UNIT 2 JDBC Introduction

What is JDBC (Java Database Connectivity)?

1. JDBC is a Java API used to connect and interact with databases.
2. It provides a set of classes and interfaces for database operations.
3. JDBC enables Java applications to execute SQL queries and retrieve data.
4. It acts as a bridge between Java programs and database servers.
5. Supports various relational databases like MySQL, Oracle, and PostgreSQL.
6. JDBC uses SQL for performing database operations like CRUD (Create, Read, Update, Delete).
7. It includes drivers for communication with specific databases.
8. JDBC is part of Java Standard Edition (Java SE).

Why is JDBC used?

1. JDBC allows seamless database interaction for Java applications.
2. It helps in executing SQL queries and stored procedures.
3. JDBC provides a platform-independent way to interact with databases.
4. It supports transactions, ensuring data consistency.
5. Useful for performing operations like insert, update, delete, and select.
6. JDBC simplifies database operations through standard Java methods.
7. Supports multiple databases without changing application code.
8. Essential for Java applications that require database connectivity.

Advantages of JDBC POSTDOCS W

1. **Platform-independent**: JDBC is a part of Java, so it runs on any platform.
2. **Database-independence**: Works with various databases using different drivers.
3. **Transaction management**: Ensures ACID compliance with transactional support.
4. **Simplified SQL execution**: Allows easy execution of SQL commands.
5. **Connection pooling**: Improves performance by reusing database connections.
6. **Supports multiple operations**: Allows CRUD operations and batch processing.
7. **Open-source support**: Integrates well with open-source databases.
8. **Widely used**: JDBC is a mature and well-supported API in Java development.

Disadvantages of JDBC

1. **Complex error handling**: JDBC exceptions can be verbose and difficult to manage.
2. **Manual resource management**: Developers need to manually close connections, statements, and result sets.
3. **Limited to relational databases**: Does not support NoSQL databases natively.
4. **Performance overhead**: JDBC may cause performance issues when handling large datasets.
5. **High learning curve**: Beginners may find the API complex to understand and use.
6. **Driver dependency**: Requires proper drivers for each database type.
7. **Network overhead**: Communication between Java and databases may result in network latency.
8. **Security concerns**: SQL injection risks if queries are not properly parameterized.

** **Application of JDBC**

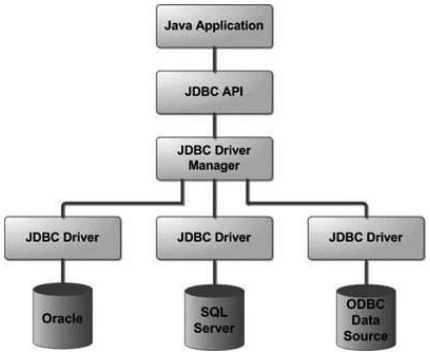
 Database Connection: Connect Java applications to various databases like MySQL, Oracle, and SQL Server.
 CRUD Operations: Perform Create, Read, Update, and Delete operations on database records.
 Transaction Management: Handle transactions with commit and rollback functionalities.

 Stored Procedures: Call and execute stored procedures within the database from Java applications.

** JDBC Architecture

The architecture of **JDBC (Java Database Connectivity)** can be understood in the following points:

1. **JDBC API**: This is the interface provided by Java for database access. It contains classes and methods to connect, send queries, and retrieve results from a database. The API enables the application to interact with various databases using standard SQL.
2. **JDBC Driver Manager**: It manages the communication between Java applications and database drivers. The DriverManager class loads the appropriate driver and establishes the connection to the database.
3. **JDBC Driver**: This is the database-specific implementation of the JDBC interface. It translates Java calls into database-specific operations. There are four types of JDBC drivers:
    - Type 1: JDBC-ODBC Bridge Driver
    - Type 2: Native-API Driver
    - Type 3: Network Protocol Driver
    - Type 4: Thin Driver (Pure Java Driver)
4. **Connection Interface**: This interface represents the connection with the database. It provides methods to create statements, manage transactions, and handle connection properties.
5. **Statement Interface**: This interface is used to execute SQL queries against the database. The Statement, PreparedStatement, and CallableStatement interfaces are its variations for executing queries, executing precompiled queries, and calling stored procedures, respectively.
6. **ResultSet Interface**: It is used to store and manipulate the results returned from executing SQL queries. It provides methods to iterate through the results, retrieve data, and update rows.
7. **JDBC URL**: The database connection URL specifies the database server, port, and other connection details. It helps JDBC identify the correct database to connect to.
8. **SQL Exception Handling**: JDBC provides the SQLException class to handle database-related errors, like connection failure or incorrect SQL syntax.



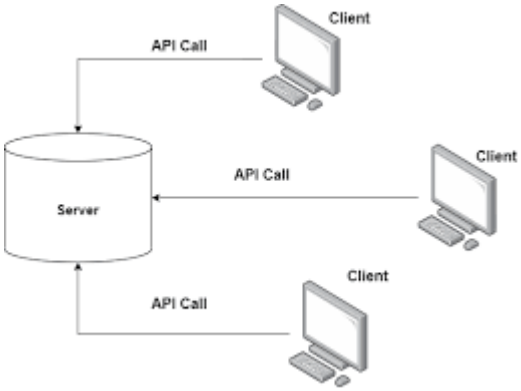
** Common JDBC components

1. **DriverManager**:
   - Manages a list of database drivers.
   - Establishes a connection between Java applications and the database.
   - Automatically selects the appropriate driver based on the database URL.
2. **Driver**:
   - An interface for database-specific drivers that communicate with the database.
   - Converts JDBC calls into database-specific commands.
   - There are four types of drivers (Type 1-4).
3. **Connection**:
   - Represents the connection to the database.
   - Provides methods to create Statement and manage transactions.
   - Manages connection attributes (auto-commit, transaction isolation).
4. **Statement**:
   - Executes SQL queries and returns results.
   - Three types:
     - Statement: For simple SQL queries.
     - PreparedStatement: Precompiled SQL queries for better performance.
     - CallableStatement: Used to execute stored procedures.
5. **ResultSet**:
   - Stores the result of SQL queries.
   - Allows iteration over results row by row.
   - Provides methods to retrieve data in various types (e.g., getInt(), getString()).
6. **SQLException**:
   - Handles database-related exceptions.
   - Provides details about the error, including error codes and messages

Two-Tier Database Design in the Context of JDBC

In a two-tier architecture, the interaction between the client and the database occurs directly. The application communicates with the database without any intermediary layers.

1. **Client-Server Architecture**: The two-tier architecture consists of two layers:
    - **Client (Application)**: This is the front-end, where the user interacts with the Java application.
    - **Server (Database)**: This is the back-end, where the database resides and processes queries.
2. **Direct Communication**: In JDBC, the Java application (client) communicates directly with the database server using SQL queries via the JDBC API.
3. **Thin Client, Heavy Server**: The application typically handles sending requests, while the database performs the heavy lifting (processing queries and returning data).
4. **Simple Architecture**: Two-tier architecture is straightforward, making it easy to implement and manage for small-scale applications.
5. **Real-time Data Access**: Since there's a direct connection, data is accessed and updated in real-time, without delay.
6. **Faster Development**: With direct database interaction, fewer components are involved, speeding up the development process.
7. **JDBC in Action**: The client uses JDBC to establish a connection with the database, execute SQL queries, and retrieve results.
8. **Example**: A Java desktop application (client) that uses JDBC to connect to a local or remote MySQL database (server), fetches records, and displays them to the user.
9. **Limited Scalability**: This design works well for small systems, but as the number of clients grows, performance and connection management can become bottlenecks.
10. **Use Case**: Suitable for standalone applications or small-scale client-server systems where users interact directly with a database, such as personal finance software or desktop inventory systems.

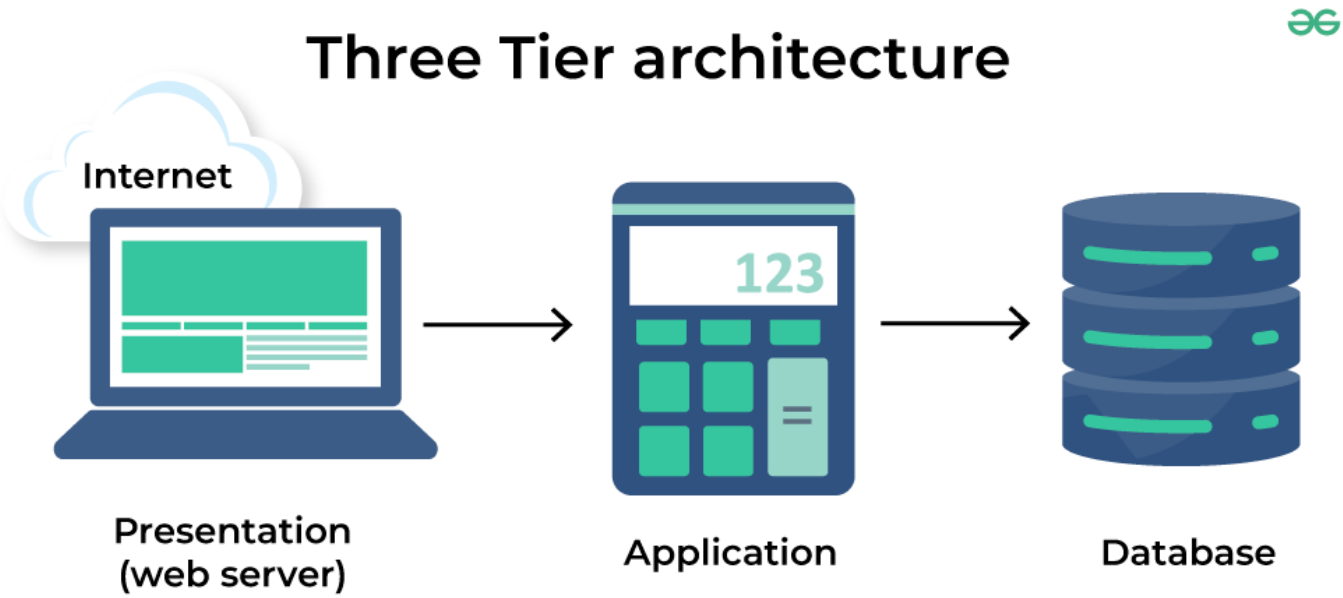


Three-Tier Database Design

The three-tier architecture introduces an additional layer between the client and the database, which helps in improving scalability, flexibility, and security.

1. **Three Layers**: The architecture is divided into three distinct layers:
    - **Presentation Layer (Client)**: The front-end where users interact with the application. This can be a web browser, desktop app, or mobile app.
    - **Application Layer (Middle-tier or Business Logic Layer)**: Contains the business logic of the application. It processes client requests, interacts with the database, and applies rules/operations before responding to the client.
    - **Database Layer (Data-tier)**: The back-end where the database resides, storing and managing data.
2. **JDBC in the Application Layer**: JDBC operates at the application layer, handling communication between the application and the database. It retrieves data from the database and sends it to the application logic for processing.
3. **Decoupling of Layers**: Each layer is independent, making the system more flexible. Changes in one layer (like updating the database) don't necessarily affect the other layers (like the client).
4. **Improved Security**: Since the client doesn't directly access the database, sensitive information like database credentials can be hidden, and security measures (like authentication and authorization) are managed in the middle tier.
5. **Scalability**: The application layer can handle multiple client requests simultaneously and distribute the load, allowing for better scalability in large systems.
6. **Business Logic Processing**: The middle tier contains the business logic (rules, algorithms, validation), so it offloads processing from the client and database. This makes the system more efficient and manageable.

7. **Ease of Maintenance**: Since the layers are separated, maintaining or updating the system becomes easier. Changes in one layer (such as upgrading the user interface) do not affect the others.
8. **Example**:
    - A web-based e-commerce application where:
        - The **client** is the web browser.
        - The **application layer** is the server-side logic (business rules, authentication) using a technology like Java EE.
        - The **database layer** holds customer information, product details, and orders in a relational database like MySQL or Oracle.
9. **Enterprise-Level Applications**: The three-tier design is commonly used in large, distributed systems like online banking, enterprise resource planning (ERP), and e-commerce platforms, where flexibility, security, and performance are critical.
10. **Better Load Balancing**: The middle tier can also implement load balancing strategies, ensuring that no single server is overwhelmed with requests, improving overall system performance.



Two-Tier Database Architecture	Three-Tier Database Architecture
It is a Client-Server Architecture.	It is a Web-based application.
In two-tier, the application logic is either buried inside the user interface on the client or within the database on the server (or both).	In three-tier, the application logic or process resides in the middle-tier, it is separated from the data and the user interface.
Two-tier architecture consists of two layers : Client Tier and Database (Data Tier).	Three-tier architecture consists of three layers : Client Layer, Business Layer and Data Layer.
It is easy to build and maintain.	It is complex to build and maintain.
Two-tier architecture runs slower.	Three-tier architecture runs faster.
It is less secured as client can communicate with database directly.	It is secured as client is not allowed to communicate with database directly.
It results in performance loss whenever the users increase rapidly.	It results in performance loss whenever the system is run on Internet but gives more performance than two-tier architecture.
Example – Contact Management System created using MS-Access or Railway Reservation System, etc.	Example – Designing registration form which contains text box, label, button or a large website on the Internet, etc.

JDBC Drivers

JDBC drivers are software components that enable Java applications to interact with a database. They handle the communication between Java applications and the database management system (DBMS). Here's an overview:

1. **Functionality**: JDBC drivers translate Java calls into database-specific calls, allowing Java applications to execute SQL statements and retrieve results from databases.
2. **Driver Types**: While there are different types of JDBC drivers, the main purpose of all JDBC drivers is to facilitate the connection between Java applications and databases.
3. **Driver Classes**: Each JDBC driver typically includes a driver class that implements the java.sql.Driver interface. This class is used to establish connections to the database.
4. **Connection Management**: JDBC drivers manage the details of establishing and maintaining a connection with the database, including handling connection parameters like URL, username, and password.
5. **SQL Execution**: They provide the necessary methods to execute SQL queries, update data, and retrieve results from the database.
6. **Result Handling**: JDBC drivers handle the processing of SQL query results, converting database result sets into Java objects for further manipulation.
7. **Error Handling**: They translate database errors into JDBC exceptions that the Java application can handle appropriately.
8. **Driver Registration**: Before using a JDBC driver, it must be registered with the DriverManager class. This can be done either by loading the driver class explicitly or by using service provider mechanism (e.g., via META-INF/services).
9. **Implementation Details**: JDBC drivers are implemented as libraries or JAR files that need to be included in the classpath of the Java application.
10. **Vendor-Specific**: Different databases have their own specific JDBC drivers, which are tailored to communicate effectively with their respective database management systems.

**** JDBC Driver types (JDBC-ODBC, Native API driver, Network Protocol, Thin Driver)**

Type 1: JDBC-ODBC Bridge Driver

1. Acts as a bridge between JDBC calls and ODBC (Open Database Connectivity).
2. Converts JDBC method calls into ODBC function calls.
3. Requires the ODBC driver to be installed on the client machine.
4. Suitable for accessing databases that provide ODBC drivers.
5. Not fully written in Java, hence not portable.
6. Slower performance due to the additional conversion layer (JDBC to ODBC).
7. Deprecated and not recommended for modern applications.
8. Example URL: jdbc:odbc:DataSourceName.

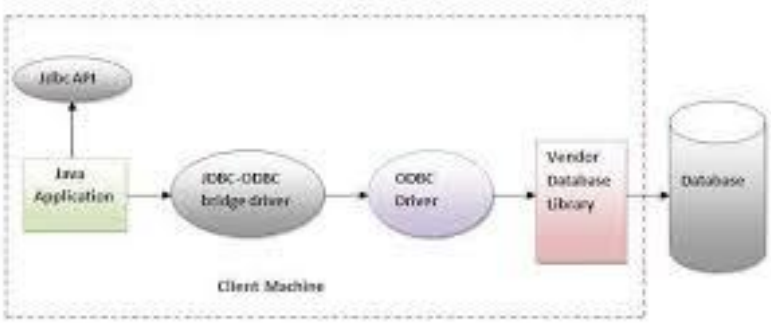


Figure- JDBC-ODBC Bridge Driver

---

2. Type 2: Native API Driver

1. Uses native (platform-specific) libraries to convert JDBC calls into database-specific calls.
2. Translates JDBC calls to a client-side library provided by the database vendor.
3. Requires the installation of native database libraries on the client machine.
4. Performance is better than Type 1 since it avoids the ODBC layer.
5. Limited portability due to platform dependence.
6. Suitable for applications where the client and server run on the same platform.
7. Vendor-specific, so each database needs its own native library.
8. Example URL: jdbc:databaseVendor://localhost:3306/database.

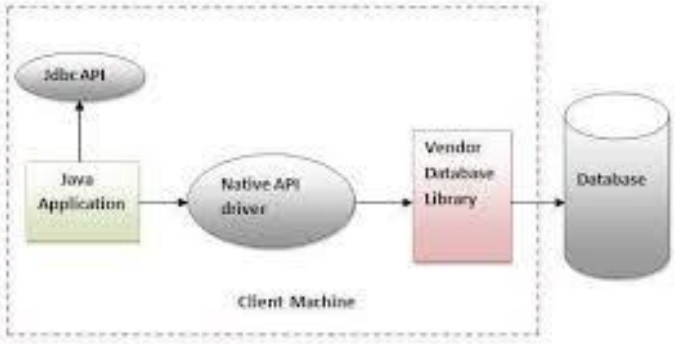


Figure- Native API Driver

3. Type 3: Network Protocol Driver

1. Uses a middleware server to convert JDBC calls to database-specific protocols.
2. The client-side driver communicates with a server-side component that talks to the database.
3. No native libraries required on the client machine.
4. Offers better portability compared to Type 2 drivers.
5. Allows communication with multiple databases through the same middleware.
6. Suitable for large-scale, distributed systems.
7. Performance depends on the efficiency of the middleware.
8. Example URL: jdbc:network://middlewareServer:port/database.

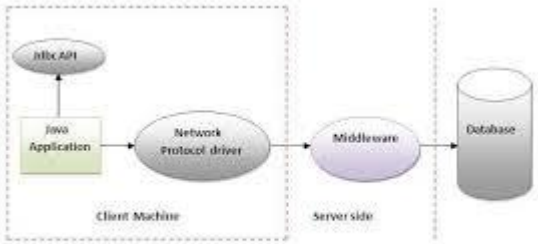


Figure- Network Protocol Driver

4. Type 4: Thin Driver (Pure Java Driver)

1. Fully written in Java, providing database connectivity directly from the client to the database.
2. Converts JDBC calls directly into database-specific protocols.
3. No native libraries or middleware are required.
4. Highly portable across different platforms due to its pure Java implementation.
5. Best performance among all driver types since no additional layer is involved.
6. Most widely used driver for modern applications.
7. Ideal for web-based applications where the client platform may vary.
8. Example URL: jdbc:mysql://localhost:3306/database.

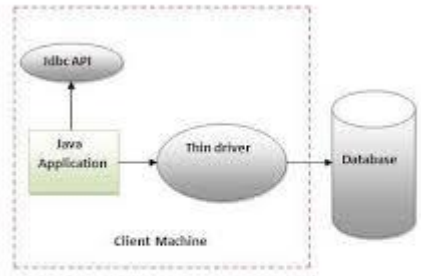


Figure- Thin Driver

**** Creating JDBC Application**

1. **Import the package**

•Requires that you include the packages containing the JDBC classes needed for database programming.

•Most often, using*import java.sql.**will suffice.

2. Register the JDBC driver:

•Requires that you initialize a driver
•so you can open a communication channel with the database.

3. Open a connection:

•Requires using the*DriverManager.getConnection()*method
•to create a Connection object,

•which represents a physical connection with the database.

4. Execute a query:
•Requires using an object of type Statement for building and

•submitting an SQL statement to the database.

5. Extract data from result set:
•Requires that you use the appropriate*ResultSet.getXXX()*method

•to retrieve the data from the result set.

6. Clean up the environment:
•Requires explicitly closing all database resources versus

•relying on the JVM's garbage collection.

<u> Steps for connecting java program and database</u>**

1. Load and Register the JDBC Driver

1. **Identify the driver** needed for the specific database (e.g., MySQL, Oracle).
2. **Use Class.forName()** to load the driver class (e.g., Class.forName("com.mysql.cj.jdbc.Driver")).
3. **Register the driver** with the DriverManager.
4. **For JDBC 4.0**, explicit loading may not be needed as it automatically loads the driver.
5. The **driver must be in the classpath** of the application.
6. If the driver is not found, a **ClassNotFoundException** is thrown.

2. Establish the Connection

1. **Define the JDBC URL** (e.g., jdbc:mysql://localhost:3306/databaseName).
2. **Pass the URL, username, and password** to DriverManager.getConnection().
3. **Get a Connection object** from DriverManager.
4. Ensure **proper credentials** (username and password) are provided.
5. If the connection is successful, a **Connection object is returned**.
6. If there is an issue (e.g., invalid credentials), a **SQLException is thrown**.

3. Create a Statement Object

1. **Obtain a Statement** from the Connection object (e.g., Statement stmt = con.createStatement()).
2. Choose the appropriate **statement type**: Statement, PreparedStatement, or CallableStatement.
3. **Statement** is used for simple queries, PreparedStatement for precompiled SQL.
4. **CallableStatement** is used for executing stored procedures.
5. Using **PreparedStatement** helps avoid SQL injection and improves performance.
6. Each type of statement allows sending **SQL queries** to the database.

4. Execute SQL Queries

1. For **retrieving data**, use executeQuery() (e.g., ResultSet rs = stmt.executeQuery("SELECT * FROM tableName")).
2. For **updating data**, use executeUpdate() (e.g., int result = stmt.executeUpdate("UPDATE tableName SET column = value")).
3. The execute() method is used when the SQL statement could be either a **query or update**.
4. **Check the return type** (e.g., ResultSet for queries, int for updates).
5. Handle exceptions that may arise due to **SQL syntax errors or logic errors**.
6. If successful, the result is stored in a **ResultSet (for queries)** or returns **affected rows (for updates)**.

5. Process the ResultSet (for Queries)

1. Use the **ResultSet object** to retrieve data from a SELECT query.
2. **Iterate through the ResultSet** using rs.next() to move to the next row.
3. Extract data from the columns using getString(), getInt(), etc.
4. Handle multiple columns by **specifying the column name or index**.
5. Process each row as needed (e.g., displaying it or storing it).
6. Close the ResultSet when finished using **rs.close()** to free up resources.

6. Close the Connection and Resources

1. **Close the Statement** object after the query is complete using stmt.close().
2. **Close the Connection** using con.close() to release database resources.
3. Ensure all resources (e.g., Connection, Statement, ResultSet) are properly closed.
4. Use a **try-with-resources block** in Java 7+ to automatically close resources.
5. Handle potential **SQLExceptions** during the closing process.
6. Closing the connection ensures **efficient resource management** and avoids memory leaks.

** JDBC Connection with oracle and Mysql