

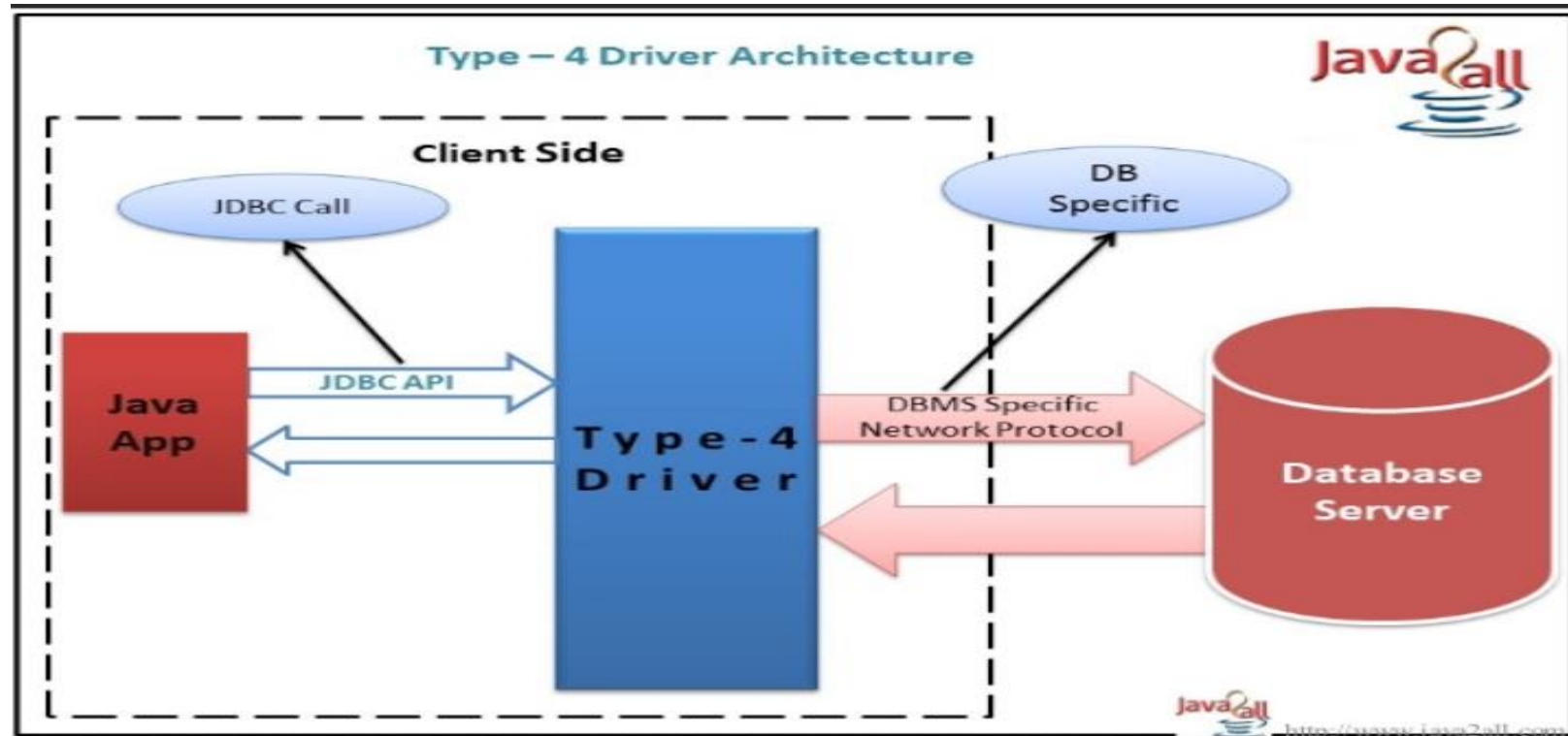


JDBC Insights

BY NEWFOUND SYSTEMS, NOV-12-2020

JDBC Insights

- **JDBC**
JDBC stands for **J**ava **D**ata**B**ase **C**onnectivity. JDBC is a Java API to connect and execute database queries. JDBC API uses JDBC drivers to connect with database. Each vendor will have their own JDBC Drivers for Client Exchange of Data.
- **What is an API**
API (**A**pplication **P**rogramming **I**nterface) is a service that describes on features of a product or software for communications. It represents on how client can exchange data.
- **Popular Database(s)**
 - Oracle, Informix, MySQL, Microsoft SQL Server, H2, SQLite



What is Database and Table

- **Database**

A database is an organized collection of table(s). Each table is collection of column(s). Table data can then be easily queried or manipulated using SQL Statements over JDBC API.

- **Table**

Data is logically organized in a row-and-column format similar to a spreadsheet. Each row represents a unique record, and each column represents a field in the record.

e.g.

Excel File is a Database, and Each Sheet is a Table.

customer_id	fname	lname	open_dt	addr_ln_1	addr_ln_2	addr_ln_3	city_nm	state_cd	ctry_cd	zip_cd
1	John	Smith	2020-11-01	1, W Wilmington Street			Phoenix	AZ	US	85260
2	Tom	Moody	2020-11-02	1, E Hillery Way			New York	NY	US	89553
3	George	Glen	2020-11-03	1, N Wilford Garden			Scottsdale	AZ	US	83456
4	Kevin	Spacey	2020-11-04	1, E Glen Way			Glendale	AZ	US	85263
5	Tracy	Guest	2020-11-05	85, Tai Seng Drive	Ang Ko Moi		Singapore	SG	SG	34560

state_id	state_cd	state_nm
1	AZ	Arizona
2	CA	California
3	NY	New York
4	SG	Singapore

ctry_id	ctry_cd	ctry_nm
1	IN	India
2	US	United States of America
3	SG	Singapore

Query

SELECT

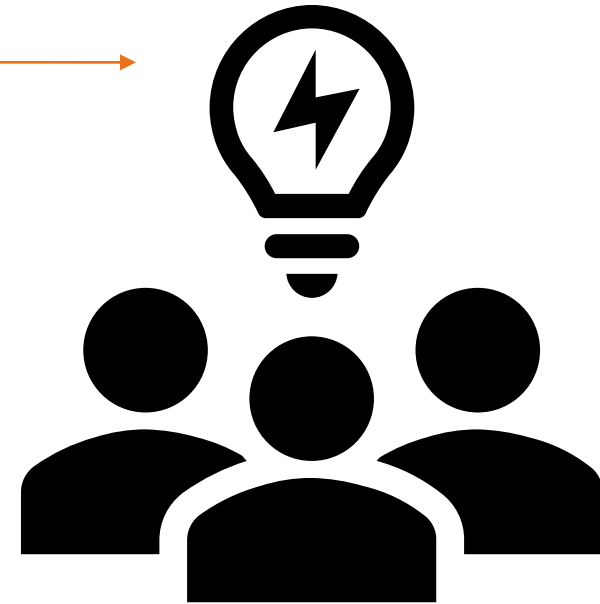
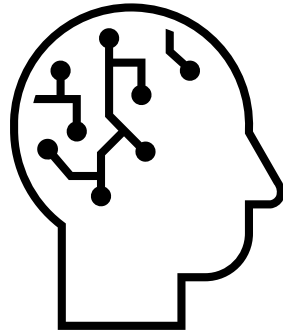
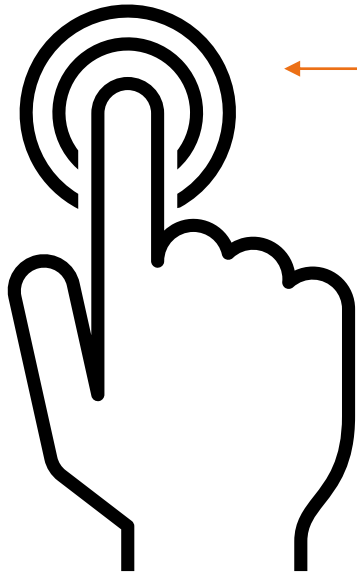
Manipulation

INSERT

UPDATE

DELETE

Q & A



Cycle

The JDBC classes are contained in Java package **java.sql** and **javax.sql**.

The **java.sql** package contains classes and interfaces for **JDBC API**.

- Driver interface
- Connection interface
- PreparedStatement interface
- ResultSet interface

JDBC connections support creating and executing statements. Data manipulation statements such as *CREATE*, *INSERT*, *UPDATE* and *DELETE*, or query statements like *SELECT*.

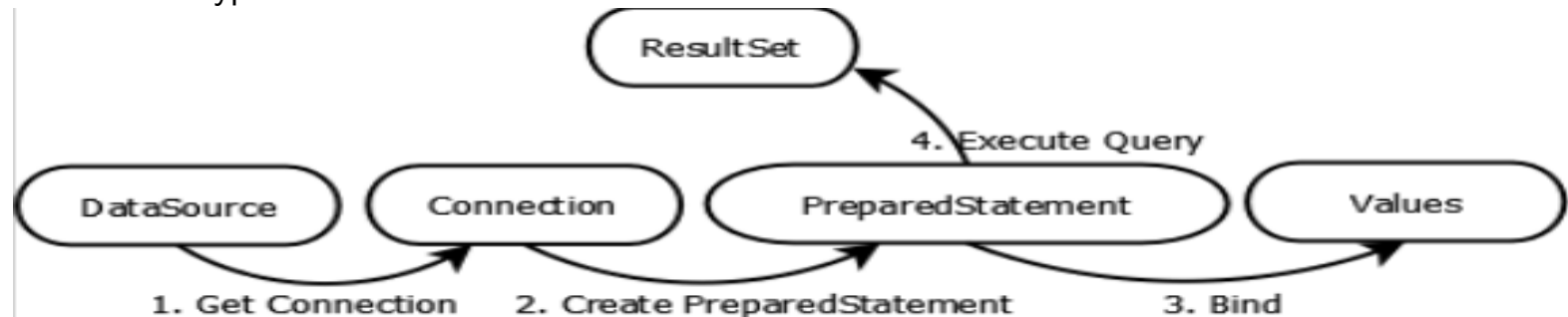
Statement – the statement is sent to the database server each and every time.

PreparedStatement – the statement is cached and then the execution path is pre-determined on the database server allowing it to be executed multiple times in an efficient manner.

CallableStatement – used for executing stored procedures on the database.

Update statements such as *INSERT*, *UPDATE* and *DELETE* return an update count that indicates how many rows were affected in the database. These statements do not return any other information.

Query statements return a JDBC row result set. The row result set is used to iterate data rows. Individual columns in a row are retrieved either by name or by column number. There may be any number of rows in the result set. Row Result has meta information about column name and associated types



Data Types

- Char
- Decimal
- Varchar
- NVarChar
- Datetime
- Date
- Boolean
- Integer

How to Create Database in SQLite

- Install SQLite from <https://sqlite.org/download.html>
- Assuming Database to be Created in D:\WebService\Project\JdbcExample\database
- SQLite e.g. is Installed in D:\SQLite-330

Create Database

```
D:\> cd D:\WebService\Project\JdbcExample\database\
```

```
D:\> \sqlite\sqlite-330\sqlite3 customer.db
```

This will create a BLANK Database *customer.db*

SQLite version 3.33.0 2020-08-14 13:23:32

Enter ".help" for usage hints.

```
sqlite> CREATE TABLE customer (  
    [customer_id] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,  
    [ctry_cd] [char](2) NOT NULL,  
    [phone_no] [varchar](18) NOT NULL,  
    [phone_type] [varchar](12) NOT NULL,  
    [email_ad] [varchar](65) NOT NULL,  
    [customer_guid] [uniqueidentifier] NOT NULL,  
    [update_token_expiry_ts] [datetime] NULL,  
    [update_token] [varchar](65) NULL,  
    [init_insert_ts] [datetime] NOT NULL,  
    [last_mdfy_ts] [datetime] NOT NULL,  
    [last_mdfy_user] [varchar](65) NOT NULL,  
    [last_mdfy_prog] [varchar](65) NOT NULL  
);
```

```
sqlite> SELECT * FROM customer;
```

This is to ensure table is created and you can select though no records

```
sqlite> .schema TO List Schema
```

```
sqlite> CONTROL-C – To Quit
```


JDBC Action

a) Register Driver Class

When a Java application needs a database connection, one of the `DriverManager.getConnection()` methods is used to create a JDBC connection.

The URL depends on particular database and JDBC driver. It will always begin with the "jdbc:" which is the protocol, but the rest is up to the Vendor specific.

```
Connection conn = DriverManager.getConnection(JDBC_URL, DB_USER, DB_PASSWORD);
```

e.g.

```
jdbc:sqlserver://localhost:1433;databaseName=testdb;integratedSecurity=true;
```

```
jdbc:sqlserver://[serverName[instanceName][:portNumber]][;property=value[;property=value]]
```

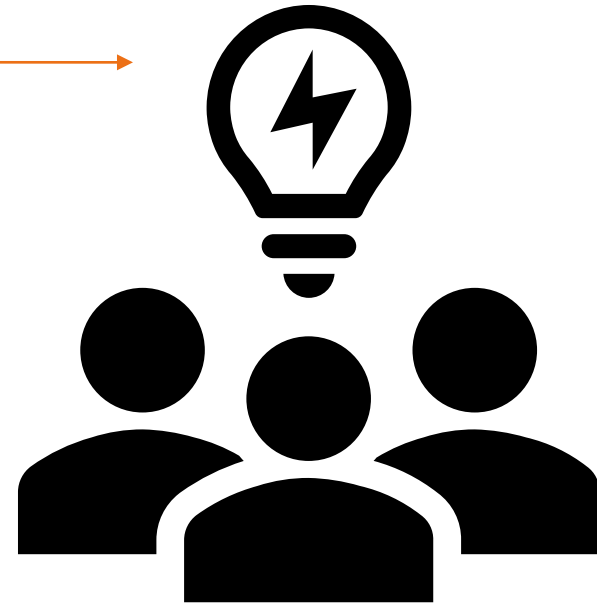
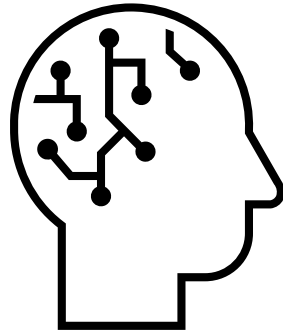
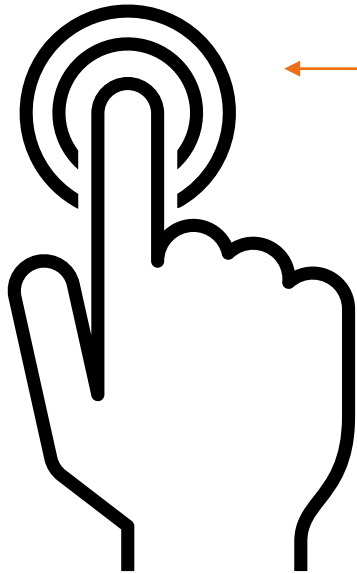
serverName: Host name or the IP address of the machine on which database server is running.

instanceName: Name of the instance to connect to on a serverName. If this parameter is not specified, the default instance is used.

portNumber: The default port number for connecting to SQL server 1433. In case this parameter is missing, the default port is used.

property=value: This parameter specifies one or more additional connection properties. To see the properties specific to the database server, you need to google vendor specifics

Q & A



b) Connection & Close

```
/**
 * Get Connection
 *
 * @return
 * @throws SQLException
 */
public Connection getConnect() throws SQLException {
    System.out.println("---Connecting to Db...");
    Connection conn = DriverManager.getConnection(JDBC_URL, DB_USER,
        DB_PASSWORD);
    return conn;
}
```

```
/**
 * Close Connection
 *
 * @param conn
 * @throws SQLException
 */
public void close(Connection conn) throws SQLException {
    System.out.println("---Closing Db Connection...");
    conn.close();
}
```

c) CRUD Process –

C(reate)

R(read)

U(pdate)

D(DELETE)

Create

```
/**
 * Get Connection
 */
Connection conn = DbService.getConnect();
String sql = " INSERT INTO customer (ctry_cd, phone_no, phone_type, email_ad,
customer_guid, last_mdfy_user, "
+ " last_mdfy_prog, update_token, update_token_expiry_ts, init_insert_ts,
last_mdfy_ts) "
+ " VALUES (?, ?, ?, ?, ?, ?, ?, ?, CURRENT_TIMESTAMP, CURRENT_TIMESTAMP,
CURRENT_TIMESTAMP) ";
/**
 * Prepare Statement
 */
PreparedStatement pstmt = conn.prepareStatement(sql);
/**
 * Set Values for INSERT
 */
pstmt.setString(1, customer.getCtry_cd());
pstmt.setString(2, customer.getPhone_no());
pstmt.setString(3, customer.getPhone_type());
pstmt.setString(4, customer.getEmail_ad());
pstmt.setString(5, customer.getCustomer_guid());

pstmt.setString(6, customer.getLast_mdfy_user());
pstmt.setString(7, customer.getLast_mdfy_prog());

pstmt.setString(8, customer.getUpdate_token());

int rowsUpdated = pstmt.executeUpdate();
success = rowsUpdated > 0;
```

CRUD Process –

C(reate)

R(read)

U(pdate)

D(elete)

Read

```
/**
 * Get Connection
 */
Connection conn = DbService.getConnect();
String sql = " SELECT customer_id, ctry_cd, email_ad, phone_no, customer_guid FROM
customer ";
/**
 * Prepare Statement
 */
PreparedStatement pstmt = conn.prepareStatement(sql);
/**
 * Get Result Set
 */
ResultSet rs = pstmt.executeQuery();
Customer customer = null;
while (rs.next()) {
    customer = new Customer();

    customer.setCustomer_id(rs.getLong("customer_id"));
    customer.setCtry_cd(rs.getString("ctry_cd"));
    customer.setEmail_ad(rs.getString("email_ad"));
    /**
     * You can also use Query Column Number - Not Recommended though
     */
    customer.setCustomer_guid(rs.getString(5));

    System.out.println(customer);
}
```

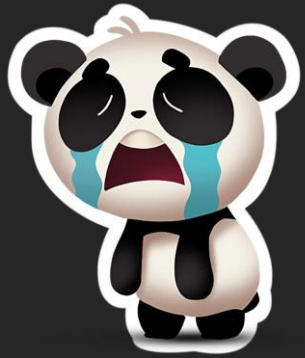
CRUD Process - Update

```
/**
 * Get Connection
 */
Connection conn = DbService.getConnect();
String sql = " UPDATE customer SET phone_no = ?, last_mdfy_ts =
CURRENT_TIMESTAMP WHERE email_ad = ? ";
/**
 * Prepare Statement
 */
PreparedStatement pstmt = conn.prepareStatement(sql);
/**
 * Set Value for Delete
 */
pstmt.setString(1, phoneNo);
pstmt.setString(2, emailAd);

int rowsUpdated = pstmt.executeUpdate();
success = rowsUpdated > 0;
```

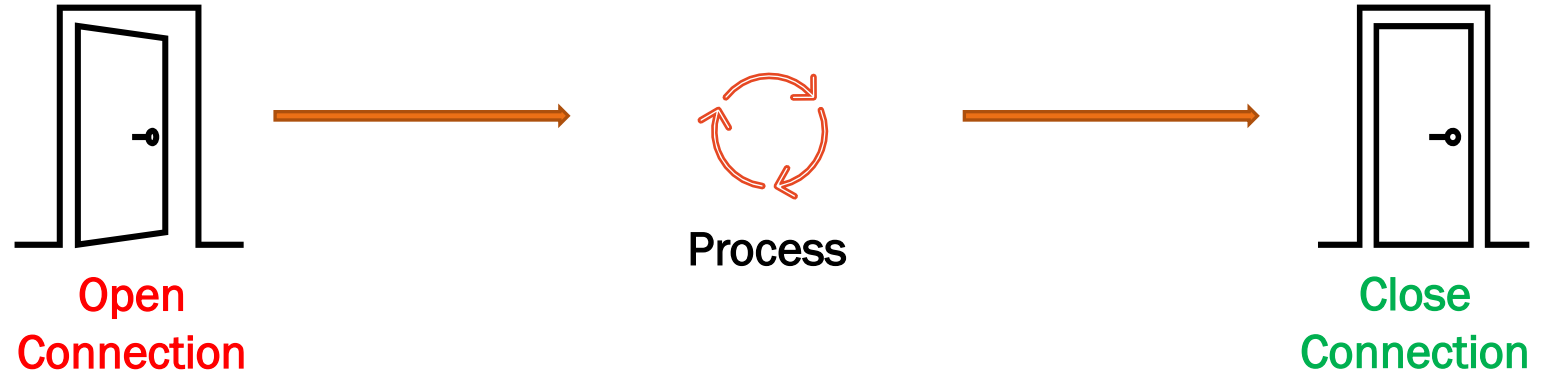
CRUD Process - Delete

```
/**
 * Get Connection
 */
Connection conn = DbService.getConnect();
String sql = " DELETE FROM customer WHERE email_ad = ? ";
/**
 * Prepare Statement
 */
PreparedStatement pstmt = conn.prepareStatement(sql);
/**
 * Set Value for Delete
 */
pstmt.setString(1, emailAd);
int rowsUpdated = pstmt.executeUpdate();
success = rowsUpdated > 0;
```



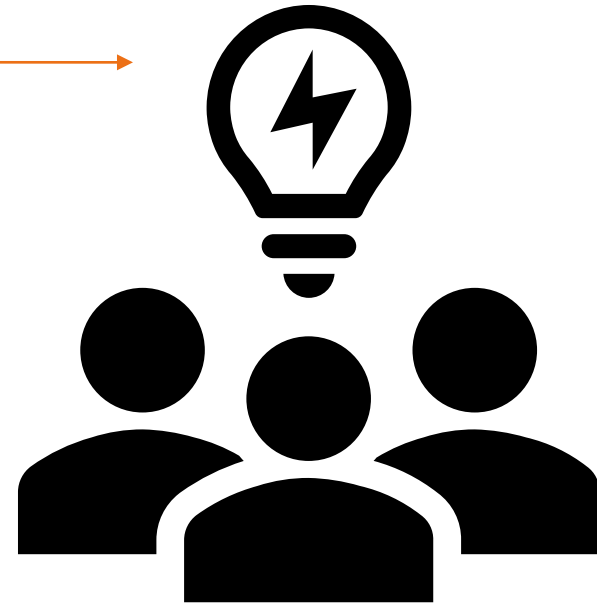
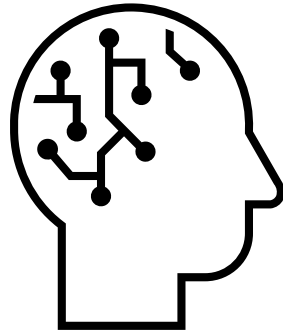
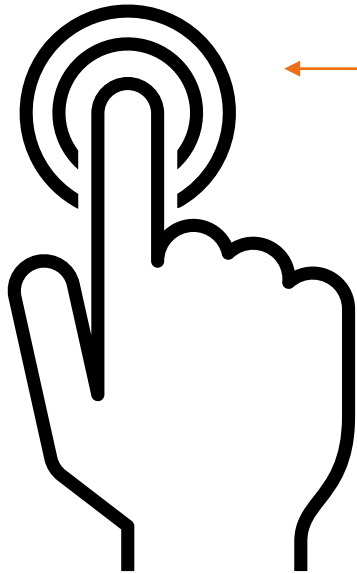
REMEMBER

To CLOSE Database
Connection Finally



```
/**
 * Close Connection
 *
 * @param conn
 * @throws SQLException
 */
public void close(Connection conn) throws SQLException {
    System.out.println("---Closing Db Connection...");
    conn.close();
}
```


Q & A



Database Transaction

A **database transaction** symbolizes a unit of work performed within a [database management system](#) against a data set(s) and treated in a logical work. A transaction generally represents change to data table(s) like *UPDATE* and/or *DELETE* and/or *INSERT*.

In a database management system, a transaction is a single unit of logic or work, sometimes made up of multiple operations. e.g: Transfer from one bank account to another: the complete transaction requires subtracting the amount to be transferred from one account and adding that same amount to the other.

Example

```
boolean success = false;
Boolean autoCommit = false; // Commit on Demand
Connection conn = DbService.getConnect(autoCommit);
/**
 * Add Customer
 */
success = add(conn, customer);
if(success) {
    /**
     * Update Customer
     */
    success = update(conn, "test@somewhere.com", "1-800-CALL-HELP");
}
if (success) {
    System.out.println("Commit Transaction...");
    conn.commit()
} else {
    System.out.println("Rollback Transaction...");
    conn.rollback();
}
conn.close();
```

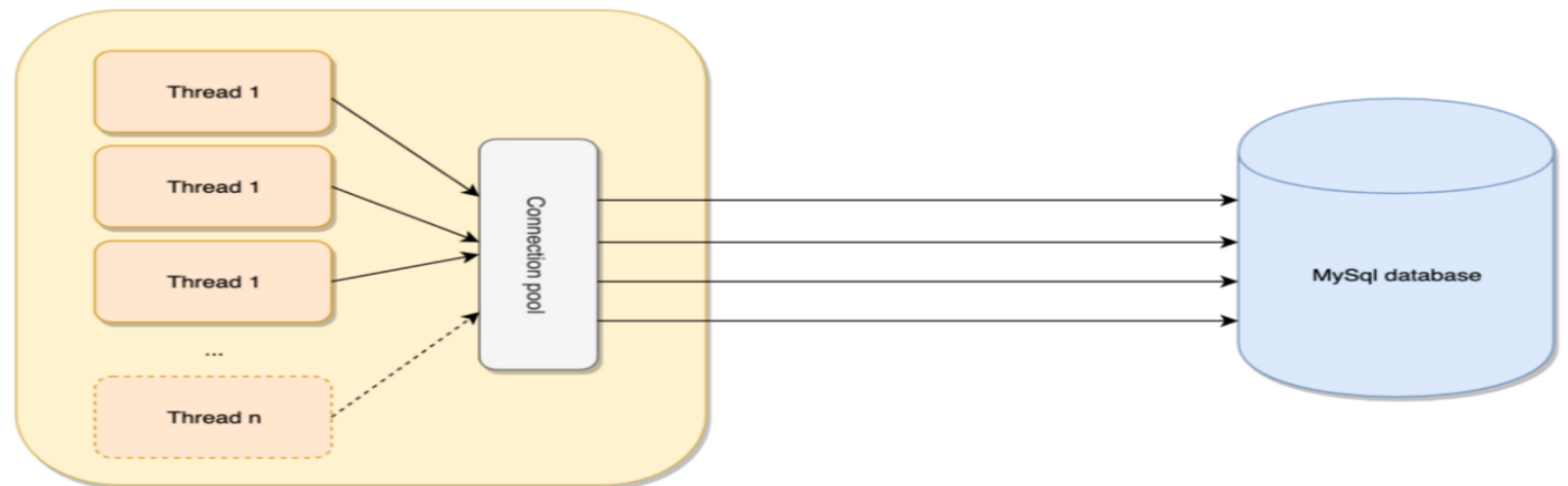
Database Connection Pool

Traditional Database Connection Trip

1. Opening a connection to the database using the database driver
2. Opening a [TCP socket](#) for reading/writing data
3. Reading / writing data over the socket
4. Closing the connection
5. Closing the socket

A **connection pool** is a cache of **database connection** objects. The objects represent physical **database connections** that can be used by an application to **connect** to a **database**. At run time, the application requests a **connection** from the **pool**.

Implementing a database connection container, which allows us to reuse a number of existing connections, we can effectively save the cost of performing a huge number of expensive database trips, hence boosting the overall performance of our database-driven applications.

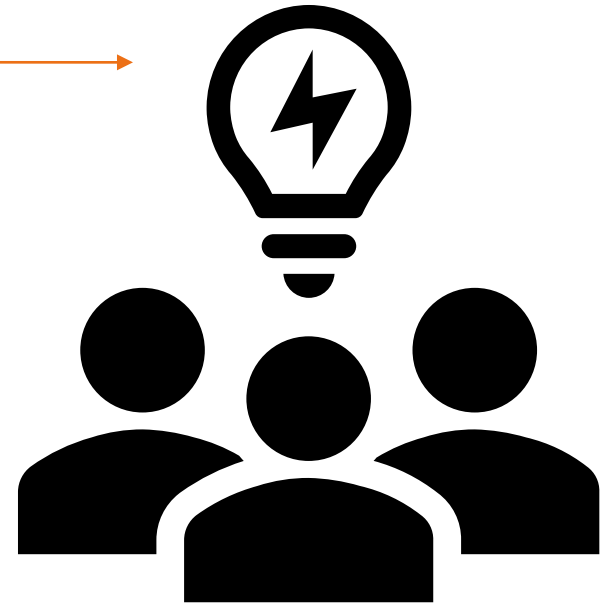
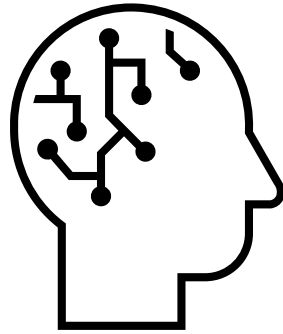
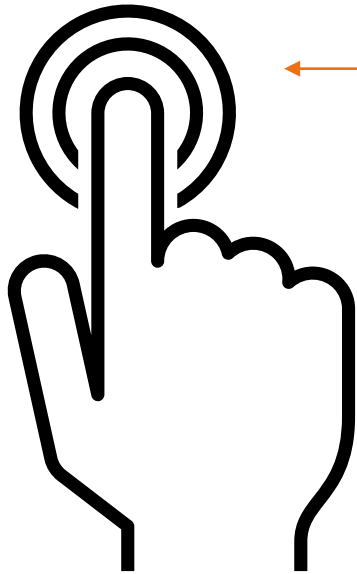


DB connection pooling

Popular Connection Pool Frameworks

- Apache Commons DBCP
- HikariCP – Most Preferred for Light Weight and Fast
- C3PO

Q & A



Introduction to JPA (Java Persistence API)

The Java Persistence API ([JPA](#)) is a Java specification for accessing, persisting, and managing data between Java Entity Classes and a Relational Database.

JPA allows [POJO](#) (Plain Old Java Objects) to be easily persisted without requiring the classes to implement any interfaces or methods

In our Example ***Customer.java*** is a POJO which can be converted to Entity Class for Object Relational Map

A [JavaBean](#) is a POJO that is [serializable](#), has a no-argument [constructor](#), and allows access to properties using [getter and setter methods](#) that follow a simple naming convention.

Entity Bean

```
@Entity
@Table(name = "country")
/**
    Using Lombok which is a java library tool which is used to minimize/remove the
    boilerplate code. @Data will generate Constructor, Getter / Setter / toString() by
    Default
*/
@Data
public class Country implements Serializable {

    private static final long serialVersionUID = 9120997940945205046L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long country_id;

    private String ctry_cd;
    private String ctry_nm;

    @Temporal(TemporalType.TIMESTAMP)
    private Date last_mdfy_ts;

    private String last_mdfy_prog;
    private String last_mdfy_user;
}
```


Query Entity *Country, Customer* Using JPA with Spring Boot

```
/**
 * Find by Country Code
 *
 * @param ctry_cd
 * @return
 */
@Query(" SELECT o FROM Country o WHERE o.ctrtry_cd = ?1 ")
@Cacheable("country")
Country findByCtryCd(@Param("ctrtry_cd") String ctry_cd);

/**
 * Find All
 *
 * @return
 */
@Query(" SELECT o FROM Country o ORDER By o.ctrtry_cd ")
List<Country> findAll();

/**
 * Find Customer by Email
 *
 * @param email_ad
 * @return
 */
@Query(" SELECT o FROM Customer o WHERE o.email_ad = ?1 ")
Customer findByEmail(@Param("email_ad") String email_ad);
```

Traditional JDBC Transaction

```
import java.sql.Connection;

Connection connection = dataSource.getConnection(); // (1)

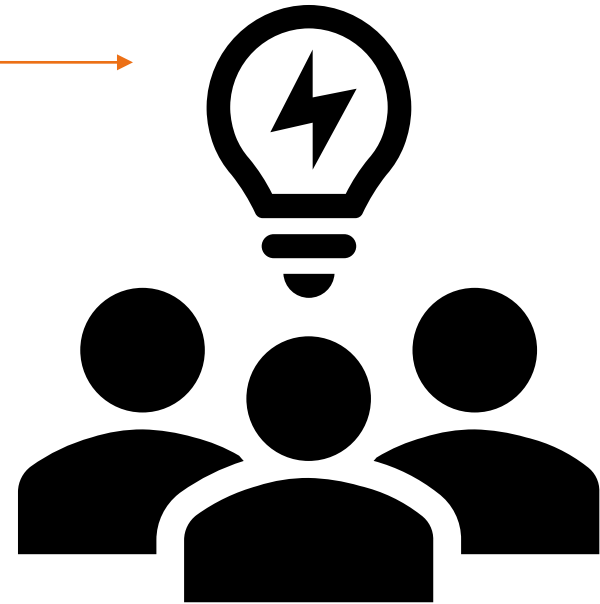
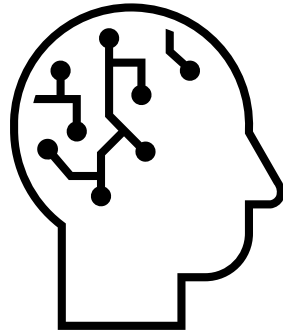
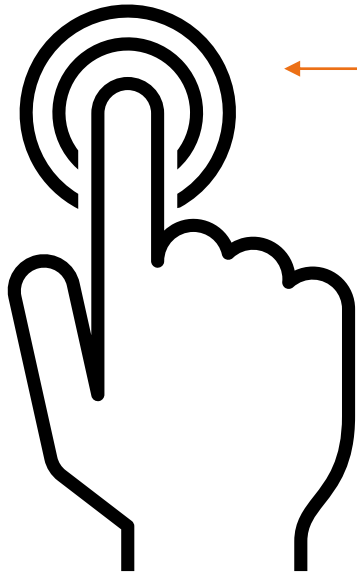
try (connection) {
    connection.setAutoCommit(false); // (2)
    // execute some SQL statements...
    connection.commit(); // (3)
} catch (SQLException e) {
    connection.rollback(); // (4)
}
```

Spring JPA Transaction using Annotations

```
@Transactional(propagation = Propagation.REQUIRES_NEW, readOnly = false,  
rollbackFor = { Exception.class })  
public synchronized boolean persist(Customer customer,  
    CustomerAddress customerAddress) throws RuntimeException, Exception {  
    boolean success = false;  
    try {  
        customer = customerRepository.saveAndFlush(customer);  
        customerAddress =  
customerAddressRepository.saveAndFlush(customerAddress);  
        if (customerAddress != null) {  
            updateCustomerLastMdfyTs(customer);  
            success = true;  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    return success;  
}
```

```
@Transactional(propagation = Propagation.MANDATORY, readOnly = false, rollbackFor  
= { Exception.class })  
private boolean updateCustomerLastMdfyTs(Customer customer) {  
    customer.setLast_mdfy_ts(new Date());  
    customer.setLast_mdfy_prog(CLAZZ);  
    /**  
     * Update  
     */  
    customerRepository.save(customer);  
    return true;  
}
```

Q & A



Project Technologies

- Eclipse IDE - <https://www.eclipse.org/>
- Apache Maven as Build Tool - <https://maven.apache.org>
- GitHub – Source Control Management - <https://github.com>
- DBVisualizer – Database tool - <https://dbvis.com>
- SQLite - <https://sqlite.org/>
- Java JDK 1.8x

References

Project Source with Presentation (ppt) - <https://github.com/newfound-systems/JdbcExample>

- Build Tool Maven - <https://maven.apache.org/>
- Log4J Tutorials - <https://mkyong.com/logging/apache-log4j-2-tutorials/>
- SQLite - <https://sqlite.org/>
- SQLite Tutorials - <https://www.tutorialspoint.com/sqlite/index.htm>
- JDBC Wiki - https://en.wikipedia.org/wiki/Java_Database_Connectivity
- Spring Boot - <https://spring.io/projects/spring-boot>
- Spring Boot with JPA - <https://www.javatpoint.com/spring-boot-jpa>

Thank You



Stay and Keep Safe



Project is posted in github

<https://github.com/newfound-systems>

<https://github.com/newfound-systems/JdbcExample>