Interacting with Database

Introduction to JDBC and ODBC

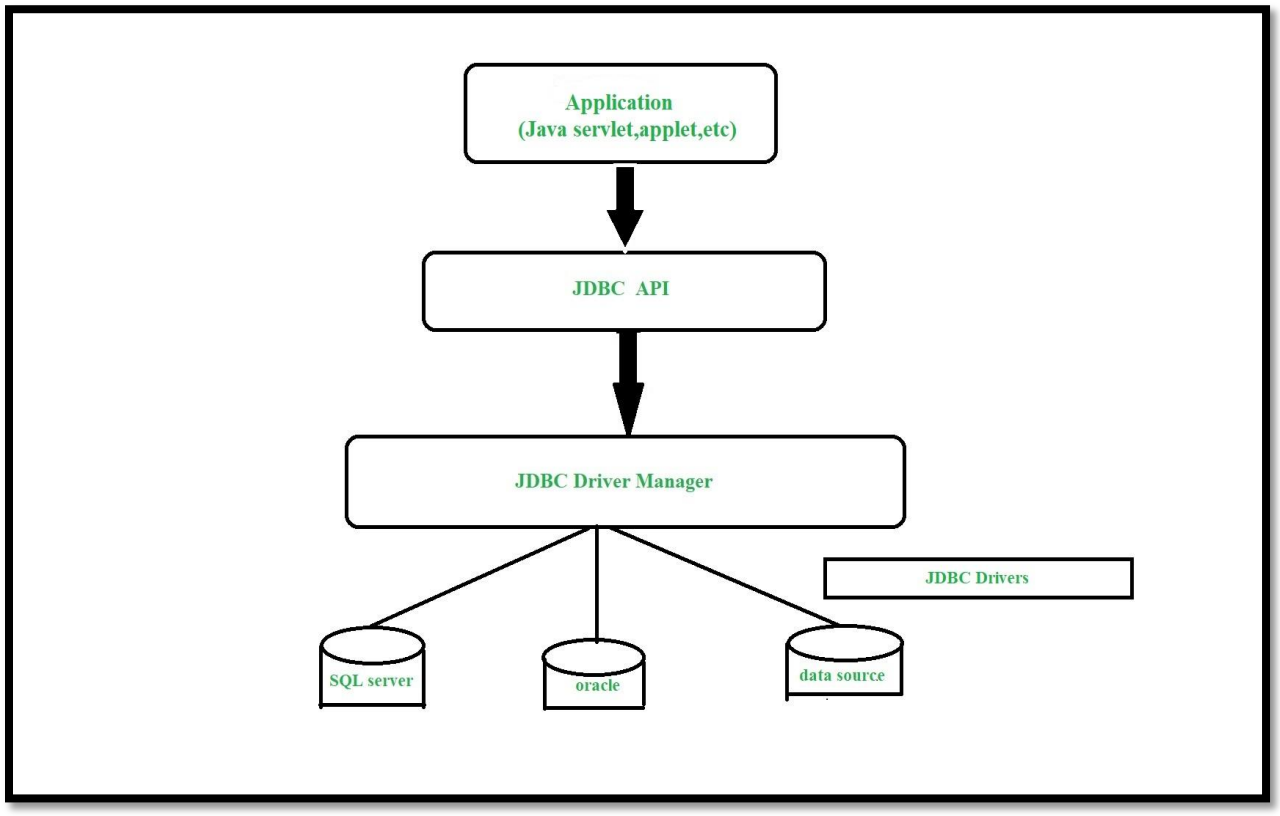
ODBC (Open Database Connectivity)

- A standard API developed by Microsoft to access database systems.
- Language-independent.
- Allows any application to communicate with any database, as long as an ODBC driver is available.
- Mostly used in C/C++ or other system-level languages.

JDBC (Java Database Connectivity)

- A Java-based API provided by Oracle to connect and execute queries with databases.
- Designed specifically for Java applications.

. JDBC Architecture



□ What is JDBC?

JDBC (Java Database Connectivity) is an API that enables Java applications to interact with databases using SQL. It provides a standard way to connect, execute queries, and retrieve results.

□ JDBC Architecture Components

JDBC architecture mainly consists of two layers:

1. JDBC API (Application Layer)

- Provides classes and interfaces (like `Connection`, `Statement`, `ResultSet`, etc.) to Java developers.
- Hides the underlying complexity of interacting with different databases.

2. JDBC Driver (Driver Layer)

- Converts JDBC API calls into database-specific calls.
- Different drivers are available for different databases (MySQL, Oracle, etc.).
-

Two-Tier Architecture

Definition:

A **Two-Tier architecture** is a client-server model where the application (client) communicates **directly** with the database without any intermediary.

Components:

1. **Client (Application Layer):**
   - Contains the user interface and the application logic.
   - Sends SQL queries directly to the database.
2. **Database (Data Layer):**
   - Processes the queries and returns the result to the client.

Workflow:

- The Java application uses JDBC to establish a connection with the database.
- SQL queries are sent directly to the database.
- The database responds with results, which are processed and displayed by the application.

Diagram:

```
[Client Application]  <------>  [Database Server]
        JDBC
```

Advantages:

- Simple and easy to implement.
- Good for small-scale applications.

Disadvantages:

- Not scalable.
- Security and performance issues if used in large systems.
- Changes in the database logic affect the client application directly.

□ Three-Tier Architecture

Definition:

A **Three-Tier architecture** introduces an intermediate layer (application server or middleware) between the client and the database, separating the presentation, logic, and data layers.

Components:

1. **Client (Presentation Layer):**
   - User interface (e.g., web browser or mobile app).
   - Sends requests to the application server.
2. **Application Server (Logic Layer):**
   - Contains business logic and processes client requests.
   - Interacts with the database using JDBC.
3. **Database Server (Data Layer):**
   - Stores and manages data.

- Responds only to the application server.

Workflow:

- The client sends a request to the application server.
- The server processes the request, performs JDBC operations, and gets data from the database.
- The result is returned to the client.

Diagram:

```
 [Client]  <----->  [Application Server]  <----->
[Database Server]
                  (JDBC used here)
```

Advantages:

- Better scalability and performance.
- Easier to manage and maintain (loose coupling).
- Enhanced security (clients don't access the DB directly).
- Supports distributed systems.

Disadvantages:

- More complex to set up.
- Higher development and maintenance cost.

Types of JDBC Drivers

JDBC (Java Database Connectivity) supports **four types of drivers**, which are used to connect Java applications to various databases. Each driver type has different characteristics in terms of performance, portability, and architecture.

□ Type 1: JDBC-ODBC Bridge Driver

➢ Description:

- Translates JDBC calls into **ODBC (Open Database Connectivity)** calls.

- Requires **ODBC driver** installed on the client machine.

➢ ***Architecture:***

```
Java Application → JDBC API → JDBC-ODBC Bridge → ODBC → Database
```

➢ ***Advantages:***

- Easy to use for testing and prototyping.
- Allows access to any database with an ODBC driver.

➢ ***Disadvantages:***

- **Platform-dependent** (requires ODBC setup).
- **Slower** due to multiple layers.
- Deprecated and removed from newer Java versions.

□ Type 2: Native-API Driver

➢ ***Description:***

- Converts JDBC calls into **native C/C++ API** calls specific to the database.
- Requires **native DB library** installed on the client.

➢ ***Architecture:***

```
Java Application → JDBC API → Native-API Driver → Native DB Library → Database
```

➢ ***Advantages:***

- **Better performance** than Type 1.
- Uses database-specific features efficiently.

➢ ***Disadvantages:***

- **Not portable** (platform-specific).
- Requires client-side native libraries.

□ Type 3: Network Protocol Driver (Middleware Driver)

➢ *Description:*

- Sends JDBC calls to a **middleware server** that translates them to DB-specific calls.
- Middleware handles DB connectivity, not the client.

➢ *Architecture:*

```
Java Application → JDBC API → Type 3 Driver →
Middleware → Database
```

➢ *Advantages:*

- **Fully portable**, as no native code is needed.
- Supports access to multiple databases via one driver.
- Suitable for enterprise applications.

➢ *Disadvantages:*

- Requires a **middleware server**.
- Slower than Type 4 in simple setups.

□ Type 4: Thin Driver (Pure Java Driver)

➢ *Description:*

- Directly converts JDBC calls into **database-specific protocol**.
- Written entirely in **Java** – no native code required.

➢ *Architecture:*

```
Java Application → JDBC API → Type 4 Driver → Database
```

➢ ***Advantages:***

- **Fastest and most efficient**.
- **Pure Java** – platform independent.
- Ideal for web and enterprise applications.

➢ ***Disadvantages:***

- Driver is **database-specific**, so you need different drivers for different databases.

 Comparison Table

Type	Description	Platform Dependent	Performance	Requires Native Code	Use Case
1	JDBC-ODBC Bridge	Yes	Low	Yes	Obsolete/testing
2	Native-API Driver	Yes	Medium	Yes	Client-side apps
3	Network Protocol Driver	No	Medium	No	Multi-tier apps
4	Thin Driver	No	High	No	Most Java apps

1. `Class` Class in JDBC

 Theory:

- Used to **load the JDBC driver class at runtime**.
- Common method:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

- This step is required to register the driver with `DriverManager`.

 Example:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

 2. `DriverManager` Class

 Theory:

- Part of `java.sql` package.
- Manages the **set of JDBC drivers**.
- Used to **establish a connection** with a database.
- Static method:

```
DriverManager.getConnection(url, user, password);
```

 Example:

```
Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/testdb", "root",
"password");
```

 3. `Connection` Interface

 Theory:

- Represents an **active connection** with a database.
- Provides methods to create `Statement`, `PreparedStatement`, etc.

 Common Methods:

- `createStatement()`
- `prepareStatement(String sql)`
- `close()`

 Example:

```
Connection con = DriverManager.getConnection(...);
```

4. `Statement` Interface

Theory:

- Used to execute **static SQL queries** (without parameters).
- Provides methods like `executeQuery()`, `executeUpdate()`

Example Program:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM
students");
while(rs.next()) {
    System.out.println(rs.getInt(1) + " " +
rs.getString(2));
}
```

5. `PreparedStatement` Interface

Theory:

- Used for **parameterized SQL queries** (dynamic values).
- Prevents **SQL injection** and improves performance for repeated queries.

Example Program:

```
PreparedStatement ps = con.prepareStatement("SELECT *
FROM students WHERE id = ?");
ps.setInt(1, 101);  // set parameter value
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    System.out.println("Name: " +
rs.getString("name"));
}
```

6. `ResultSet` Interface

 Theory:

- Stores the **result of a query**.
- Provides methods like:
  - `next()`
  - `getInt()`, `getString()`, etc.

 Example:

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

 Complete Mini JDBC Program Example

```
import java.sql.*;

public class JDBCExample {
    public static void main(String[] args) {
        try {
            // Load JDBC Driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Connect to DB
            Connection con =
DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/testdb",
"root", "password");

            // Use PreparedStatement
            PreparedStatement ps =
con.prepareStatement("SELECT * FROM students WHERE id =
?");
            ps.setInt(1, 101);
            ResultSet rs = ps.executeQuery();

            // Process ResultSet
            while (rs.next()) {
```

```
                System.out.println("ID: " +
rs.getInt("id"));
                System.out.println("Name: " +
rs.getString("name"));
            }

            // Close connection
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```