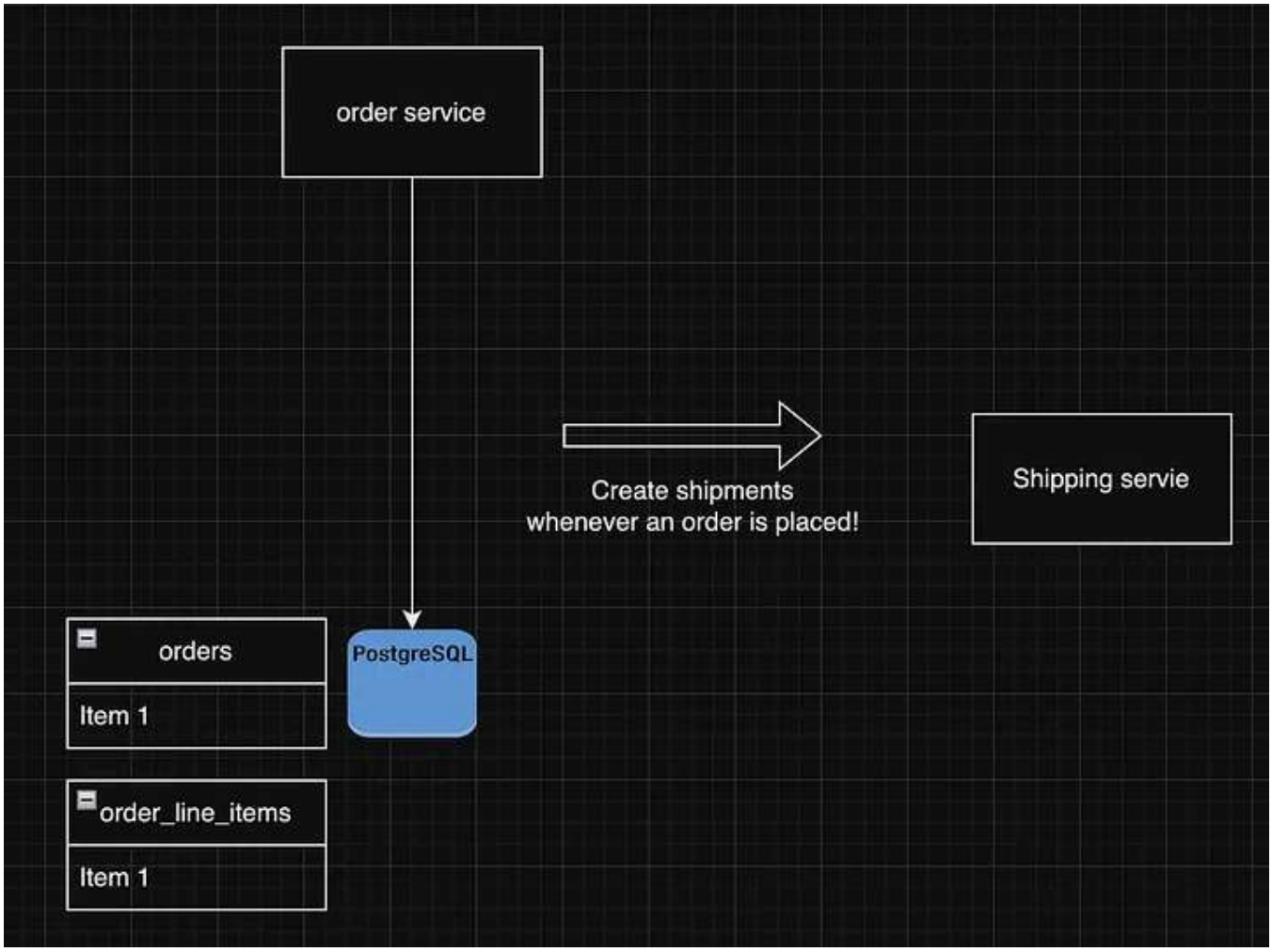
Change Data Capture (CDC) using Debezium in a SpringBoot application

Ishan Soni · Follow

8 min read · Apr 19, 2024

Change Data Capture (CDC) is a technique used in distributed architectures to capture and propagate data changes across different systems. It allows for real-time data synchronization and enables services to react to changes in data without the need for constant polling or querying.

Example use case:



Our e-commerce application requirement

Our e-commerce application has an order service that manages orders and a shipping service that manages shipments. Our requirement is to create shipments out of an order whenever an order is created.

Solution 1.0

Make an RPC call to the shipment service from the order service whenever an order is created. This solution is fragile and does not scale well. See more details at — What is RabbitMQ and why do you need a Message Broker?

Solution 1.1

Whenever an order is created, send an OrderCreated event to a messaging broker. The shipping service can then read this message and create shipments. Sample code in order service:

```
@Transactional
public void createOrder(OrderRequest orderRequest) {
    Order order = toOrder(orderRequest);
    orderRepository.save(order);
```

```
            messagePublisher.publish(new OrderCreated(/* .. */));
        }
```

The problem with this design is that the OrderCreated event is published before the transaction is guaranteed to commit. If the transaction that saves the order is rolled back for any reason after the event is published, the shipping service would still receive the OrderCreated event and attempt to create a shipment for an order that doesn't exist in the database.

Solution 1.2

```
@Transactional
public void createOrder(OrderRequest orderRequest) {
    Order order = toOrder(orderRequest);
    orderRepository.save(order);

    applicationEventPublisher.publishEvent(new OrderCreated(/* .. */));
}
```

```
@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
public void listen(OrderCreated orderCreated) {
    messagePublisher.publish(orderCreated);
}
```

This improved design uses Spring's event publishing mechanism within the scope of a transaction. By using TransactionalEventListener with the phase set to **TransactionPhase.AFTER_COMMIT** (default), you ensure that the OrderCreated message is only published to the messaging broker after the transaction has successfully committed. This means the shipping service will only receive the message if the order is successfully saved, preventing it from creating shipments for orders that were rolled back. Read more at — Using Spring Application Events within Transactional Contexts.

Even though this solution is optimal for most use cases, there are still some problems you could encounter:

1. What happens if there is a loss of connectivity (N/W issue) to the message broker?

2. What if the application goes down while the listener is executing?

> *In both the above cases, we might not be able to reliably send messages to the message broker. Also, there is no retry capability baked into this design. What if a consumer(s) is not able to consume a message due to a data/code issue? You spot it and quickly do a hotfix, but you may also want to retry the messages that were generated by the producer which the consumer(s) was not able to consume before the hotfix!*

Solution 1.3 — The Transactional Outbox pattern

I've discussed the theory around CDC and transactional outbox in detail here — Saga, CDC with Transactional Inbox/Outbox. This pattern involves storing the events to be published in a local database table (the "outbox") as part of the same database transaction that updates the business aggregate. After the transaction commits, a separate process or service retrieves the events from the outbox and publishes them to the message broker. This ensures that the event publishing is reliable and can survive application restarts and message broker connectivity issues. This will also allow you to retry messages when needed.

Sample code:

```
@Transactional
public void createOrder(OrderRequest orderRequest) {
    Order order = toOrder(orderRequest);
    orderRepository.save(order);

    outboxRepository.save(new OutboxEntry(new OrderCreated(/ .. /)));
}
```

```
// A separate service or scheduled job
public void relayOutboxMessages() {
    List<OutboxEntry> entries = outboxRepository.findUnpublishedEntries();
    for (OutboxEntry entry : entries) {
        try {
            messagePublisher.publish(entry.getEvent());
            entry.markAsPublished();
        } catch (Exception e) {
            // Handle connectivity issues, e.g., by retrying later
            entry.markError(e);
```

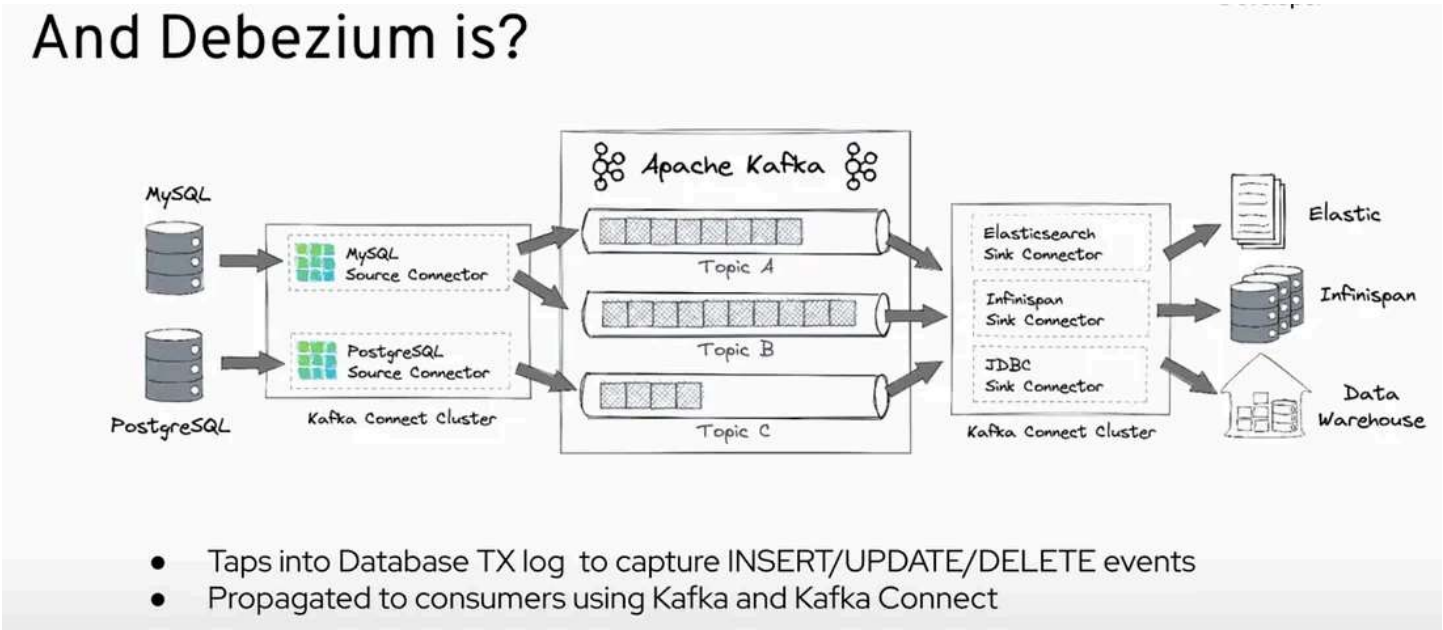
```
            } finally {
                outboxRepository.save(entry);
            }
        }
    }
```

> *To retry messages, mark their status as "UNPUBLISHED". Make sure you implement idempotent consumers — The Transactional Inbox pattern can help you here!*

Note — There are still some things you'll have to take care of. If your publishing job runs inside the same service, think about how you'll handle multiple job executions in a clustered environment. What if your application is deployed to a K8 cluster and you have multiple pods or autoscaling enabled? Some solutions include — distributed locking (using **Redis**, **ShedLock**, etc.) to make sure only 1 job instance runs at any point of time!

. . .

Using Debezium as the CDC platform



Debezium. source — https://youtu.be/0d_I2aQm4LE?si=Hw6fGWGu9EL7NBuQ

Debezium is an open-source CDC platform that captures and streams database changes. It supports various databases, including PostgreSQL, MySQL, SQL Server, and MongoDB. Debezium provides connectors for each of these databases, making it easier to integrate CDC into your application.

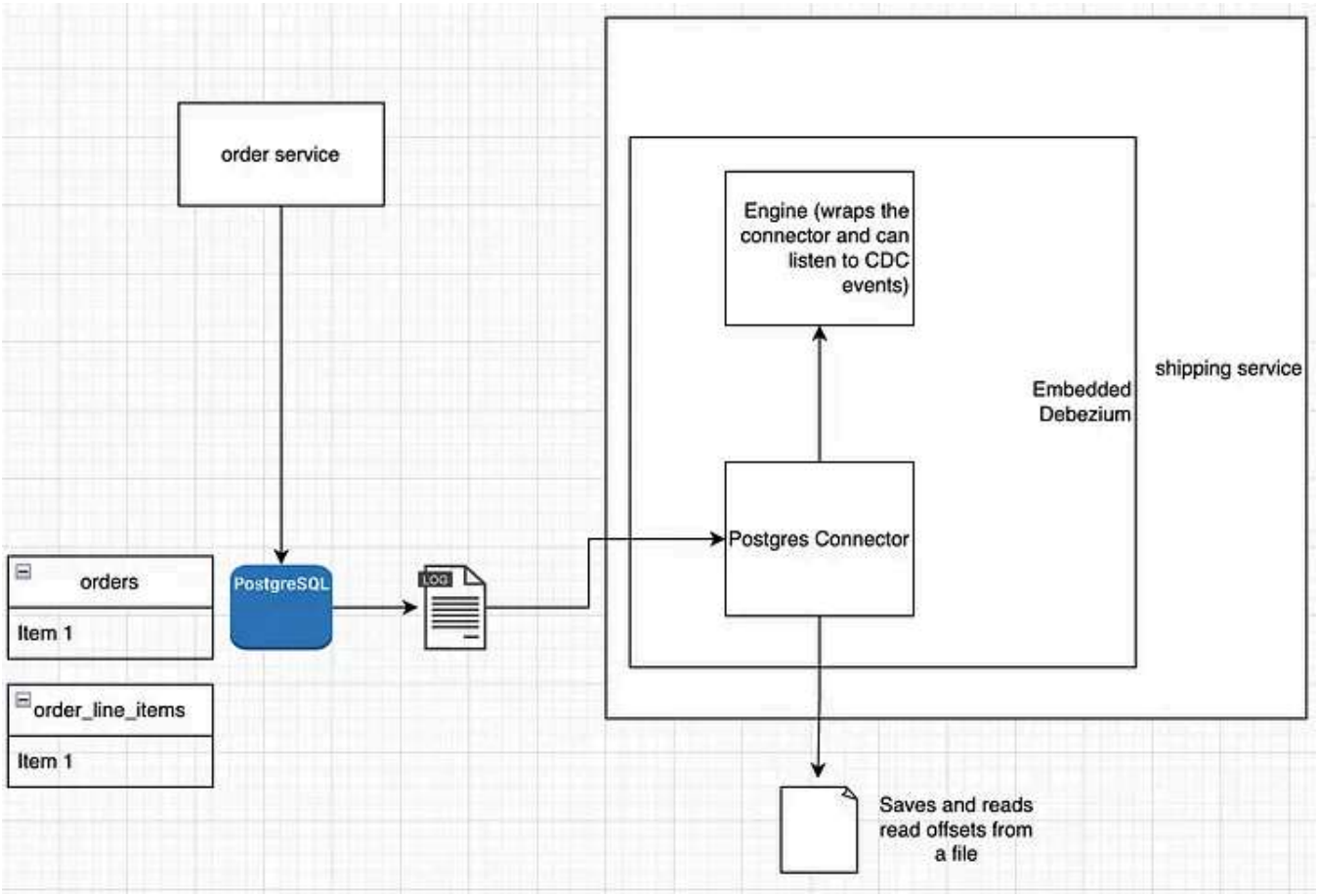
i.e Debezium captures row/document-level changes from database transaction logs/oplog using connectors and transforms them into a CDC event stream. Each CDC event has:

- Old and new row/document state (**before**, **after**)
- Metadata (**txId**, etc.)
- Operation type (**op** — create(c), update(u), delete(d))
- timestamp (**ts_ms**)

You can tap into this stream and react to changes!

Solution 2.0 — Embedded Debezium

Embedded Debezium refers to a mode of running Debezium where the change data capture (CDC) connectors are directly integrated into your application, rather than being deployed as separate services. This allows the application to read changes from the database's transaction log and handle those changes internally without the need for a standalone Kafka Connect cluster. It's particularly useful for applications that don't require the full scale and fault tolerance provided by Kafka Connect or for those looking to minimize infrastructure complexity.



Embedded Debezium HLD

Add the following dependencies in your shipping service:

```
<dependency>
    <groupId>io.debezium</groupId>
    <artifactId>debezium-api</artifactId>
    <version>2.5.2.Final</version>
</dependency>
<dependency>
    <groupId>io.debezium</groupId>
    <artifactId>debezium-embedded</artifactId>
    <version>2.5.2.Final</version>
</dependency>
<dependency>
    <groupId>io.debezium</groupId>
    <artifactId>debezium-connector-postgres</artifactId>
    <version>2.5.2.Final</version>
</dependency>
```

Add the debezium source connector configuration:

```
@Configuration
public class DebeziumConnectorConfig {

    @Value("${db.host}")
    private String dbHost;
    @Value("${db.port}")
    private String dbPort;
    @Value("${db.username}")
    private String postgresUsername;
    @Value("${db.password}")
    private String postgresPassword;
    @Value("${schema.include.list}")
    private String schemaIncludeList;
    @Value("${table.include.list}")
    private String tableIncludeList;

    @Bean
    public io.debezium.config.Configuration postgresConnector() {
        Map<String, String> configMap = new HashMap<>();

        //This sets the name of the Debezium connector instance. It's used for
        configMap.put("name", "shippingservice-cdc-connector");
        //This specifies the Java class for the connector. Debezium uses this t
```

```
        configMap.put("connector.class", "io.debezium.connector.postgresql.Post

        //This sets the Java class that Debezium uses to store the progress of
        /* Not-working
        In this case, it's using a JDBC-based store, which means it will store
        configMap.put("offset.storage", "io.debezium.storage.jdbc.offset.JdbcOf
        //This is the JDBC URL for the database where Debezium stores the conne
        configMap.put("offset.storage.jdbc.url", dbUrl);
        configMap.put("offset.storage.jdbc.user", postgresUsername);
        configMap.put("offset.storage.jdbc.password", postgresPassword);
        */

        File offsetStorageTempFile = new File("offsets_.dat");
        configMap.put("offset.storage",  "org.apache.kafka.connect.storage.File
        configMap.put("offset.storage.file.filename", offsetStorageTempFile.get
        configMap.put("offset.flush.interval.ms", "60000");

        //Connect Debezium connector to the source DB
        configMap.put("database.hostname", dbHost);
        configMap.put("database.port", dbPort);
        configMap.put("database.user", postgresUsername);
        configMap.put("database.password", postgresPassword);
        configMap.put("database.dbname", "order-service");
        configMap.put("database.server.name", "order-service"); //Why is this u
        configMap.put("plugin.name", "pgoutput");

        configMap.put("schema.include.list", schemaIncludeList);
        configMap.put("table.include.list", tableIncludeList);

        configMap.put("topic.prefix", "cdc_"); //Why is this needed here?

        return io.debezium.config.Configuration.from(configMap);
    }

}
```

```
spring.application.name=shippingservice
server.port=8082

db.host=localhost
db.port=5432
db.username=postgres
db.password=password

schema.include.list=public
table.include.list=public.orders,public.order_line_items
```

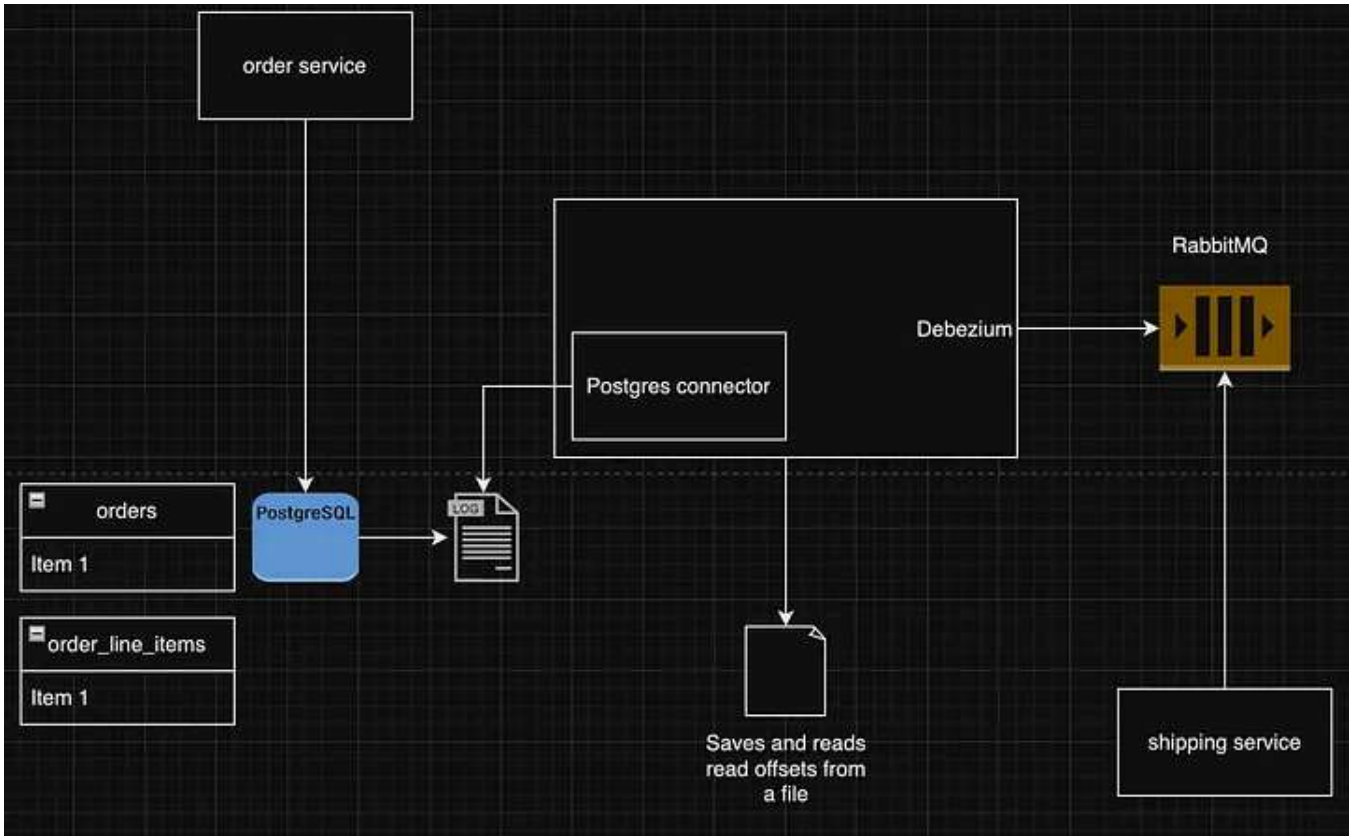
Create the engine:

```
import io.debezium.config.Configuration;
import io.debezium.engine.ChangeEvent;
import io.debezium.engine.DebeziumEngine;
import io.debezium.engine.format.Json;

...

@Slf4j
@Component
public class DebeziumSourceEventListener {

    private final Executor executor;

    //DebeziumEngine serves as an easy-to-use wrapper around any Debezium conne
    private final DebeziumEngine<ChangeEvent<String, String>> debeziumEngine;

    public DebeziumSourceEventListener(Configuration postgresConnector) {
        // Create a new single-threaded executor.
        this.executor = Executors.newSingleThreadExecutor();

        // Create a new DebeziumEngine instance.
        this.debeziumEngine = DebeziumEngine.create(Json.class)
                .using(postgresConnector.asProperties())
                //This is where your CDC events will be passed to
                .notifying(this::handleEvent)
                .build();
    }

    private void handleEvent(ChangeEvent<String, String> event) {
        System.out.println("Key: " + event.key());
        System.out.println("Value: " + event.value());
    }

    @PostConstruct
    private void start() {
        this.executor.execute(debeziumEngine);
    }

    @PreDestroy
    private void stop() throws IOException {
        if (this.debeziumEngine != null) {
            this.debeziumEngine.close();
        }
    }

}
```

You'll start receiving CDC events in the handleEvent() when any changes (create/update/delete) are made to the order or order_line_items tables.

Solution 2.1 — Standalone Debezium

Let's add a bit more resiliency to our solution. Instead of running embedded debezium, let's run a standalone debezium server instance which listens to changes in the orders and order_line_items tables and pushes these changes to a RabbitMQ exchange. Our shipping service can then listen to these CDC events from RabbitMQ.



Standalone Debezium HLD

Let's bring up our infrastructure using docker-compose. Learn more about docker and docker compose here. This will bring up a standalone debezium server as well.

```
version: "3.8"
services:
  rabbitmq:
    image: rabbitmq:3.11-management-alpine
    container_name: rabbitmq
    ports:
      - 5672:5672
      - 15672:15672
    environment:
      RABBITMQ_DEFAULT_USER: guest
      RABBITMQ_DEFAULT_PASS: guest
```

```
      RABBITMQ_DEFAULT_VHOST: /

  postgres:
    image: quay.io/debezium/example-postgres:2.1
    container_name: postgres
    volumes:
      - ~/apps/postgres-5433:/var/lib/postgresql/data
    ports:
      - 5433:5432
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=order-service

  debezium:
    image: debezium/server:2.4
    container_name: debezium-server
    ports:
      - 8080:8080
    volumes:
      - ./debezium:/debezium/conf
    depends_on:
      - rabbitmq
      - postgres
```

debezium server's application.properties

```
debezium.sink.type=rabbitmq
debezium.sink.rabbitmq.connection.host=rabbitmq
debezium.sink.rabbitmq.connection.port=5672
debezium.sink.rabbitmq.connection.username=guest
debezium.sink.rabbitmq.connection.password=guest
debezium.sink.rabbitmq.connection.virtual.host=/
debezium.sink.rabbitmq.connection.port=5672
debezium.source.connector.class=io.debezium.connector.postgresql.PostgresConnec
debezium.source.offset.storage.file.filename=offsets.dat
debezium.source.offset.flush.interval.ms=60000
debezium.source.database.hostname=postgres
debezium.source.database.port=5432
debezium.source.database.user=postgres
debezium.source.database.password=password
debezium.source.database.dbname=order-service
debezium.source.topic.prefix=orders
debezium.source.schema.include.list=public
debezium.source.table.include.list=public.orders,public.order_line_items
debezium.source.plugin.name=pgoutput
```

The Debezium server configuration provided specifies the following:

Sink Configuration:

debezium.sink.type=rabbitmq: The sink (destination) type is set to RabbitMQ, indicating that Debezium will send the captured changes to a RabbitMQ broker.

Source Configuration:

debezium.source.connector.class=io.debezium.connector.postgresql.PostgresConnector: The source connector class is set to the PostgreSQL connector, indicating that Debezium will capture changes from a PostgreSQL database.

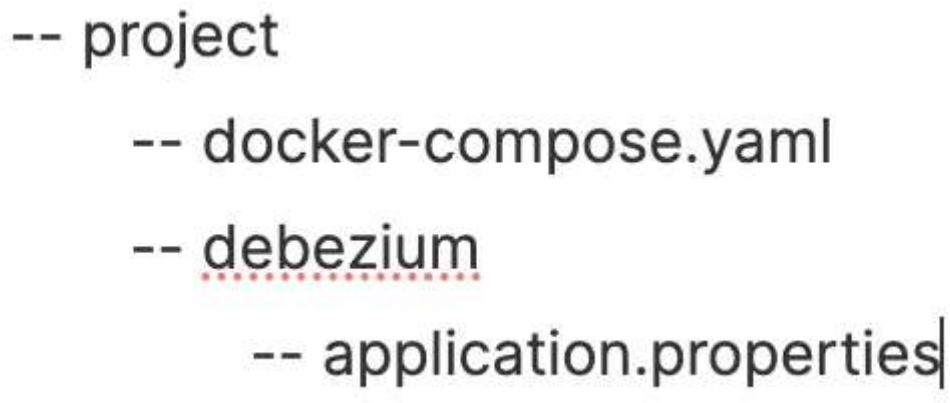
debezium.source.offset.storage.file.filename=offsets.dat: The filename for storing the offsets is offsets.dat. Offsets keep track of the last processed change in the database.

debezium.source.offset.flush.interval.ms=60000: The interval at which offsets are flushed to the storage file is set to 60000 milliseconds (or 1 minute).

debezium.source.schema.include.list=public: The schema include list is set to public, meaning only the public schema will be monitored for changes.

debezium.source.table.include.list=public.orders,public.order_line_items: The table include list specifies public.orders and public.order_line_items, indicating that only changes to these tables will be captured.

We have mounted the debezium folder in our current directory (the directory that holds the docker-compose.yaml) on the host to the container's debezium/conf directory. Here is the directory structure:

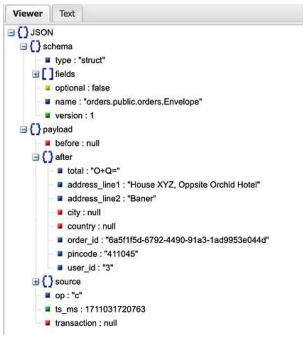


Create a consumer in your shipping service to listen to CDC events from RabbitMQ. I am using Spring Cloud Stream here. Learn more about Spring Cloud Stream!

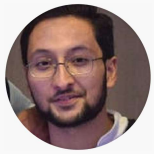
```
...
import org.springframework.messaging.Message;

@Configuration
@Slf4j
public class OrderListenerConfig {

    @Bean
    public Consumer<Message> orderListener() {
        return (message) -> {
            String payload = new String((byte[]) message.getPayload());
            log.info("Data: {}", payload);
        };
    }

}
```

```
spring.application.name=shippingservice
...

rabbitmq.host=localhost
rabbitmq.port=5672
rabbitmq.username=guest
rabbitmq.password=guest

spring.cloud.function.definition=orderListener
spring.cloud.stream.bindings.orderListener-in-0.destination=orders.public.order
spring.cloud.stream.bindings.orderListener-in-0.group=shippingservice
```

Let's try to create a new order. The CDC event is propagated to the shipping service:



Change Data Capture | Debezium | Spring Boot

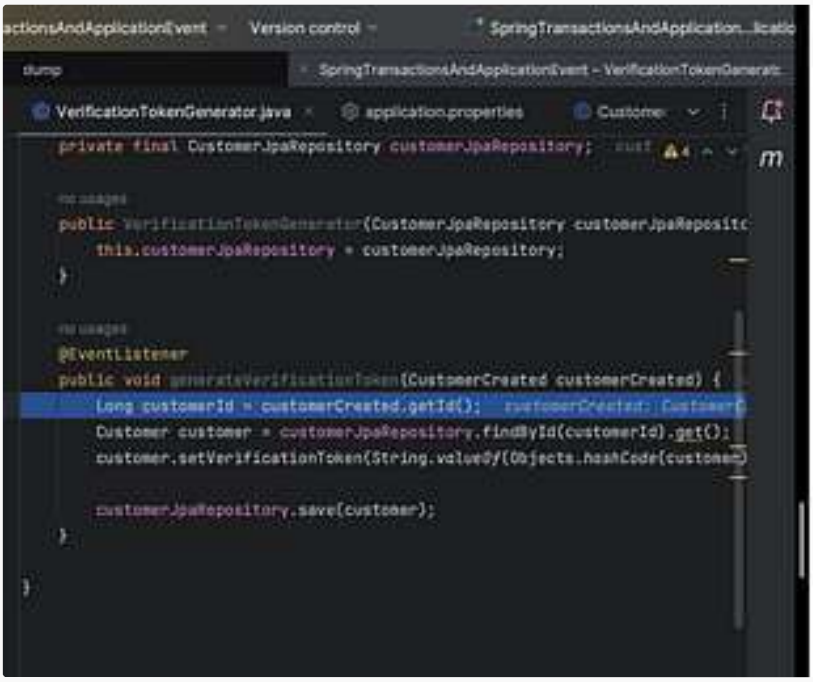


Follow

Written by Ishan Soni

86 Followers

More from Ishan Soni

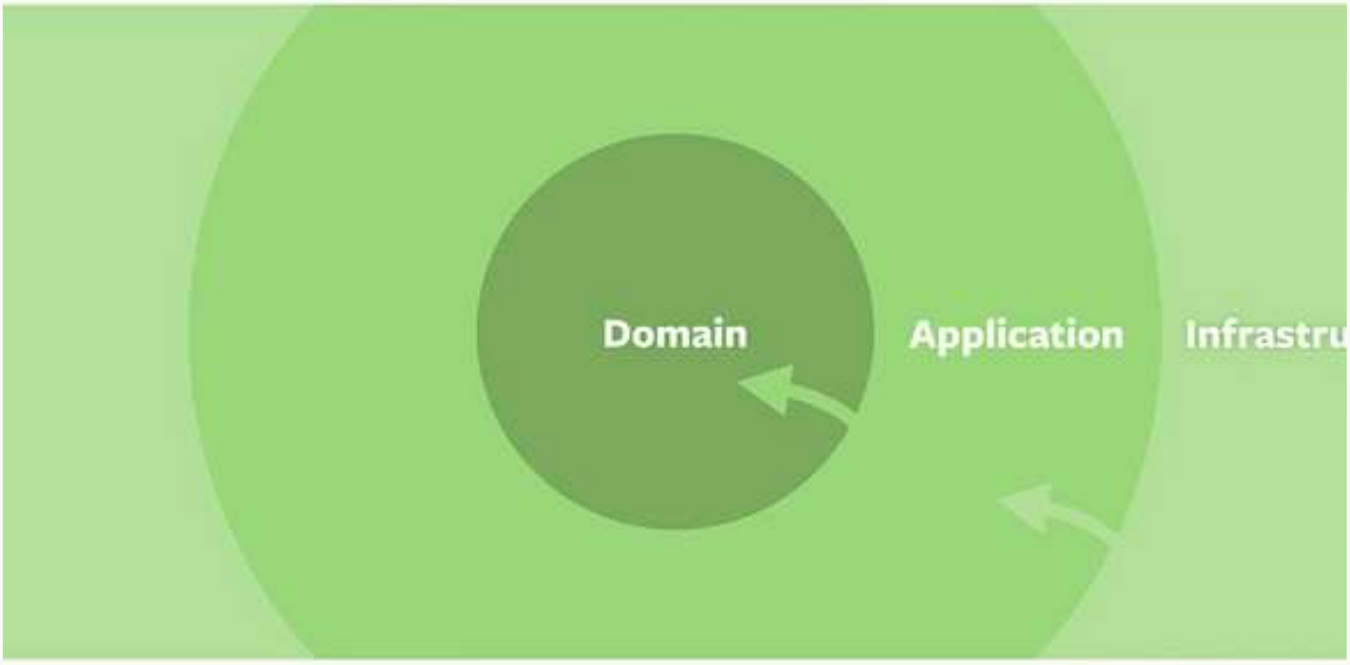


Ishan Soni

Using Spring Application Events within Transactional Contexts

This post is inspired by this talk given by Bartłomiej Słota. I highly recommend checking it out.

Dec 20, 2023 26 1

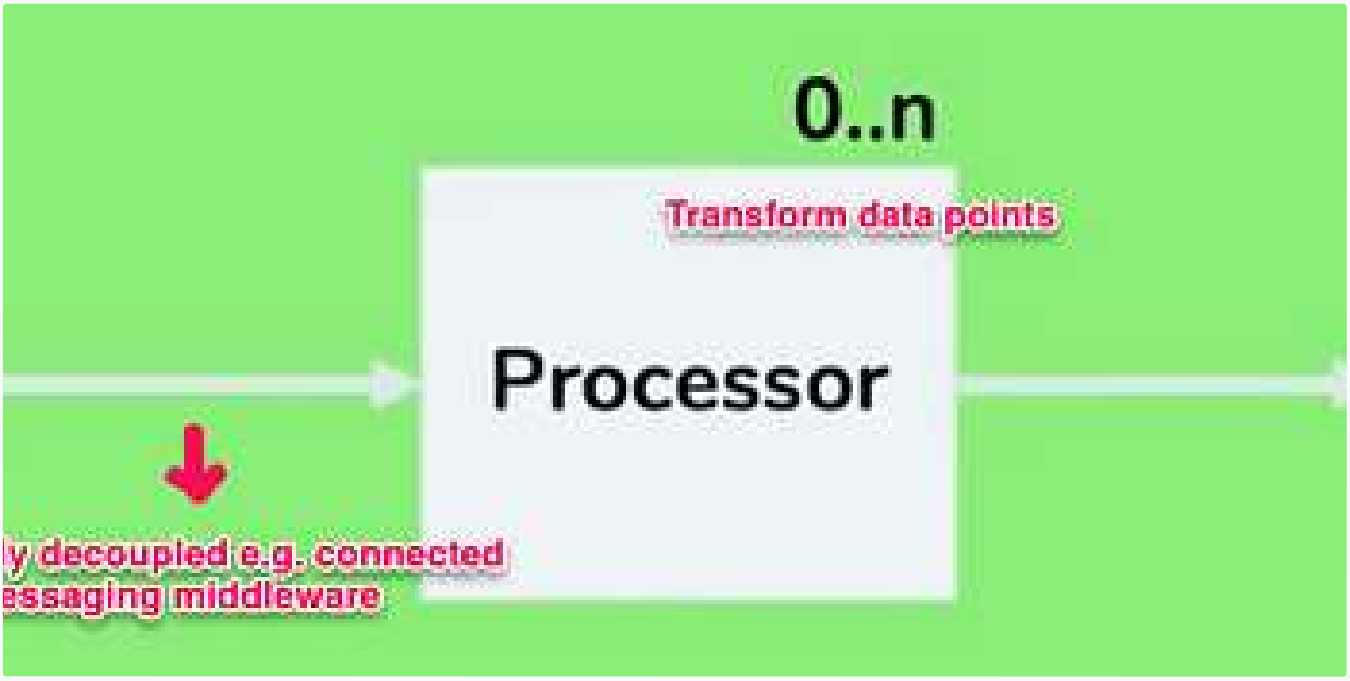


Ishan Soni

Creating a Multi-Module Monolith using Spring Modulith

Spring Modulith is an opinionated toolkit to build domain-driven, modular applications with Spring Boot. This article is based on a talk by...

May 14 19 1



Ishan Soni

Introduction to Spring Cloud Function and Spring Cloud Stream

What is Streaming and Stream processing?

Feb 8 17

Ishan Soni

Spring Data MongoDB—CRUD, Aggregations, Views and Materialized Views

Basic Setup

Jan 30 3

See all from Ishan Soni

Recommended from Medium

Krishna

Understand SOLID Principles Using Spring Boot

Introduction

Apr 11 53 1

NGU in Level Up Coding

Spring Boot 3.3: Hidden Gems Features Must-Try

Boost your development and debugging efficiency

Aug 16 118 1

Lists

Staff Picks

717 stories · 1241 saves

Stories to Help You Level-Up at Work

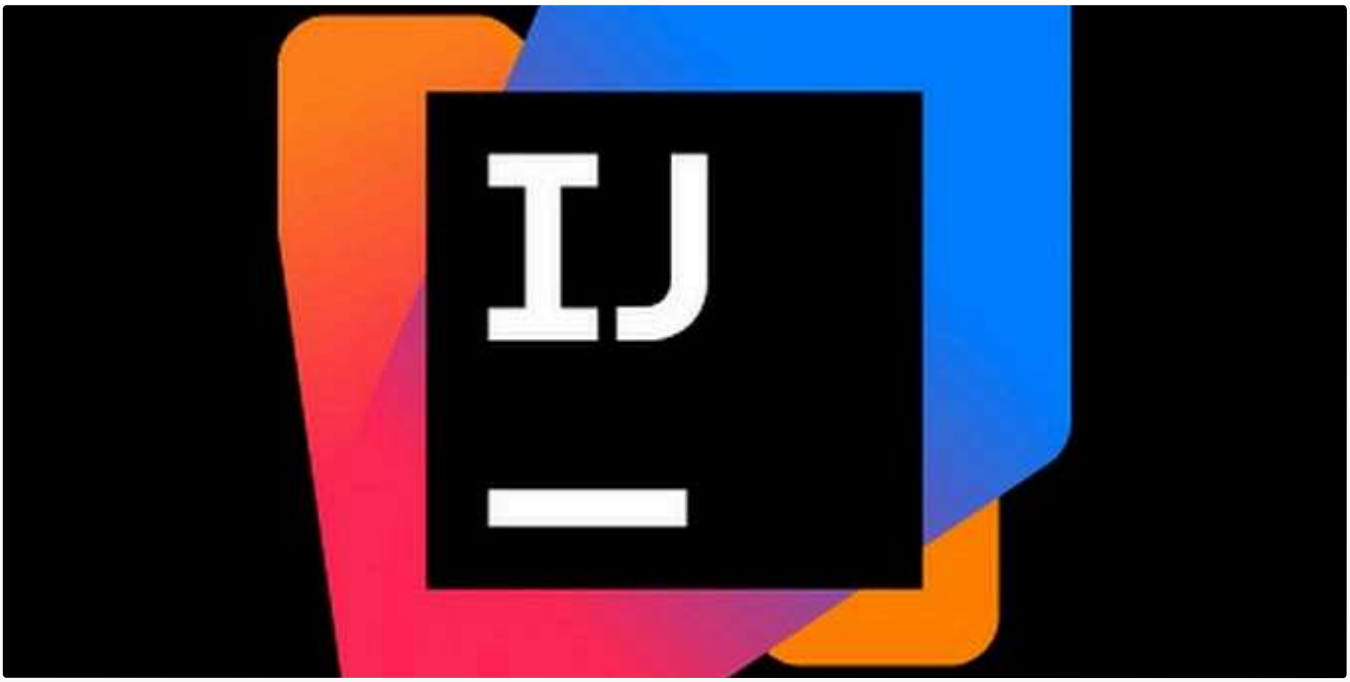
19 stories · 758 saves

Self-Improvement 101

20 stories · 2604 saves

Productivity 101

20 stories · 2240 saves



Mayank Sharma in Nerd For Tech

I discovered this amazing IntelliJ feature after 6 years.

How a random debugging session helped me discover this amazing feature.

Aug 7 344 6

Mahesh Babu in JavaToDev

Synchronizing Scheduled Tasks Across Spring Boot Instances with ShedLock

Ever found yourself in a Spring Boot project, jazzed up with scheduling features, only to hit a snag when scaling up? Picture this: as your...

Apr 21 71

Rabinarayan Patra

A Better Way to Schedule Tasks in Spring Boot: Best Practices for Asynchronous Execution

Learn how to efficiently schedule tasks in Spring Boot with async execution, and follow best practices to optimize performance and...

Aug 13 184 2

Bubu Tripathy

Semantic Validation in a Spring Boot Application

Validation involves checking that data meets certain criteria before it is processed by the system. This can include simple checks, such as...

Aug 14 47 1

See more recommendations