Producer-Consumer Using `BlockingQueue` with SQLite and Java

Overview

This project demonstrates the **Producer-Consumer concurrency pattern** using Java's high-level `BlockingQueue` mechanism. The objective is to simulate how data flows between two decoupled systems in real time:

- The **Producer** reads order data from a JSON file (source container).
- The **Consumer** writes this data into an SQLite database (destination container).
- A **shared buffer** using `BlockingQueue<OrderEntity>` enables synchronized and thread-safe communication between them.

This setup mimics real-world data ingestion systems and is built using standard Java tools, with a focus on multithreading, file I/O, inter-thread coordination, and persistent storage.

Concept: Producer–Consumer Pattern

The **Producer–Consumer problem** is a classic concurrency design where:

- The **Producer** generates data and adds it to a shared buffer.
- The **Consumer** retrieves and processes data from that buffer.
- Synchronization ensures correctness and prevents race conditions, data loss, or deadlocks.

In this project, we use Java's `BlockingQueue` to manage buffer logic. It automatically blocks the producer when the queue is full and blocks the consumer when the queue is empty, removing the need for low-level synchronization (`wait()/notify()`).

How It Works

1. **Producer**
   - Reads order data from a JSON file.
   - Converts each JSON object to an `OrderEntity`.
   - Inserts the order into a `BlockingQueue`.
2. **Consumer**
   - Continuously polls the queue for `OrderEntity` objects.

- Inserts each into the `orders` table of the SQLite database.
- Terminates when it receives a special sentinel object (`orderId = -1`, also called a poison pill).

3. **Shared Buffer**
   - Implemented as `ArrayBlockingQueue<OrderEntity>`.
   - Acts as a fixed-size thread-safe container between producer and consumer.
4. **DBManager**
   - Manages SQLite connection and schema.
   - Handles all insert and fetch operations via JDBC.
5. **OrderEntity**
   - A simple data class (POJO) that holds fields such as `orderId`, `customerName`, `status`, `amount`, and `orderDate`.

Project Structure

```
Producer_Consumer_BlockingQueue/

  pom.xml
  orders.json                      # Source JSON data
  orders.db                        # SQLite DB file (auto-created)
  src/
      main/
          java/org/example/
              Main.java                # Entry point for producer-consumer execution
              OrderEntity.java         # POJO representing an order
              Producer.java            # Producer logic
              Consumer.java            # Consumer logic
              DBManager.java           # SQLite interaction layer
              SharedQueue.java         # BlockingQueue setup
          resources/orders.json         # Input data
      test/java/org/example/
          OrderEntityTest.java          # Unit test for OrderEntity
          DBManagerTest.java            # Unit test for DB operations
          ProducerConsumerIntegrationTest.java  # End-to-end flow test
```

Containers Used

alt text

Figure 1: alt text

alt text

Figure 2: alt text

Type	Description	Implementation
Source Container	JSON file (`orders.json`)	Read using `Files.readAllBytes()`
Destination	SQLite DB (`orders.db`)	Written using JDBC prepared statements
Shared Buffer	`BlockingQueue<OrderEntity>`	Thread-safe communication between threads

Running the Project

1. Compile and Run

Make sure you're in the root directory of the Maven project:

```
mvn compile
mvn exec:java -Dexec.mainClass="org.example.Main"
```

You can also run the `Main.java` file from your IDE (e.g., IntelliJ IDEA or Eclipse).

---

2. Run All Tests

Use the following Maven command to run unit and integration tests:

```
mvn test
```

This executes all test cases to verify correctness of entity classes, database operations, and the end-to-end producer-consumer workflow.

Output

Running Main file

Running tests

Test Overview

The test suite ensures that each part of the Producer-Consumer system works as intended, from entity formatting to database interactions and full execution flow using a blocking queue.

Test Class	Purpose
`OrderEntityTest`	Validates that the `toString()` method in `OrderEntity` produces the correct string format.
`DBManagerTest`	Tests inserting orders into the SQLite database and verifies them using raw JDBC queries.
`ProducerConsumerIntegrationTest`	Simulates the full producer-consumer workflow: reading from JSON, queuing, and database insertion. It checks that the `Producer` correctly adds parsed orders and a poison pill to the queue, and that the `Consumer` inserts them accurately into the database.

Dependencies

Dependencies are declared in `pom.xml` and managed via Maven:

```
<dependencies>
    <dependency>
        <groupId>org.xerial</groupId>
        <artifactId>sqlite-jdbc</artifactId>
        <version>3.44.1.0</version>
    </dependency>
    <dependency>
        <groupId>org.json</groupId>
        <artifactId>json</artifactId>
        <version>20231013</version>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Native Access Warning (Optional)

If you encounter a warning like:

```
WARNING: java.lang.System::load has been called by org.sqlite.SQLiteJDBCLoader...
```

You can suppress it by adding the following JVM argument:

```
--enable-native-access=ALL-UNNAMED
```

IntelliJ Setup

- Run > Edit Configurations > Add VM Options:

  ```
  --enable-native-access=ALL-UNNAMED
  ```

Completion

This project provides a complete, tested implementation of the **Producer-Consumer pattern** using a high-level `BlockingQueue` in Java, along with JSON-based input and persistent SQLite storage. It demonstrates real-time data transfer between decoupled systems using clean synchronization and proper testing.