SOFT3202

Adam Ghanem

April 29, 2022

Contents

1 Tutorial 6 EXPORT

1.1 Databases

1.1.1 Relation to Tasks

This is extremely important, although we do not go into too much detail, it will be used in your assignments:

- Task 3 (add DB to task 2)
- Major Project (need to get a credit)

1.1.2 Description and CRUD

Database (DB) have reached a point of ubiquity and even though this is about software design it is integral to go over databasing and managing them. To start off, the core philosophy surrounding operations of DBs is **CRUD**:

Create

- we can **INSERT** (SQL) or we can **PUT** (RESTful)

Read

- we can **SELECT** (SQL) or we can **GET** (RESTful)

Update

- we can **UPDATE** (SQL) or we can **PUT** (RESTful)

Delete

- we can **DELETE** (SQL) or we can **DELETE** (RESTful)

1.1.3 Relational DB and SQLite

With a relation DB we have the data is displayed within tables in the form of relations and the commands we use work on the data in this tabular format. The key point is that there is a structure to this data and along with column names, data type and rules (PRIMARY KEYs etc).

SQLite is quite similar to other SQL query languages but has some unique difference and use cases:

- it usually only has a single file
- very minimal setup process
- SQLite is written in `C`
- does not have a separate server
- only supports a few data types

SQL rundown (also we can try this out at `https://sqliteonline.com/`):

SELECT, FROM, WHERE Assuming we have a DB of:

`SELECT * FROM Shows`:

Show	Rating
Westworld	7.0
Loki	6.0
Seinfeld	9.2
Office (UK)	8.9

Usually the format of this is:

```
-- We get the columns we want to select
SELECT show
-- We are retrieving from the Shows database
FROM Shows
-- We are selecting only shows that match this
WHERE rating < 8.5;
```

The output will be this:

Show
Westworld
Loki

INSERT Inserting is quite straightforward:

Show	Rating
Westworld	7.0
Loki	6.0
Seinfeld	9.2
Office (UK)	8.9

```
INSERT INTO Shows (show, rating)
VALUES
('Breaking Bad', 9.2),
('Courage the Cowardly Dog', 8.5),
('Ed Edd and Eddy', 8.5),
('Teen Titans', 8.4);
```

As an exercise try practicing commands by inserting your favourite TV shows.

What happens if I try to add the same entry in? In this case with our table, what should be our unique identifier?

UPDATE Using the same database to update:

```
UPDATE shows
SET rating = 9.5
WHERE show = 'The Wire' OR show = 'Ren and Stimpy'
OR show.director_id IN
    (SELECT director_id
    FROM Directors
    WHERE director = 'Nolan, Christopher');
```

DELETE Deleting follows a similar logic to `SELECT`:

```
DELETE FROM Shows
WHERE LOWER(show) LIKE 'break%'
```

JOIN Sometimes we don't just want to deal with one table, what if we wanted to find the actors who starred in a movie with a high rating. Well we can connect the two tables based on some common value. In this case assume `Roles` contains actor names and the films they starred in. The databse, `Film`, contains the film name and the rating of that film.

```
SELECT R.first_name, F.rating
FROM Roles R, F Film
WHERE F.rating >= 8.3 AND R.film = F.film;

SELECT R.first_name, F.rating
FROM Roles R INNER JOIN Film F ON (R.film = F.film)
WHERE F.rating >= 8.3
```

Even better however, we can make use of a performance improvement by using the `ON` command to filter out rows before we join them up:

```
SELECT R.first_name, F.rating
FROM Roles R INNER JOIN Film F ON (R.film = F.film AND F.rating >= 8.3)
```

CREATE We can create the table using the `CREATE TABLE` command:

```
CREATE TABLE IF NOT EXISTS Directors (
-- Remember how we talked about unique identifiers.
-- This PRIMARY KEY is our unique restriction on the database
-- It must be UNIQUE and NOT NULL
director_id INTEGER PRIMARY KEY,
-- contains a varying character of length 20
first_name TEXT NOT NULL,
last_name TEXT NOT NULL,
earnings REAL
)
```

Data Types We have data types of:

- `TEXT` which stores all data as a text string
- `BLOB` this is the data as it was inputted, actual raw data
- `REAL` floating point numbers and can be sub divided down
- `INTEGER` these are our standard integer numbers $i \in \mathbb{Z}$
- `NULL` the standard `NULL` value which is it's own type

Be careful, in SQLite there is no real restriction on a column type and in a way they are treated as strings. So if I perform a command to change an integer using divide and it results in a REAL value, it will update the row's value to a REAL type. Make sure to `CAST` or use `CONSTRAINTS`.

1.1.4 Connecting to a Database

For our `build.gradle` we have to add an implementation dependency:

```
implementation org.xerial:sqlite-jdbc:3.36.0.3
```

- this does not require a separate process running to interact with

To connect to our databse we use JDBC which is the Java Database Connectivity:

```
import java.sql.Connection;
import java.sql.ResultSet;
```

```
public class SQLExample {

  private String dbName;
  // Should be something like "jdbc:sqlite:"
  private String dbURL;

  public void initDB() {
    File db = new File(dbName);
    if (db.exists()) {
      System.err.println("Invalid, already exists");
      return;
    }

    try (Connection ignored = DriverManager.getConnection(dbURL)) {
      System.out.println("Database has been created.")
    } catch (SQLException e) {
      System.err.println(e);
      System.exit(-1);
    }
  }
}
```

To create statements we follow the same logic as shown in the SQL section. The idea is we just create `SQLite` queries in the form of strings which we then pass through to the SQLite.

```
public class SQLExample {
  public void createShows() {
    String createShowTableSQL =
      """
      CREATE TABLE IF NOT EXISTS Shows (
                                        show_id integer PRIMARY KEY,
                                        film_name text NOT NULL,
                                        rating REAL NOT NULL,
                                        );
    """;


      try (Connection conn = DriverManager.getConnection(dbURL);
```

```
          Statement statement = conn.createStatement()) {
        statement.execute(createShowTableSQL);
        System.out.println("Created the Show table.");
      } catch (SQLException e) {
        System.out.println(e.getMessage());
        System.exit(-1);
      }
  }


  public void addShowsToTable() {
    // Then we can insert into the table and add some rows
    String addShowsToTable =
      """
      INSERT INTO shows(id, film_name, rating) VALUES
      ('Breaking Bad', 9.2),
      ('Courage the Cowardly Dog', 8.5),
      ('Ed Edd and Eddy', 8.5),
      ('Teen Titans', 8.4);
    """;

      try (Connection conn = DriverManager.getConnection(dbURL);
           Statement statement = conn.createStatement()) {
        statement.execute(addShowsToTable);
        System.out.println("Inserted some shows in.");
      } catch (SQLException e) {
        System.out.println(e.getMessage());
        System.exit(-1);
      }

  }
}
```

Moreover we can have something known as `FOREIGN KEYS`, these mean that our data **references data in another table**. The issue with omitting this, is that if we delete something in a table it will affect tables that rely on it as a foreign key.

Querying in JDBC So we know how to perform `SELECT` statements in `SQLite`, but how is the data returned and more importantly how can we parse it correctly in Java. Well the first step after performing a `SELECT` query is to set the parameters and then execute the query. After executing the query we load the results, this operates in a similar fashion to `iterator` which we parse through statement by statement.

```
public class SQLExample {
  public void getShows(double minRating, boolean alphabetical) {
    String queryShowsWithMinRating =
      String.format("""
                    SELECT show_name, rating
                    FROM Shows
                    WHERE rating >= ?
                    ORDER BY show_name %s;
                    """,
                    (alphabetical) ? "ASC" : "DESC");
    // Above line is equivalent to:
    // if (alphabetical) {
    //   "ASC";
    // } else {
    //   "DESC";
    // }
    // C is the same as Java
    // Python is: "ASC" if EXPRESSION else "DESC"

    try (Connection conn = DriverManager.getConnection(dbURL);
         PreparedStatement preparedStatement =
         conn.prepareStatement(queryShowsWithMinRating)) {
      preparedStatement.setDouble(1, minRating);

      ResultSet results = preparedStatement.executeQuery();
      while (results.next()) {
        System.out.println(
                          results.getString("show_name") + " " +
                          results.getFloat("rating"));
      }
    } catch (SQLException e) {
```

```
      System.out.println(e.getMessage());
      System.exit(-1);
    }
  }
}
```

SECURITY WARNING - SQL Injection SQL injection attacks are caused by incorrect sanitisation of SQL queries. An SQL injection attack involves an attacker inserting malicious queries within the entry field for an SQL statement.

For instance, if a user wants to get user data from a prompt, instead of following the normal procedue they know the code uses a `SELECT` statement. It grabs a username in the form of:

```
// Some code above
String userQuery = "
  SELECT *
  FROM Users
  WHERE username = '" + username + "'
  AND password ='" + password + "';";
// Then send and connect to database
```

Where `username` is the text the user entered. But what if the user enters into the field: `'  OR '1'='1' --`

Let us disect this:

- we enter the username of `'` which escape quotes out
- then we add in an `OR` command to ensure we have some expression in there
- then we have, `--` this is the comment for SQL and hence removes then need to check for a passsword

The most famous example is to use: `'  OR '1'='1' --` this will ensure every entry is selected and comment out subsequent values.

Tips to help avoid this:

- Avoid trying to concatenate strings within your queries.

- map them to objects uses ORM
- sanitising strings
- parameterising statements (as shown in the examples)

1.1.5 Caching

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> – Phil Karlton

However this isn't the end of SQL and databases. Sometimes we need a performance boost, and where should we look towards. Well if we are requesting a users name consistently, then pinging a website can be very inefficient. Instead we want to make use of **caching** in order to store results. Instead of querying the database everytime we follow this process:

1. Get some data
2. If data has been stored previously then get from the cache in the local DB
3. else fetch from web
4. store data in the cache

In real scenarios you may want to have some condition that is dependent upon the volume of requests or some other metric, however, in this simple case we will not have to worry.

Moreover if we are storing items in a cache we must have a mechanism to purge out entries which are redundant or deemed unecessary. We can do this based on a number of factors:

- on timeout
- on versions
- on whether it has been updated
- on popularity metric (number of times queried or time since last query)
- or some other value which is deemed valid in that domain

1.2 Revisiting HTTP Connections and JSON

1.2.1 HTTP Headers

Just to revisit requests and responses to HTTP, the main topic here will go over headers. When we send requests over HTTP we have headers which can describe the `Content-Length` (number of bytes in the body) as well as the `Content-Type` (what is the data type we are sending through). This is not the only type of data available, within responses we can have the date, links, server as well as what we were accustomed to last week with the status codes.

But within some scenarios we need to pass information through that represents a token, or some form of authorisation such that we can access restricted data. This is usually performed within the header of our requests. There are different types of schemes:

- Bearer
- Digest
- Basic

Let us go through a example:

```
import java.util.Base64;
import java.nio.Charset.StanadardCharsets;

public class ExampleHTTP {
  public void getSecretDetails() {

    // We have to encode the username and password in base64 encoding
    String passwd = Base64.getEncoder().encodeToString("secretagent:hunter12")

    // Create the HTTP request, but in this case are adding a new post
    HttpRequest request = HttpRequest.newBuilder(new URI("https://.typicode.com/po
      .GET()
      // Make sure to include a Content-Type
      .header("Content-Type", "application/json")

      // And then our header for authorisation,
```

```
      // this is where we send through our token
      // it involves the Base64 encoding of "username:password"
      .header("Authorization", "Basic " + passwd)

      // As an example for another authorisation scheme
      // We can have different authorisation schemes in the form of bearer tokens
      .header("Authorization", "Bearer " + token)
      .build();
  }
}
```

1.2.2 Nested Classes

We can also use `Gson` to convert between nested classes. This is actually quite straightforward:

```
package t5;
import com.google.gson.Gson;
import java.util.List;
import java.util.LinkedList;

public class App {

  public static void nestedRequest() {
    // Create some nested class
    Post p1 = new Post(10, "Sample title", "This is a test post.");
    p1.addComment("new comment");

    // Create the GSON
    Gson gson = new Gson();
    // Convert to JSON
    String postJSON = gson.toJson(p1);
    System.out.println(postJSON);

    // Change it back to a Post object
    Post post = gson.fromJson(postJSON, Post.class);
    System.out.println(post);
```

```
    List<Post.Comment> commentList = post.getComments();
    System.out.println(commentList);


    // {
    //   "system" {
    //     "key": "value",
    //     "key": "value",
    //     "key": "value",
    //      }
    // }
  }

  public static void main(String[] args) {
    nestedRequest();
  }
}

class Post {
  private int userId;
  private String title;
  private String body;
  private List<Comment> comments;

  public Post(int userId, String title, String body) {
    this.userId = userId;
    this.title = title;
    this.body = body;
    this.comments = new LinkedList<>();
  }

  public List<Comment> getComments() {
    return this.comments;
  }

  public void addComment(String comment) {
    if (comment != null) {
      comments.add(new Comment(comment));
```

```
  }
}

public int getUserId() {
  return userId;
}

public String getTitle() {
  return title;
}

public void setTitle() {
  this.title = title;
}

public void setBody() {
  this.body = body;
}

public String getBody() {
  return body;
}

@Override
public String toString() {
  return "User ID: " + userId + " | " + title + "\n" + body;
}

public class Comment {
  private String content;

  public Comment(String content) {
    this.content = content;
  }

  @Override
  public String toString() {
    return content;
```

```
    }
  }
}
```

1.2.3 HTTPie QueryString Parameters

HTTPie is a command line mechanism to make HTTP requests with. Best resource to go over these sort of requests is:

`https://httpie.io/docs/cli/querystring-parameters`

This will assist when trying to send through parameters through the url.

1.2.4 Deserialising Classes Directly

When we want to deserialise our classes we can make use of two things, either you map this to a separate class which contains a that object to load them in. Or you can use JsonObject and JsonElement to fetch the object out of the json then map it to the class you want. The third way will be a bit more verbose and involves manually mapping each parameter back to the object class.

Say the json is this:

```
{
   "Stock":{
    "code": "XXX",
    "price": 1000000,
    "description": "ballin",
    "dividends": 1
    }
}
```

And a stock class:

```
public class Stock {
    protected String code;
    protected Double price;
    protected String description;
    protected Double dividends;
```

```
}
```

Then we are in a bit of a pickle, since we want just the inner {} to map back to a Stock. So what we do is:

```
JsonElement element = gson.fromJson(response.body(), JsonElement.class);
JsonObject jsonObj = element.getAsJsonObject();
Stock stock = gson.fromJson(jsonObj.get("Stock"), Stock.class);
```

The other way works as well (of having a class that holds a User object that you deserialise to), it is up to you.