT
- url: Another string of up to 256 characters

Enter the SQL command necessary to create this table. Now you may download the reviews.sql file from blackboard and run it to INSERT some reviews. You may also like to INSERT your own reviews.

Fifth, we need to create a database user to query the database from our applications. Database usernames and passwords are stored in plain text, so we need a different username + password that can be shared. A user is referred to as a *login* and needs to be granted privileges to a database.

```
CREATE LOGIN jdbcUser
WITH PASSWORD = 'mySecur3Passw0rd!';

CREATE USER jdbcUser
FOR LOGIN jdbcUser;

GRANT SELECT, INSERT, UPDATE, DELETE
TO jdbcUser;
```

2.1 Queries

Now that the database contains some data, you can start querying. You have already used the simplest query (SELECT FROM table name;), but you can make much more complex (and useful) ones. Revise your SQL skills by writing the following queries:

1. Select just the title and year of a movie: SELECT title, year FROM movie;
2. Select just the title and url of only those movies made in 1996: ... FROM movie WHERE year = 1996;
3. Select all attributes of movies made between 1995 and 1998 inclusive: ... WHERE year BETWEEN 1995 AND 1998;
4. Sort the previous query by year: ... BETWEEN 1995 AND 1998 ORDER BY year;
5. Count the number of movies in each year: SELECT year, COUNT(*) FROM movie GROUP BY year ORDER BY year;
6. Select the title, reviewer and score of each review tuple
7. Select all reviews with a score of 8 or more
8. Select all reviews by 'Richard'
9. Select all movies and reviews: SELECT m.*, r.* FROM movie m, review r;
10. You will notice that the previous query joins all movies with all reviews, even when the title and year do not match. To combine the movie and review tables so that only matching movies are pairs, you use: ... WHERE m.title = r.title AND m.year = r.year;
11. Select all movie titles with their corresponding review urls: SELECT m.title, r.url ...
12. Select all movie and review information (but do not double-up the title and year), for all movies made between 1995 and 1998, ordering them by year.

2.1.1 Directed Task - Normalisation

Normalisation is an important aspect of relational database design. So far, our database is not normalised, and their is no referential integrity.

Revise your SQL skills by normalising this database into BCNF (Boyce Codd Normal Form). At a minimum, add primary keys on the movies and reviews tables, then enforce the movies-reviews 1:many relationship through a foreign key. Building upon this, you may like to normalise the year and reviewer columns of the reviews table. Is there anything else you can do to normalise this database?

3 JDBC DataSource

Utilising raw JDBC, to interact with a database server we:

- Create a connection string (url)
- Direct JDBC to create a connection
- Create or prepare statements
- Execute queries
- Process the results

In Java web applications, using JDBC is slightly different as the application server (Tomcat) manages the connection. This means we no longer need to create a connection to the database server as this provided to us by Tomcat, through the definition of a DataSource.

A DataSource represents the SQL server we're interacting with. It allows for the creation of JDBC connections, that we can utilise in our applications. Interally, the data source pools connections to enable them to be reused between requests. This is as creating connections is a resource-intensive process, and reusing them provides greater efficiency.

3.1 Tomcat context.xml

Tomcat allows for the definition of a DataSource in the context.xml file. This file allows us to define scoped resources that can be consumed by our application.

First, create the context.xml file in:

- apache-tomcat/webapps/[APPNAME]/META-INF/context.xml

Second, define our DataSource resource:

```
<Context>
    <Resource name="jdbc/MyConnection"
        type="javax.sql.DataSource"
        maxActive="10"
        maxIdle="5"
        username="jdbcUser"
        password="mySecur3Passw0rd!"
        driverClassName="com.microsoft.sqlserver.jdbc.SQLServerDriver"
        url="jdbc:sqlserver://localhost[\INSTANCENAME];databaseName=seng2050_lab06" />
</Context>
```

Note: The name attribute is the JNDI name of our DataSource. We need to know this name in order to retrive the DataSource later. The username and password has to match that defined in Section 2. The url has to specify the same databaseName that we created in Section 2. Watch the associated MSSQL GUI Install video to find out your INSTANCENAME.

Third, we need to provide Tomcat a copy of our MSSQL JDBC driver. This lets JDBC talk to the MSSQL server. Copy the MSSQL JDBC driver on blackboard to:

- apache-tomcat/webapps/[APPNAME]/WEB-INF/lib/

Fourth, we will create a bean to retrieve this DataSource that will allow us to lease connections. Note, this bean needs to be in a package.

```
import javax.sql.*;
import java.sql.*;
import javax.naming.*;

public class ConfigBean {
    private static final DataSource dataSource = makeDataSource();
    private static DataSource makeDataSource() {
        try {
            InitialContext ctx = new InitialContext();
            return (DataSource) ctx.lookup("java:/comp/env/jdbc/MyConnection");
        } catch (NamingException e) { throw new RuntimeException(e); }
    }
    public static Connection getConnection() throws SQLException {
        return dataSource.getConnection();
    }
}
```

4 Using JDBC

In this task you will access your database from inside a Java Bean using JDBC.

Create a Java Bean called Movie.java. This class will represent a single record stored in the movies table in the databse. This bean will have one field per table attribute. A point of interest is that JDBC will convert the SQL data types to native Java objects. This mapping is specific to each JDBC driver, so it can vary. However, there is a standard set of representations of SQL types. This url may be of help to identify what SQL types are mapped to Java types:

- `https://docs.microsoft.com/en-us/sql/connect/jdbc/understanding-data-type-conversions?view=sql-server-ver15`

In the Movie class, add a static method that queries the database for all movies and returns them in a List. This method should work as follows:

1. Get a connection from the class created at the beginning of this task
2. Create a statement
3. Execute the statement with the SELECT query

4. The above step yields a ResultSet (`http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/ResultSet.html`) that you can extract your results from

5. Process the ResultSet into a List

6. Return the list

Have a go at doing this yourself by utilising the docs and the examples in the lecture slides. A sample is provided on blackboard. Note, our implementation may differ to yours. This is expected, but doesn't matter as long as it works!

To make sure it works, create a JSP that uses this bean and prints the values returned from the getAllMovies() method. You can call this method and iterate through the results with:

```
<%@page import="pkg.Movie" %>       <!-- Update this for your application! -->
<%@page import="java.util.List" %>  <!-- You may need more imports! -->
<% List<Movie> movies = Movie.getAllMovies(); %>
<ul>
    <% for (Movie movie : movies) { %>
        <li>
            <%= movie.getTitle() %>
            ... more fields ...
        </li>
    <% } %>
</ul>
```

4.1 Directed Tasks

4.1.1 Viewing Movie Reviews

In the previous directed task we normalised our database. At a minimum you should have created a primary key (id) for the movie and reviews table. In this task we utilise the ids to be able to query each movie's reviews. Complete the following tasks:

1. For each movie listed in our JSP page, create a hyperlink to a new page that will display this movie. In this link, pass in the 'id' of the movie.

2. In the second JSP page, display all the fields of this movie. You will need to implement a new static method on the Movie class to retreive a **single** Movie from the database.

3. In this page, print a list of reviews that have been left for the movie.

4.1.2 Creating Movies + Reviews

Extend your application to enable the creation of movies. You will need to create a new JSP form page allowing the user to specify the movie attributes. Create a servlet to handle this form input, and insert the movie into the database.

Further extend this application to allow the creation of reviews. Consider how you will pass in the id of the movie the review is being created for. In such cases, we typically pass in the id of the movie in the URL, and include a hidden input element in the form to specify the movie id upon review creation.