**JDBC (Java Database connectivity)**

* JDBC stands for java database connectivity. It is the technology , the only one standard API in java which is used to connect java application with any database application.

**What is data?**

It is a collection of values or raw facts, which convey information any specific entity, its properties, facts, quantity, quality and other information.

**What is Data Storage?**

It is the process of storing the data, considering how and where to store it digitally.

It can be stored in :-

* ***File System***
* ***DataBase***
  + ***RDBMS***
  + ***No SQL DataBases***

**Q. What is the difference between File System and DataBase?**

|  |  |
| --- | --- |
| **File System** | **Database** |
| A way to store and organize files on storage devices like HDD or SSD. | An organized collection of data managed by a Database Management System (DBMS) |
| Stores data in plain files (e.g., .txt, .csv). | Stores data in structured tables (rows and columns). |
| No Efficient Query Processing (Manual Access) | Efficient Query Processing (Using SQL) |
| Provides Less Security (Basics) | Provides More Security (Roles, authentication, encryption) |
| **File System** | **DataBase** |
| Redundant data can be present. Less Consistent | Redundant data can be avoided. More data consistency |
| Concurrency is less flexible. Difficult to manage – no built-in control. | Built-in concurrency control (multiple users can access safely) |
| Backup & Recovery is manual and time consuming | Backup & Recovery is Automated, with rollback, recovery, and backups. |
| Hard to manage large data efficiently, not suitable for complex transactions | Efficient for managing large data and complex queries |
| Less Complex to use | More Complex to use |
| Windows File Explorer, Linux File System | |  | | --- | |  |  |  | | --- | | MySQL, Oracle, SQL Server, PostgreSQL | |

**Q. What is Database?**

* A database is a structured and organized collection of data or information, typically stored virtually in the computer system.

**The need of connecting java application with the database application.**

Within the java application it is not possible to store the data permanently, so that we will lose the processed data once the application is closed, in order to overcome this problem, the application data has to be stored somewhere outside the application permanently so that the data can be accessed in the application whenever required in future.

**API : API stands for Application Programming Interface**.

It is a software(contains set of rules and protocol) which will help two different software communicate with each other which are incompatible to each other.

**Real Time Examples:**

1.Vyantkat knows : Tamil

Martin knows: Spanish

* So, the third person(some name ) which will know **both Tamil and Spanish** will help **Vyankat** and **Martin** to communicate with each other!

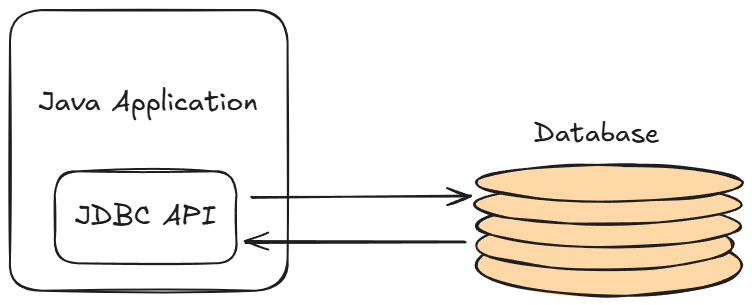
2. API is like a **waiter in a restaurant**

* Customer(**End-User/Client**)tell the waiter(**API**) what they want.
* The waiter(**API**) transfers the order(**request**) to the Kitchen(**Back-end logic/Database**)
* The kitchen takes the order(**request**),prepares the food(**response**) and the waiter(API) brings it back to the Customer(**End-User/Client**)

* **JDBC is an API** which will help java application to communicate with database application which are incompatible to each other.

**Incompatibility between java application and database application**

Since the *java application understands java commands* or instructions and *the database application understands structured queries****(SQL)*** which makes both the application incompatible to each other. To overcome this incompatibility and to establish the connection between java application and database application we can make use of JDBC technology that's why JDBC is also called as API.

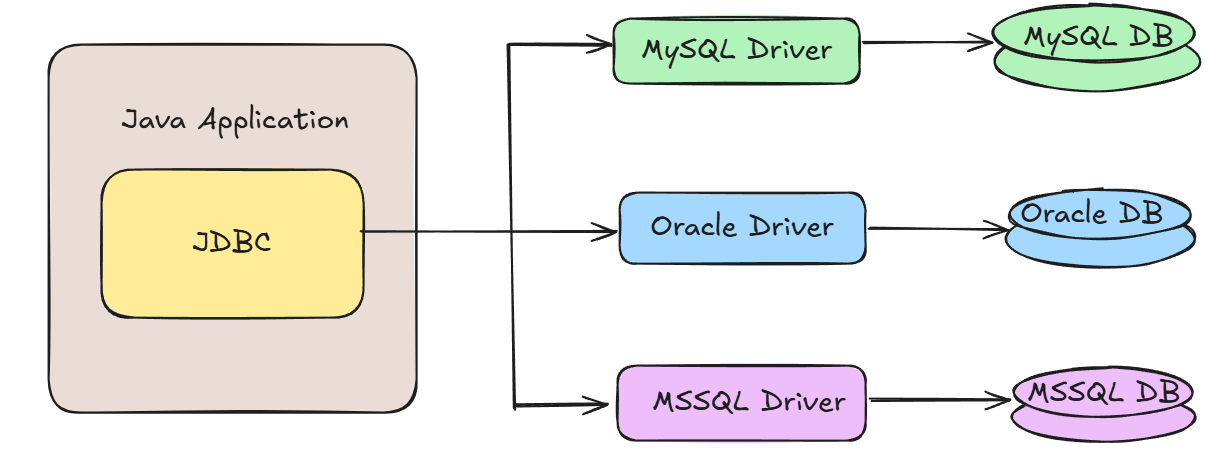


**JDBC API**

* Data stored in a relational database can be accessed with the help of **JDBC API** and respective **Database Driver**
* *The* ***API*** *provides a mechanism for dynamically loading the correct java packages and registering them with the* ***JDBC Driver Manager***

**JDBC with Database Driver**

* Driver is a software which will help JDBC API to communicate with database application. Since JDBC API is a part of java application it will take the help of database driver to communicate with database application, each and every database application comes up with dedicated driver software. This software will be developed by the database application developers.
* **Every Database has its own Driver Software!** using which a JDBC API can communicate with database. So, **Drivers are dependent on the Database Application!**
* **JDBC API is independent of any Driver Software/Database**, but to communicate with database, a particular Driver must be integrated in the java application!.
* JDBC API has a built in mechanism which loads the respective Driver classes into the memory if integrated.



**Steps to connect the Java application with the database application**

* ***Step 1: Load or register the driver* (Not required from JDBC 4+/Java 6, as it will automatically load and register the driver!)**
* ***Step 2: Open/Create the connection between java application and the database application***
* ***Step 3: Create or prepare the statement***
* ***Step 4: Execute the statement/Query***
* ***Step 5: Close the connection between java application and database application***
* ***Step 6: Deregister the Driver if registered***
* *In java library we have a package called as* ***java.sql*** *package which represents JDBC API.*

**Postgre SQL Installation**

**Step 1:** Visit postgresql.org ,click on download installer and download compatible PostgreSQL

**Step 2:** Run as administrator

-->Select all except stack builder

--> Select Intallation directory

-->Set Password

-->press next and finish the installation

--> Open pgAdmin and run as administrator

**Creating a simple Maven Project**

***Step 1: Create a simple maven project, provide following***

* Group ID: ***package name***(com/edu/org.package\_name)
* Artifact ID: ***project name***

**Step 2: *Add Driver dependency in pom.xml***

* Create a parent tag:

**<dependencies>**

**</ dependencies>**

* In the standard web browser search for [mvnrepository.com](https://mvnrepository.com/) and click on the link of official website of the maven repository
* In the search bar search for **postgresql** and click on search or press enter
* Select the first search result as **PostgreSQL JDBC Driver**
* Choose the appropriate version with high usage and no vulnerabilities ([**42.7.3**](https://mvnrepository.com/artifact/org.postgresql/postgresql/42.7.3))
* Copy the maven code and paste it inside <dependencies>

</ dependencies>

**ResultSet**

* ResultSet Interface is present in **java.sql** package.It is a table of data representing a database result set, which is usually generated by executing a statement that queries the database.
* It’s object holds the data, which is returned from the database when select query is executed.
* We can get ResultSet Object by calling some built in methods like **executeQuery(), getResultSet()**, etc
* The ResultSet Object maintains the cursor that points to row of the result set fetched from the select query
* The **next()** method is used to move the cursor to the next position in a forward direction
* boolean next() throws SQLException;

🡪*Moves the cursor forward one row from its current position. A ResultSet cursor is initially positioned* ***before the first row****; the first call to the method next makes the first row the current row; the second call makes the second row the current row, and so on.*

*When a call to the next method returns false, the cursor is positioned after the last row*

* We can get the data from the column by calling **built in overloaded getter methods** for e.g getInt(int columnIndex), getString(String columnLabel),etc. We can pass column name or column position to it!
* int getInt(String columnLabel) throws SQLException;
* int getInt(int columnIndex) throws SQLException;, etc

**Connection**

* Connection is an interface present in java in **java.sql** package
* We can get the **connection object** by calling **getConnection()** method
* **getConnection()** *is a public static overloaded method present in DriverManager class*

public static Connection

getConnection(formal arguments..) throws SQLException

* ***Connection object is created only when the URL, user and password are accurate!***

**Statement**

* Statement is an interface present in **java.sql** package
* We can get the statement object by calling createStatement() method

**public Statement createStatement() throws SQLException**

* Statement objects are generally used to execute **the static queries**!
* **Static queries** is an SQL command which will not change during runtime i.e **it will not include user input**

**e.g. Select queries, Creation queries with hardcoded values**

* ***Insert, Update , delete is not recommended using statement objects!***

**PreparedStatement**

* It is an interface present in **java.sql** package. It is the child of (sub-interface)of Statement.
* We use Prepared Statement Object in order to execute **dynamic queries.**
* **Dynamic query:**  Query consisting **input from User or Inputs constructed at Runtime**
* To execute a dynamic query, we have to write it with **Placeholders(?)**

e.g. String query=”INSERT INTO products VALUES**(?,?,?,?,?)**”**;**

* We can create a prepared statement object by calling **prepareStatement()** method

**public PreparedStatement prepareStatement(args) throws SQLException;**

* We can set the data from the column by calling **built in setter methods** for e.g setInt(), setString(), etc.

**e.g.: "INSERT INTO products VALUES(?,?,?,?)";**

**\*\*\*Note: In JDBC, all the statements are auto-committed!!!\*\*\***

|  |  |
| --- | --- |
| Statement | PreparedStatement |
| Used for executing a static SQL | Used for executing a dynamic SQL |
| Cannot accept parameters at Runtime | Can accept parameters at Runtime |
| Slower as compared to PreparedStatement | Faster because it is used for executing precompiled SQL statement |
| Suitable for executing DDL commands  (CREATE, ALTER, DROP, TRUNCATE) | Suitable for executing DML commands  (INSERT, UPDATE, DELETE) |
| Cant be used for storing /retrieving image and file in database | Can be used for storing /retrieving image and file in database (CLOB/BLOB) |
| Same SQL query can’t be executed repeatedly for different values | Same SQL query can be executed repeatedly for different values |
| Less Secure(Prone to SQL injection) | More Secure |

**Methods used to execute statements**

1. **boolean execute(String sql) throws SQLException**

* It is used to execute both select as well as non-select queries **i.e.** any type of statements **(DDL, DML, DCL, TCL, DQL) .**
* This method returns a **boolean** value.
* Returns **true** if the result is a ResultSet object (**i.e. if it contains tabular data)**; **false** if it is an update count or there are no results **i.e** It returns **true** if the statement to be executed is **DQL** otherwise it returns false.
* We cannot store the result of execute() directly in the ResultSet object!. We have to make use of **getResultSet(),** which returns the current result as a ResultSet object or null if the result is an update count or there are no more results.

1. **ResultSet executeQuery(String sql) throws SQLException;**

* This method is used to execute only select queries **i.e only** **DQL** statements.
* This method returns the object of type ResultSet which will contain result returned form the query.
* This method cannot be called on a PreparedStatement or CallableStatement.

1. **int executeUpdate(String sql) throws SQLException**

* This method is used to execute only non-select queries **i.e. only** **DML** statements**(INSERT, UPDATE, DELETE)**.
* This method returns int type value i.e. **number of rows affected** after execution of DML statement.
* This method cannot be called on a PreparedStatement or CallableStatement.

**Batch Execution/Processing**

* **Batch Processing/Execution** in JDBC is a Process/ mechanism that allows multiple SQL statements (usually of the same type, like **INSERT, UPDATE, or DELETE**) to be grouped together and executed once as a batch, instead of executing them one by one.
* It allows us to group ***related SQL statements*** into a batch and submit them with one **executeBatch()** call statement, which returns **int[] where each index contains updated count.**
* It is particularly useful for improving **performance** and **efficiency**, minimizing the overall execution time.
* Ideal approach when we want to **manipulate bulk data.**
* We can use **executeBatch()**  to perform **DML operations LIKE**

INSERT, UPDATE, DELETE

* **DDL (not recommended) and DQL, TCL and DCL** are not allowed!

A diagram of a driver

AI-generated content may be incorrect.

**With Batch**

A diagram of a driver and driver

AI-generated content may be incorrect.

**Closing the Connection**

* If we don’t close the connection, it may lead to memory leakage.
* Connections are **costly in terms of execution**, so to manage the load overhead and to maintain efficiency, closing the connection is recommended and a good practice!**[Calling a friend and not cutting it example]**
* It is always recommended and a safe practice to close the connection in **finally block** *to ensure the connection release.*
* **Closing the connection using try-resource**
* After Java 8, it is recommended to use **try - resource block** , where

**Compiler will** **close all the connections when we exit try block!**

**Stored procedures**

* Stored procedures are the set of SQL statements (grouped together) to be executed to perform specific task on the database.
* These are beneficial when we are dealing with ***multiple tables with complex scenarios.***
* It is similar to **a method with void return type of java.**
* Stored Procedures:

 **Does not return a value** (directly).

 Is called with CALL (not SELECT).

 Can have **IN, OUT or both** parameters.

 Can perform **transaction control** (COMMIT, ROLLBACK)

**Syntax to create stored procedure**

**Step 1:**

* Go to Procedures segment under Schema
* Provide Procedure name and Parameter information
* Write Code for that procedure

e.g.

**BEGIN**

**INSERT INTO** teacher

**VALUES**(tech\_id, tech\_name, tech\_gender, tech\_age, tech\_email, tech\_phone, tech\_blood\_group);

**INSERT INTO** id\_card

**VALUES**( tech\_id, tech\_name, tech\_phone, tech\_blood\_group );

**END;**

**To call in the Procedure**

**Step 2:**

**call sp\_create\_teacher\_records(1,'salunke','female',50,'sal@mail.com','9876898976','A+')**

**Functions**

* A **function** is a named block of SQL/PLpgSQL code used to execute

SQL queries .

**Function:**

* **Returns a value**
* Can be called from a **SELECT, WHERE, JOIN, or another function**.
* Can be **used inside SQL expressions**.

**Syntax to create functions**

**Step 1:**

* Go to Functions segment under Schema
* Provide Function name and Parameter information
* Write Code for that function

e.g.:

**DECLARE** tech\_count **integer**;

**BEGIN**

**SELECT** **COUNT(\*) INTO** tech\_count

**FROM** teacher

**WHERE** gender=tech\_gender;

**RETURN** tech\_count;

**END**;

**To execute the function:**

**Step 2: select count\_by\_gender('female');**

| **Feature** | **Function** | **Procedure** |
| --- | --- | --- |
| **Returns a value** | ✅ Yes (using RETURN) | ❌ No |
| **Called with** | SELECT or inside SQL expressions | CALL |
| **Can be used in SQL queries** | ✅ Yes | ❌ No |
| **Transaction control (COMMIT, ROLLBACK)** | ❌ Not allowed | ✅ Allowed |

**Statements** Only **DQL and DML All**

**CallableStatement**

* It is an interface, which can be used to call or execute **the stored procedures and functions**
* prepareCall() method is used ***to get CallableStatement object***

[**CallableStatement**](eclipse-javadoc:%E2%98%82=jdbc.shop/C:%5C/Program%20Files%5C/Java%5C/jdk-22%5C/lib%5C/jrt-fs.jar%60java.sql=/javadoc_location=/https:%5C/%5C/docs.oracle.com%5C/en%5C/java%5C/javase%5C/22%5C/docs%5C/api%5C/=/=/module=/true=/=/maven.pomderived=/true=/%3Cjava.sql(Connection.class%E2%98%83Connection~prepareCall~Ljava.lang.String;%E2%98%82java.sql.CallableStatement)[**java**](eclipse-javadoc:%E2%98%82=jdbc.shop/C:%5C/Program%20Files%5C/Java%5C/jdk-22%5C/lib%5C/jrt-fs.jar%60java.sql=/javadoc_location=/https:%5C/%5C/docs.oracle.com%5C/en%5C/java%5C/javase%5C/22%5C/docs%5C/api%5C/=/=/module=/true=/=/maven.pomderived=/true=/%3Cjava)**.**[**sql**](eclipse-javadoc:%E2%98%82=jdbc.shop/C:%5C/Program%20Files%5C/Java%5C/jdk-22%5C/lib%5C/jrt-fs.jar%60java.sql=/javadoc_location=/https:%5C/%5C/docs.oracle.com%5C/en%5C/java%5C/javase%5C/22%5C/docs%5C/api%5C/=/=/module=/true=/=/maven.pomderived=/true=/%3Cjava.sql)**.**[**Connection**](eclipse-javadoc:%E2%98%82=jdbc.shop/C:%5C/Program%20Files%5C/Java%5C/jdk-22%5C/lib%5C/jrt-fs.jar%60java.sql=/javadoc_location=/https:%5C/%5C/docs.oracle.com%5C/en%5C/java%5C/javase%5C/22%5C/docs%5C/api%5C/=/=/module=/true=/=/maven.pomderived=/true=/%3Cjava.sql(Connection.class%E2%98%83Connection)**.prepareCall (**[**String**](eclipse-javadoc:%E2%98%82=jdbc.shop/C:%5C/Program%20Files%5C/Java%5C/jdk-22%5C/lib%5C/jrt-fs.jar%60java.sql=/javadoc_location=/https:%5C/%5C/docs.oracle.com%5C/en%5C/java%5C/javase%5C/22%5C/docs%5C/api%5C/=/=/module=/true=/=/maven.pomderived=/true=/%3Cjava.sql(Connection.class%E2%98%83Connection~prepareCall~Ljava.lang.String;%E2%98%82java.lang.String) **sql) throws** [SQLException](eclipse-javadoc:%E2%98%82=jdbc.shop/C:%5C/Program%20Files%5C/Java%5C/jdk-22%5C/lib%5C/jrt-fs.jar%60java.sql=/javadoc_location=/https:%5C/%5C/docs.oracle.com%5C/en%5C/java%5C/javase%5C/22%5C/docs%5C/api%5C/=/=/module=/true=/=/maven.pomderived=/true=/%3Cjava.sql(Connection.class%E2%98%83Connection~prepareCall~Ljava.lang.String;%E2%98%82java.sql.SQLException)

* Creates a CallableStatement object for calling database stored procedures and functions.

**Database Metadata**

* **Metadata** summaries basic information about data, making finding & working with particular instances of data easier
* **DatabaseMetaData** is an interface present in **java.sql ,**that provides a variety of methods to obtain comprehensive information about the database.
* There are different methods available to get the metadata
* **Table**: getTables(), getColumns(), getPrimaryKeys()
* **User and schema**: getUserName(), getSchemas()
* **Database-Level:** getDatabaseProductName(), getDatabaseProductVersion(), getDriverName(), getDriverVersion(), supports FullOuterJoins(), supports Stored Procedures(), supports Transactions(), supports BatchUpdates() tadata

**ResultSet Metadata**

* **ResultSetMetaData** interface provides methods to get metadata from the ResultSet object.
* There are different methods available to get the meta data
* getColumnCount(), getColumnLabel(), getColumnName(), getColumnTypeName(), getTableName(), getSchemaName()

**Connection Pool**

[In general, pool is the collection or group of entities]

* **E.g. One company(c1) has pool of employees(resources), other companies will ask the resources from c1 for some project, after job is done, c2 will release the resources back to c1 and pool will get refilled!**

A diagram of a company

AI-generated content may be incorrect.

* If we do **DriverManager.getConnection(URL,user,password):-**
* Insert
* Update
* Delete ……
* *For each and every task/ operation, Driver Manager have to verify all the details and then connection object is created!, which will consume more time!*
* So to optimize this, We can make us of **Connection pool**! ,which will contain ***collection of connection objects.***
* **Connection Pool** is the collection of connection objects
* Connection object are utilized on request, and returned back to **connection pool** after the usage.

**A diagram of a pool

AI-generated content may be incorrect.**

**Buiding a connection pool and utilizing it**

**Step 1: Create a ConnectionPool class**

* Create one **List<Connection> connectionPool**

To store connection objects.

**Step 2: Create methods to**

**returnConnectionObject()** and **receiveConnectionObject()** as static methods.

**Step 3: Create one Driverclass**

* Call **ConnectionPool.returnConnectionObject()** methodto ***receive connection object from the ConnectionPool***
* Perform operations using this **connection object**
* Call **ConnectionPool.receiveConnectionObject()** method

to ***return connection object back to the Connection Pool.***

**Transaction Management**

* Multiple parts/units of actions grouped together are called as ***transactions.***

e.g.--> **select route, search flights, selecting one, providing more information, payment:-**

* **If successful, booking confirmed!(Transaction is complete)**
* **If failed, (Transaction is failed)**
* **So to handle this, we can manage it like:**
* **If all operations succeed, then it will be stored into the table as a single transaction! (COMMIT)**
* **If one of the operation fails, we will revert all the related information (ROLLBACK)**

A diagram of action and action

AI-generated content may be incorrect.

* **It is used to prevent unnecessary data/junk data to be stored in the table**