# Introduction to JDBC API

## Overview of JDBC

The Java Database Connectivity (JDBC) API is a standard Java interface that allows Java applications to interact with relational databases in a platform-independent manner. It provides a unified set of classes and methods for connecting to a database, executing SQL queries, and retrieving results. JDBC is a vital component of Java’s ecosystem and enables developers to build robust, data-driven applications.  
  
JDBC acts as a bridge between a Java application and a database management system (DBMS). Through the use of JDBC drivers, applications can interact with different databases such as MySQL, PostgreSQL, SQLite, Oracle, and SQL Server without needing database-specific code.

## Architecture of JDBC

The JDBC architecture consists of two primary layers:  
  
1. Application Layer:  
 - This layer consists of the Java application that uses JDBC API to send SQL queries and retrieve results from the database.  
 - The application interacts with the JDBC API, which is part of the java.sql package.  
  
2. Driver Layer:  
 - This layer includes database-specific JDBC drivers. Each driver implements the JDBC API to translate application calls into database-specific calls.  
 - JDBC drivers communicate directly with the database to execute SQL commands and fetch results.

### JDBC Workflow

The typical workflow for using JDBC in a Java application is as follows:  
  
1. Load the JDBC Driver:  
 - Ensure the appropriate JDBC driver for the target database is available.  
 - The driver is automatically loaded by calling DriverManager.getConnection.  
  
2. Establish a Connection:  
 - Use the DriverManager class to establish a connection to the database.  
  
3. Create and Execute Statements:  
 - Use the Connection object to create Statement or PreparedStatement objects to execute SQL queries.  
  
4. Process Results:  
 - Retrieve query results using the ResultSet object.  
  
5. Close Resources:  
 - Always close the ResultSet, Statement, and Connection objects to release database resources.

## Key Components of the JDBC API

### DriverManager

Manages the list of database drivers and establishes connections to the database.

### Connection

Represents an active connection to the database. It is used to create Statement and PreparedStatement objects and to manage transactions.

### Statement

Used to execute static SQL queries. Suitable for queries that do not require input parameters.

### PreparedStatement

A subclass of Statement used for executing parameterized SQL queries. Prevents SQL injection attacks and improves performance.

### ResultSet

Represents the result of a SELECT query. Provides methods to iterate through rows and retrieve column values.

### SQLException

Represents errors that occur while interacting with the database.

## SQL Injection and the Role of PreparedStatement

SQL injection is a security vulnerability that occurs when untrusted user input is included in an SQL query without proper sanitization. This can allow attackers to manipulate SQL queries, potentially exposing sensitive data or damaging the database.

Example of Vulnerable Code:

String sql = "SELECT \* FROM users WHERE username = '" + username + "' AND password = '" + password + "'";  
Statement stmt = conn.createStatement();  
ResultSet rs = stmt.executeQuery(sql);

Secure Code with PreparedStatement:

String sql = "SELECT \* FROM users WHERE username = ? AND password = ?";  
PreparedStatement pstmt = conn.prepareStatement(sql);  
pstmt.setString(1, username);  
pstmt.setString(2, password);  
ResultSet rs = pstmt.executeQuery();

## Practical Example: Connecting to SQLite

import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
import java.sql.Statement;  
  
public class JdbcExample {  
 public static void main(String[] args) {  
 String dbUrl = "jdbc:sqlite:sample.db";  
  
 try (Connection conn = DriverManager.getConnection(dbUrl)) {  
 // Create a table  
 String createTable = "CREATE TABLE IF NOT EXISTS employees (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)";  
 try (Statement stmt = conn.createStatement()) {  
 stmt.execute(createTable);  
 }  
  
 // Insert data  
 String insertSQL = "INSERT INTO employees (name, age) VALUES (?, ?)";  
 try (PreparedStatement pstmt = conn.prepareStatement(insertSQL)) {  
 pstmt.setString(1, "Alice");  
 pstmt.setInt(2, 28);  
 pstmt.executeUpdate();  
  
 pstmt.setString(1, "Bob");  
 pstmt.setInt(2, 35);  
 pstmt.executeUpdate();  
 }  
  
 // Query data  
 String query = "SELECT \* FROM employees";  
 try (Statement stmt = conn.createStatement();  
 ResultSet rs = stmt.executeQuery(query)) {  
  
 while (rs.next()) {  
 System.out.println("ID: " + rs.getInt("id") + ", Name: " + rs.getString("name") + ", Age: " + rs.getInt("age"));  
 }  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

## Summary

JDBC is a powerful and flexible API for interacting with relational databases in Java. By understanding its components, such as Connection, Statement, PreparedStatement, and ResultSet, developers can build secure, efficient, and scalable database-driven applications. With its standardized interface, JDBC enables seamless interaction with a variety of database systems, making it an essential tool in a Java developer’s toolkit.