LS/EECS

Building E-commerce Applications

Integrating SQLite into your Student Information System project.

Objectives

By the end of this lab, students should be able to:

- Install and configure SQLite for use with Java applications.
- Understand basic SQL operations: CREATE, INSERT, SELECT, UPDATE, and DELETE.
- Modify the Student Information System to store data persistently in an SQLite database.
- Implement a Listener to count the total number of API calls.

Introduction to SQLite

SQLite is a software library that provides a relational database management system. A prominent feature of SQLite is its zero-configuration, meaning no setup or administration is required for it to run. SQLite is not a client-server database engine—rather, it's embedded into the end program.

Features of SQLite:

- Serverless, zero-configuration, and transactional.
- Compact size with minimal setup and administration.
- Public domain source code with a large community.
- Supports databases up to 2 terabytes in size.
- Provides a standalone command-line interface (CLI) client that can be used to administer SQLite databases.

When to Use SQLite:

- Applications needing an internal database or embedded database.
- Prototyping applications where the simplicity of SQLite can speed up development.
- Situations where setup and administration of a larger database system are not practical.

Introduction to SQL

SQL (Structured Query Language) is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS). SQL offers a more robust system for managing and querying data than its non-relational counterparts.

Key Concepts of SQL:

- **Tables**: Data is stored in tables in SQL, where each table has a different structure with a fixed number of columns and a variable number of rows.
- **Queries**: SQL's main function is to query data. The most basic query you could write would be to select data from a table.
- **Schema**: The structure that defines the organization of data, which includes tables, the relationships between tables, and the data type of each column.
- **Keys**: Used to define relationships between tables (foreign keys) and uniqueness (primary keys) in tables.

Basic SQL Commands:

- **SELECT**: Retrieves data from one or more tables.
- **INSERT**: Adds new records into a table.
- **UPDATE**: Modifies existing records in a table.
- **DELETE**: Removes records from a table.
- **CREATE**: Sets up new tables, views, or stored procedures.
- **ALTER**: Modifies an existing database object, like a table.

Introduction

In this lab, you will integrate SQLite into the Student Information System developed in previous labs. SQLite is a C library that provides a lightweight, disk-based database. It doesn't require a separate server process and allows access to the database using a nonstandard variant of the SQL query language. You will install SQLite, configure your Java application to connect to an SQLite database, and perform basic SQL operations to store and retrieve student data.

Task 1: Install SQLite

1. **Download SQLite:** Visit the official SQLite download page and download the precompiled binaries for your operating system.
2. **Install SQLite:** Follow the installation instructions specific to your operating system.
3. **Verify Installation:** Open a command prompt or terminal and run `sqlite3`. You should see the SQLite command prompt.

Task 2: Create Database Schema Using SQLite Studio

SQLite Studio is a robust SQLite database management tool. It provides a user-friendly interface to perform various database management operations, including creating tables and defining schemas.

1. Download and Install SQLite Studio: If you haven't installed SQLite Studio, download it from the official website and follow the installation instructions.
2. Create a New SQLite Database:
   a. Open SQLite Studio.

b. Click on the "Add a database" icon.
c. Choose a name and location for your database file (e.g., `students.db`).
d. Click "OK" to create the database.

3. Define the Schema:
   a. With your new database selected, right-click in the Database Structure tab and select `Create a table`.
   b. Name the table `students`.
   c. Add the following columns:
      - `id` (type: `INTEGER`, constraints: `PRIMARY KEY`, `AUTOINCREMENT`)
      - `name` (type: `TEXT`, constraints: `NOT NULL`)
      - `age` (type: `INTEGER`, constraints: `NOT NULL`)
      - `major` (type: `TEXT`, constraints: `NOT NULL`)
   d. Save the table.
   e. Open an SQL editor and qxplore SQL statements in DLL tab.
   f. You might want to copy the SQL statement from the DLL into the SQL editor

Example SQL Statements.

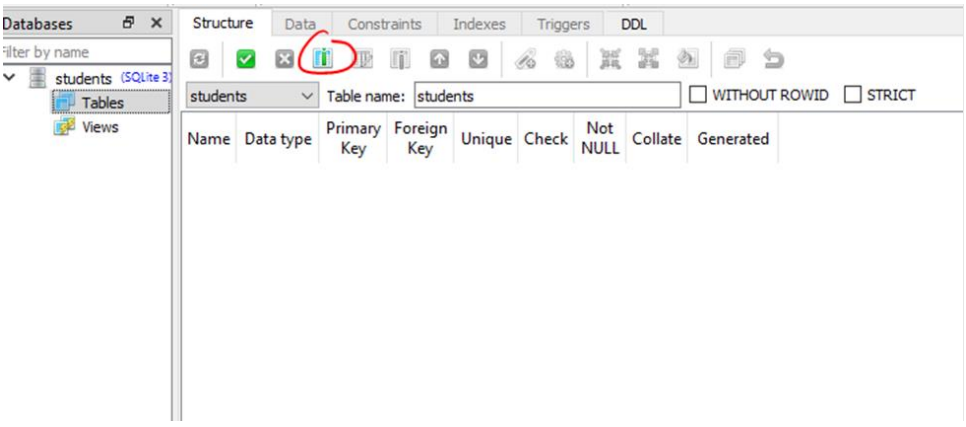
In the SQL editor, execute the following statements:

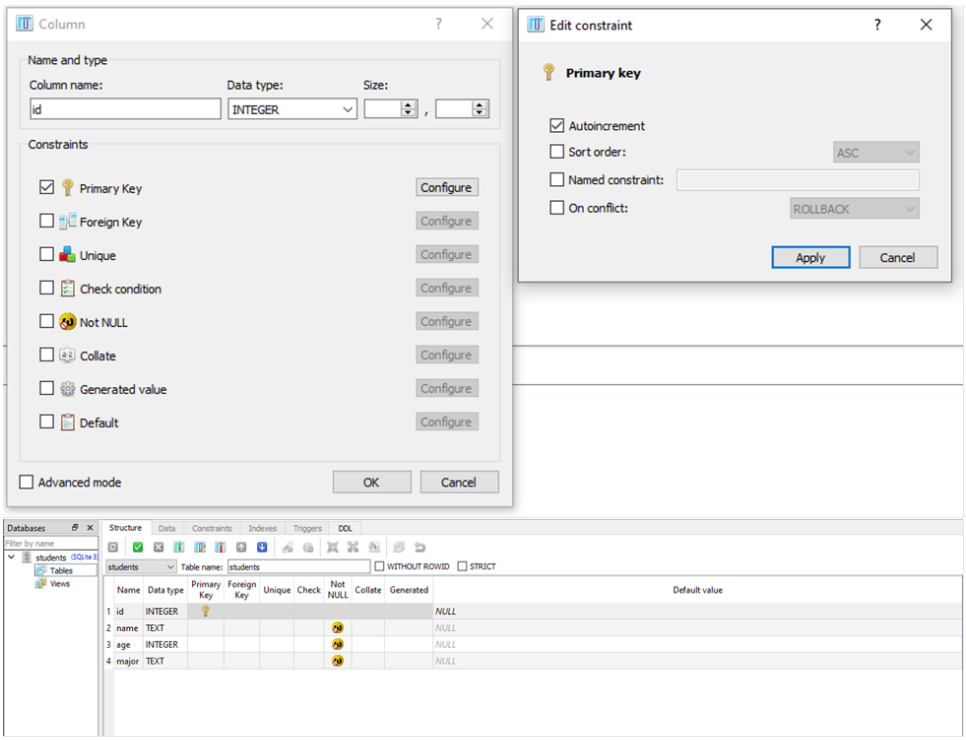
```
CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER NOT NULL,
    major TEXT NOT NULL
);

insert into students (id, name, age, major) values (5, "Vinod", 22, "EE");
insert into students (id, name, age, major) values (1, "Mary", 22, "CS");

select * from students;
```

Example on the SQLite Studio UI:





Now, that you have a simple database, you can close the database connection.

Task 3: Configure Application to Use SQLite

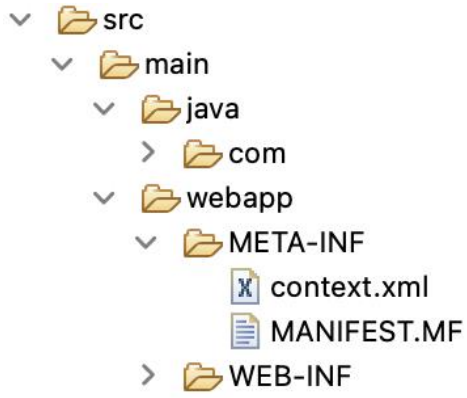
1. **Add SQLite JDBC Driver:** Add the following dependency to your `pom.xml` file:

```
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.34.0</version> <!-- use the latest version available -->
</dependency>
```

2. Add a context.xml file to your meta-info directory, with the following content but with the correct path for your database. Below, */Users/ML/Documents/SIS-2023 is the path for my database on my MAC OS system.*

```
<?xml version="1.0" encoding="UTF-8"?>
<Context privileged="true" reloadable="true">
       <WatchedResource>WEB-INF/web.xml</WatchedResource>
       <Manager pathname="" />
       <Resource name="jdbc/EECS"
       factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
       type="javax.sql.DataSource" driverClassName="org.sqlite.JDBC"
       url="jdbc:sqlite:/Users/ML/Documents/SIS-2023/students.db"/>
       <ResourceLink global="jdbc/EECS" name="jdbc/EECS"
       type="javax.sql.DataSource" />
```

```
</Context>
```



3. **Establish Connection:** Create a `DatabaseConnection` class to establish and return a connection to the SQLite database.
   Example `DatabaseConnection` class:

```
import java.sql.Connection;
import java.sql.SQLException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

public class DatabaseConnection {

    public static Connection connect() {
        Connection conn = null;

        try {
            // Obtain our environment naming context
            Context initCtx = new InitialContext();
            Context envCtx = (Context) initCtx.lookup("java:comp/env");

            // Look up our data source
            DataSource ds = (DataSource) envCtx.lookup("jdbc/EECS");

            // Allocate and use a connection from the pool
            conn = ds.getConnection();
        } catch (SQLException | NamingException e) {
            System.out.println(e.getMessage());
        }

        return conn;
    }
}
```

Task 3: Update DAO Class for SQLite

1. **Modify `StudentDAO` Class**: Update the `StudentDAO` class to use SQL queries and the `DatabaseConnection` class for CRUD operations.
2. **Handle SQL Exceptions:** Ensure that SQL exceptions are properly handled.

Example StudentDAO Class:

```
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class StudentDAO {

    public List<Student> readAll() {
        String sql = "SELECT id, name, age, major FROM students";
        List<Student> students = new ArrayList<>();

        try (Connection conn = DatabaseConnection.connect();
             Statement stmt  = conn.createStatement();
             ResultSet rs    = stmt.executeQuery(sql)){

            while (rs.next()) {
                Student student = new Student();
                student.setId(rs.getInt("id"));
                student.setName(rs.getString("name"));
                student.setAge(rs.getInt("age"));
                student.setMajor(rs.getString("major"));
                students.add(student);
            }
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
        return students;
    }

    public void create(Student student) {
        //we use prepared statements, Q: why?
        String sql = "INSERT INTO students(name, age, major) VALUES(?,?,?)";

        try (Connection conn = DatabaseConnection.connect();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setString(1, student.getName());
            pstmt.setInt(2, student.getAge());
            pstmt.setString(3, student.getMajor());
            pstmt.executeUpdate();
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }

    public Student read(int id) {
        // Use prepared statements
        String sql = "SELECT name, age, major FROM students WHERE id = ?";
        Student student = null;

        try (Connection conn = DatabaseConnection.connect();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {
            // Set the corresponding parameter
            pstmt.setInt(1, id);
```

```
            // Execute the query and get the result set
            try (ResultSet rs = pstmt.executeQuery()) {
                // Check if a result was returned
                if (rs.next()) {
                    student = new Student();
                    // Set the properties of the student object
                    student.setId(id);
                    student.setName(rs.getString("name"));
                    student.setAge(rs.getInt("age"));
                    student.setMajor(rs.getString("major"));
                }
            }
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }

        return student;
    }

    public void update(int id, Student student) {
        //use prepared statments
        String sql = "UPDATE students SET name = ?, age = ?, major = ? WHERE id =
?";

        try (Connection conn = DatabaseConnection.connect();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {
            // Set the corresponding parameters
            pstmt.setString(1, student.getName());
            pstmt.setInt(2, student.getAge());
            pstmt.setString(3, student.getMajor());
            pstmt.setInt(4, id);
            // Update the student record
            pstmt.executeUpdate();
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }

    public void delete(int id) {
        String sql = "DELETE FROM students WHERE id = ?";

        try (Connection conn = DatabaseConnection.connect();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {
            // Set the corresponding parameter
            pstmt.setInt(1, id);
            // Delete the student record
            pstmt.executeUpdate();
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Explanation

- `readAll` **Method:** This method constructs an SQL `SELECT` statement and executes it to retrieve all student records from the `students` table in the database. It initializes an empty list of `Student` objects. Then, it iterates through each record in the result set, creates a `Student` object, sets its fields with the values from the current record using the `getInt` and `getString` methods of the `ResultSet` object, and adds it to the list. Finally, it returns the list of `Student` objects.

- `create` Method: This method takes a `Student` object as a parameter, constructs an SQL `INSERT` statement, and executes it to insert a new student record into the `students` table in the database. The `?` placeholders in the SQL statement are replaced with the actual values from the `Student` object using the `setString` and `setInt` methods of the `PreparedStatement` object.

- `update` Method: This method takes a `Student` object as a parameter, constructs an SQL `UPDATE` statement, and executes it to update the student record with the specified id in the database. The `?` placeholders in the SQL statement are replaced with the actual values from the `Student` object using the `setString`, `setInt` methods of the `PreparedStatement` object.

- `delete` Method: This method takes an id as a parameter, constructs an SQL `DELETE` statement, and executes it to delete the student record with the specified id from the database. The `?` placeholder in the SQL statement is replaced with the actual id value using the `setInt` method of the `PreparedStatement` object.

To test the application,

-Update Maven project, Run as Maven Install and Run on Server

-Use the curl commands from the previous lab to test read/create/update/delete rest methods

Task 4: Implement a Listener for Counting API Calls

- **Create a Listener Class:** Create a new Java class that implements `javax.servlet.ServletContextListener`.
- **Initialize Counter:** Initialize a counter in the `contextInitialized` method.
- **Increment Counter:** Increment the counter for each API call. You might need to use a `Filter` or modify the existing controller methods to increment the counter on each call.
- **Display Counter:** Optionally, create a method to retrieve and display the current counter value.

Below is a basic example of how you might implement this:

1. Add the Jakarta Dependency for Servlets in your pom.xml

```
<dependency>
 <groupId>jakarta.servlet</groupId>
 <artifactId>jakarta.servlet-api</artifactId>
 <version>6.0.0</version>
 <scope>provided</scope>
</dependency>
```

2. **Create Listener Class**
   Create a new Java class named `ApiCallCounterListener`.

```
import jakarta.servlet.ServletContext;
import jakarta.servlet.ServletContextEvent;
import jakarta.servlet.ServletContextListener;
import jakarta.servlet.annotation.WebListener;

@WebListener
public class ApiCallCounterListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext ctx = sce.getServletContext();
        ctx.setAttribute("apiCallCounter", 0);
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        // You can add code here if you need to do something when the
application is stopped
    }
}
```

3. **Increment Counter on API Call**
   You might need a `Filter` to increment the counter on each API call. Below is an example of a `Filter` that increments the counter:

```
import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.FilterConfig;
import jakarta.servlet.annotation.WebFilter;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;

@WebFilter("/*")
public class ApiCallCounterFilter implements Filter {

    @Override
```

```
        public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) {
            ServletContext ctx = request.getServletContext();
            Integer counter = (Integer) ctx.getAttribute("apiCallCounter");
            counter++;
            ctx.setAttribute("apiCallCounter", counter);

            chain.doFilter(request, response);
        }

        // Implement init and destroy methods if necessary
    }
```

Task 5: Add API Method to Return Total API Calls

1. **Create API Method:** Add a new method to your controller class that will return the total number of API calls.
2. **Retrieve Counter:** Retrieve the counter value from the `ServletContext`.
3. **Return Counter:** Return the counter value as a response.

Below is a basic example of how you might implement this in your controller class:

Add API Method in Controller

In your controller class (`StudentController`), add a new method to return the total number of API calls:

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.core.Response;

@Path("/students")
public class StudentController {

    @GET
    @Path("/totalApiCalls")
    public Response getTotalApiCalls() {
        Integer counter = /* retrieve counter from application context. Hint: You
can inject the ServletContext in your REST method, similar to how you inject
HttpServletRequest in doGet in the previos labs.  */;
        if (counter == null) {
            counter = 0; // Handle the case where the counter attribute is not set
        }
        return Response.ok(counter.toString()).build();
    }
}
```

Testing

You can test the endpoint using the curl command-line tool. (You can also use Postman from the previous lab to call the API). Open a terminal or command prompt and run the following curl command:

```
curl http://localhost:8080/your-app-context/api/totalApiCalls
```

Create a Student

```
curl -X POST -H "Content-Type: application/json" -d '{"name":"John", "age":20,
"major":"Computer Science"}' http://localhost:8080/YourApp/students
```

> Windows CMD can't read single quotes, so you may have to use \" to replace the single and quotes in your data payload

```
curl -X POST -H "Content-Type: application/json" -d "{\"name\":\"John\",
\"age\":20, \"major\":\"Computer Science\"}"
http://localhost:8080/YourApp/students
```

List All Students

```
curl -X GET http://localhost:8080/YourApp/students
```

Get a Specific Student

```
curl -X GET http://localhost:8080/YourApp/students/1
```

Update a Student

```
curl -X PUT -H "Content-Type: application/json" -d '{"name":"John Doe", "age":21,
"major":"Software Engineering"}' http://localhost:8080/YourApp/students/1
```

Delete a Student

```
curl -X DELETE http://localhost:8080/YourApp/students/1
```