# Basics of Events and streams (Queus)

## How Event-Driven Architectures Benefit from Stream Processing

Suppose you’re an app developer or an ML/AI engineer. In that case, it’s safe to say that some of your top priorities include making your apps faster, more reliable, easily scalable, and decoupled. One of the best ways to achieve that and improve any app’s performance is to create an event-driven architecture that uses event streams.

## What Is an Event-Driven Architecture?

An [event-driven architecture](https://cloud.google.com/eventarc/docs/event-driven-architectures) is a software design paradigm that enables the communication between back-end systems.

Some of its key components are producers and consumers. Producers or publishers are apps that generate events (state changes), while consumers or subscribers are apps that consume events and perform various tasks.

They communicate via messages, which message brokers distribute based on subscriptions to certain channels or topics, subsets of events in a category. In addition, they can use message queues or event stream processing to transmit messages.

## What Are Message Queues?

Message queues are components of early event-driven architectures. Serverless architectures and microservices that don’t use event streams still use queues to send messages to specific consumers.

With message queues, producers are familiar with all their consumers. Therefore, when they send a message to a queue, they target specific consumers, alerting them about a certain event. Consumers then obtain and process the message before performing the necessary task.

Most message brokers that use message queues follow the FIFO (First-In, First-Out) order. Thus, each of them deletes every message once the consumer retrieves it.

## What Is Event Stream Processing?

[Event stream processing](https://www.bmc.com/blogs/event-stream-processing/) includes processing real-time events in an [event-driven architecture](https://pandio.com/blog/what-is-event-streaming-why-is-it-growing-in-popularity/) to make it asynchronous, decouple its services, and ensure easy scaling. It’s about processing events continuously and storing them for later retrieval.

One of the most popular models for transferring event messages between back-end systems in event-driven architectures is the pub/sub-design pattern. It enables message brokers to receive events from various sources (e.g., IoT devices, sensors, databases, apps, etc.) and send messages to all subscribers within a specific channel.

Thanks to this pattern, producers and consumers don’t need to be familiar with one another. Their services are decoupled, so they perform tasks independently. There’s no specific targeting, and multiple consumers can receive numerous simultaneous messages.

Apart from decoupling services, another critical feature of event stream processing is temporal durability. For example, brokers don’t delete messages in streams after consumers process them, so they can retrieve them anytime from an event history.

## Message Queues vs. Event Streams

Both event streams and message queues have their advantages, so what should you use in your event-driven architecture?

Let’s go over the most important benefits of each.

### Benefits of message queues

Message queues can work great when a message broker knows a particular routing structure. It can help control high data volumes and streamline batch processing.

When a system uses exactly-once processing semantics to ensure that every message is delivered only once and prevent data duplicates, message queueing is the way to go.

For instance, it’s a perfect choice when you need to connect legacy systems with certain dependencies or when you need to process events in a system that uses different back-end and front-end programming languages.

Ecommerce websites are the most common application of message queues. This is because they have an established routing logic that message brokers are familiar with to ensure decoupling and asynchronous task handling in event-driven architectures.

Some of the most popular open-source, queue-based message brokers are [RabbitMQ](https://pandio.com/blog/pulsar-vs-rabbitmq-which-messaging-bus-should-you-pick/) and ActiveMQ.

# Database integration with JDBC

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

* JDBC-ODBC Bridge Driver,
* Native Driver,
* Network Protocol Driver, and
* Thin Driver

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The **java.sql** package contains classes and interfaces for JDBC API. A list of popular interfaces of JDBC API are given below:

* Driver interface
* Connection interface
* Statement interface
* PreparedStatement interface
* CallableStatement interface
* ResultSet interface
* ResultSetMetaData interface
* DatabaseMetaData interface
* RowSet interface

A list of popular *classes* of JDBC API are given below:

* DriverManager class
* Blob class
* Clob class
* Types class

Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

# **Java Database Connectivity with 5 Steps**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:   * Register the Driver class * Create connection * Create statement * Execute queries * Close connection   Register the Driver class:  1) Register the driver class   |  | | --- | | The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class. |   Syntax of forName() method   * **public** **static** **void** forName(String className)**throws** ClassNotFoundException  **JDBC 4.0, explicitly registering the driver is optional.**Example to register the OracleDriver class Here, Java program is loading oracle driver to esteblish database connection.   * Class.forName("oracle.jdbc.driver.OracleDriver");   2) Create the connection object   |  | | --- | | The **getConnection()** method of DriverManager class is used to establish connection with the database. |   Syntax of getConnection() method   * 1) **public** **static** Connection getConnection(String url)**throws** SQLException * 2) **public** **static** Connection getConnection(String url,String name,String password) * **throws** SQLException   Example to establish connection with the Oracle database   * Connection con=DriverManager.getConnection( * "jdbc:oracle:thin:@localhost:1521:xe","system","password");   3) Create the Statement object   |  | | --- | | The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database. |   Syntax of createStatement() method   * **public** Statement createStatement()**throws** SQLException   Example to create the statement object   * Statement stmt=con.createStatement();   4) Execute the query   |  | | --- | | The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table. |   Syntax of executeQuery() method   * **public** ResultSet executeQuery(String sql)**throws** SQLException   Example to execute query   * ResultSet rs=stmt.executeQuery("select \* from emp"); * **while**(rs.next()){ * System.out.println(rs.getInt(1)+" "+rs.getString(2)); * }   5) Close the connection object   |  | | --- | | By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection. |   Syntax of close() method   * **public** **void** close()**throws** SQLException   Example to close connection   * con.close();   Statements:  There are different types of statements that are used in JDBC as follows:   * Create Statement * Prepared Statement * Callable Statement * **Create Statement:**From the connection interface, you can create the object for this interface. It is generally used for general**–**purpose access to databases and is useful while using static SQL statements at runtime.   **Syntax:**  Statement statement = connection.createStatement();  **2. Prepared Statement**represents a recompiled SQL statement, that can be executed many times. This accepts parameterized SQL queries. In this, “?” is used instead of the parameter, one can pass the parameter dynamically by using the methods of PREPARED STATEMENT at run time.  **Illustration:**  Considering in the people database if there is a need to INSERT some values, SQL statements such as these are used:  INSERT INTO people VALUES ("Ayan",25);  INSERT INTO people VALUES("Kriya",32);  To do the same in Java, one may use Prepared Statements and set the values in the ? holders, setXXX() of a prepared statement is used as shown:  String query = "INSERT INTO people(name, age)VALUES(?, ?)";  Statement pstmt = con.prepareStatement(query);  pstmt.setString(1,"Ayan");  ptstmt.setInt(2,25);  // where pstmt is an object name   * **Callable Statement**are stored procedures which are a group of statements that we compile in the database for some task, they are beneficial when we are dealing with multiple tables with complex scenario & rather than sending multiple queries to the database, we can sendtherequired data to the stored procedure & lower the logic executed in the database server itself. The Callable Statement interface provided by JDBC API helps in executing stored procedures.   **Syntax:** To prepare a CallableStatement  CallableStatement cstmt = con.prepareCall("{call Procedure\_name(?, ?}");  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **What is Meta Data?** Metadata is data about the data or documentation about the information which is required by the users. In data warehousing, metadata is one of the essential aspects.  Metadata includes the following:   * The location and descriptions of warehouse systems and components. * Names, definitions, structures, and content of data-warehouse and end-users views. * Identification of authoritative data sources. * Integration and transformation rules used to populate data. * Integration and transformation rules used to deliver information to end-user analytical tools. * Subscription information for information delivery to analysis subscribers. * Metrics used to analyze warehouses usage and performance. * Security authorizations, access control list, etc.   Metadata is used for building, maintaining, managing, and using the data warehouses. Metadata allow users access to help understand the content and find data. **examples of metadata are:**  * The table of content and the index in a book may be treated metadata for the book. * Another examples of metadata are data about the tables and figures in a report like this book. A table (which is a record) has a name (e.g., table titles), and there are column names of the tables that may be treated metadata. The figures also have titles or names.  **Types of Metadata** Metadata in a data warehouse fall into three major parts:   * Operational Metadata * Extraction and Transformation Metadata * End-User Metadata |