# SQLite Database Integration

---

## Introduction to Databases in Web Applications

### a. Importance of Databases:

- **Data Persistence**: Store data across sessions and for multiple users.
- **Data Management**: Efficient retrieval, manipulation, and storage using structured queries.
- **Scalability and Integrity**: Handle growth with constraints that ensure consistent data.

### b. Overview of SQLite:

- **Definition**: SQLite is a lightweight, file-based RDBMS that doesn't require a server.
- **Advantages**:
    - **Simplicity**: Easy to set up and use.
    - **Portability**: Entire database is stored in a single file.
    - **Zero Configuration**: No server installation required.
    - **Integration**: Works seamlessly with Java using JDBC.

### c. When to Use SQLite:

- **Small to Medium Applications**: Ideal for apps with moderate storage needs.
- **Development and Testing**: Great for prototyping before scaling to robust databases.
- **Embedded Systems**: Perfect for lightweight, embedded applications.

---

# Setting Up SQLite in Java

**a. Prerequisites:**

- **SQLite JDBC Driver:**
  *A JDBC driver is needed to connect Java applications with SQLite databases.*

**b. Adding SQLite JDBC to Your Project:**

1. **Download the SQLite JDBC Driver:**
   - Visit SQLite JDBC Repository and download the latest `.jar` file.
2. **Include the JDBC Driver in Your Project:**
   - **IntelliJ IDEA:** Right-click on *your project* → *Open Module Settings* → *Libraries* → *Add* the downloaded `.jar` file.
   - **Eclipse:** Right-click on your *project* → *Build Path* → *Add External Archives* → *Select the downloaded* `.jar` file.

**c. Establishing a Connection to SQLite:**

**JDBC URL Format:**

```
String url = "jdbc:sqlite:database.db";
```

   - *This creates a new SQLite database named `database.db` in the project directory if it doesn't exist.*

**Example Connection Code:**

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnection {
    public static Connection connect() {
        String url = "jdbc:sqlite:database.db";
        Connection conn = null;
        try {
            conn = DriverManager.getConnection(url);
            System.out.println("Connected to SQLite");
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
        return conn;
    }
}
```

# 3. Essential SQLite Methods in Java

Here is a clear list of the most common JDBC methods and how to use them:

### a. Establishing a Connection

- **Purpose**: Connect to the SQLite database.

**Method**:

```
Connection conn =
DriverManager.getConnection("jdbc:sqlite:database.db");
```

- **Notes**: Always close the connection after use.

---

### b. Creating Statements

1. **Using `Statement`**:
    - **Purpose**: Execute static SQL queries without parameters.

**Example**:

```
Statement stmt = conn.createStatement();
stmt.addBatch("query");
stmt.executeBatch();
stmt.executeQuery("SELECT * FROM users");
stmt.executeUpdate("INSERT INTO users (name) VALUES ('John')");
stmt.close();
```

2. **Using `PreparedStatement`**:
    - **Purpose**: Execute parameterized SQL queries to prevent SQL injection.

**Example**:

```
String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, "John Doe");
pstmt.setString(2, "john.doe@example.com");
pstmt.executeUpdate();
pstmt.close();
```

---

### c. Executing Queries and Updates

1. `executeQuery` **(for SELECT)**:
    - **Purpose**: Execute a SQL `SELECT` statement and return results.

**Example**:

```
String sql = "SELECT * FROM users";
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
    System.out.println(rs.getString("name"));
}
rs.close();
stmt.close();
```

2. `executeUpdate` **(for INSERT, UPDATE, DELETE, DDL)**:
    - **Purpose**: Execute SQL statements that modify the database.

**Example**:

```
String sql = "UPDATE users SET email = 'new@example.com' WHERE id
= 1";
int rowsAffected = stmt.executeUpdate(sql);
System.out.println(rowsAffected + " rows updated.");
stmt.close();
```

---

### d. Setting Parameters in Prepared Statements

- **Purpose**: Dynamically set parameters in SQL queries.
- **Methods**:

**Set parameters:**

```
pstmt.setString(1, "John Doe");
pstmt.setInt(2, 42);
pstmt.setDouble(3, 99.99);
pstmt.setDate(4, java.sql.Date.valueOf("2023-01-01"));
```

### e. Processing Results

- `ResultSet.next()`:
    - **Purpose**: Move the cursor to the next row in the result set.

**Example**:

```
while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    System.out.println("ID: " + id + ", Name: " + name);
}
```

- **Retrieving Data**:

**Getting data**:

```
String name = rs.getString("name");
double salary = rs.getDouble("salary");
java.sql.Date hireDate = rs.getDate("hire_date");
```

---

### f. Closing Resources

- **Purpose**: Free up resources and avoid memory leaks:

```
rs.close();       // Close ResultSet
stmt.close();     // Close Statement
conn.close();     // Close Connection
```

**Best Practice**: Use **try-with-resources** to handle closing automatically:

```
try (Connection conn =
DriverManager.getConnection("jdbc:sqlite:database.db");
     Statement stmt = conn.createStatement()) {
     // Execute queries...
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```