Change Data Capture Made Easy: Debezium Integration with Spring Boot, MongoDB and Postgres

Sofiene Ben Khemis · Follow
13 min read · Dec 10, 2023

Photo by Ross Findon on Unsplash

In today's applications, real-time data processing is more valuable than ever. Businesses need to react quickly to changing conditions and make data-driven decisions on the fly. To achieve this, capturing changes to your data as they happen is essential, and this is where Change Data Capture (CDC) comes into play.

In this article, we will explore the concept of Change Data Capture and how you can easily integrate it with Spring Boot using Debezium.

What is Change Data Capture?

Change Data Capture (CDC) is a process that identifies and captures changes made to data in a database. These changes can include inserts, updates, and deletes. CDC enables the tracking of data modifications so that applications can react to these changes in real-time. This is particularly valuable for scenarios such as:

- Analytics and Reporting: Immediate access to data changes allows for real-time reporting and analytics, giving businesses a competitive edge.
- Synchronization: Keeping multiple databases or systems in sync by replicating changes as they happen.
- Audit Trail: Maintaining a complete history of data changes for compliance and auditing purposes.

Debezium: A Change Data Capture Solution

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.

Why Use Debezium with Spring Boot and MongoDB?

Spring Boot is a popular Java framework for building web applications and microservices. MongoDB is a NoSQL database known for its flexibility and scalability. When combined with Debezium, you get a powerful trio for building real-time data-driven applications.

Here are some compelling reasons to use Debezium with Spring Boot and MongoDB:

1. Realtime Data: Debezium captures changes to MongoDB documents in real-time, allowing your Spring Boot application to react immediately to data changes.
2. Reliable Change Tracking: Debezium ensures that all changes are reliably tracked, even in the face of system failures or crashes.

3. No Impact on Source Database: Debezium captures changes without putting any additional load on your MongoDB database, making it a non-intrusive solution.

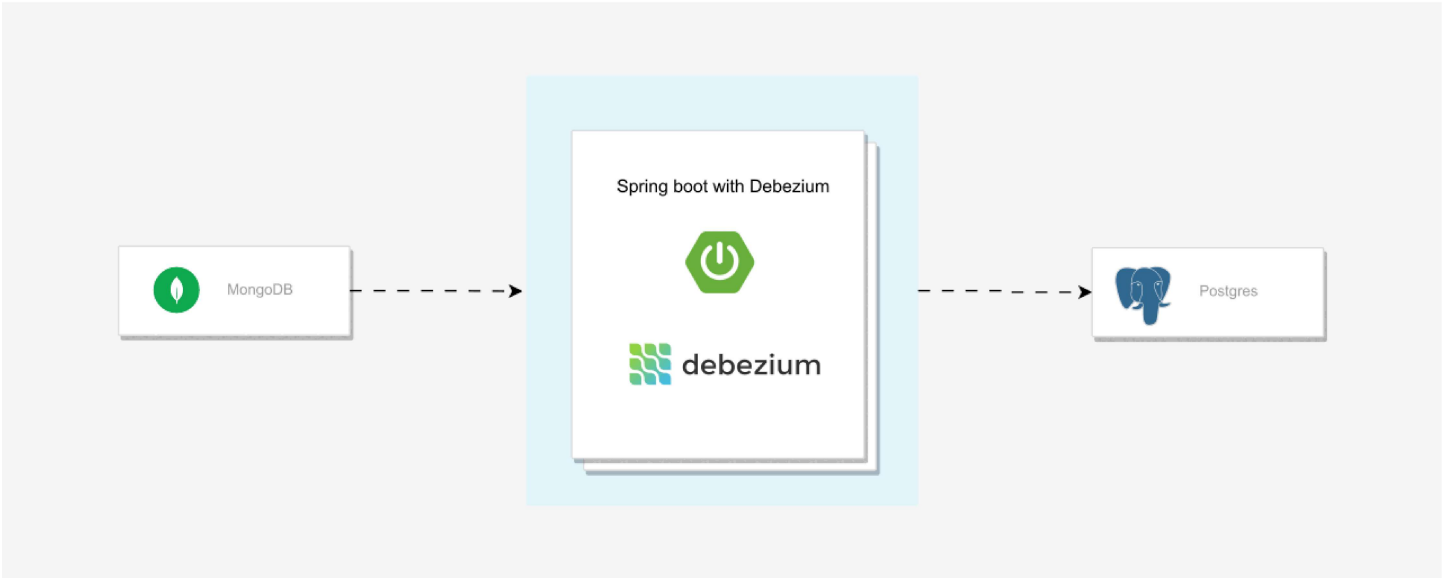
4. Easy Integration: Debezium provides a simple and straightforward integration with Spring Boot, thanks to its connectors and plugins.

Our Use Case

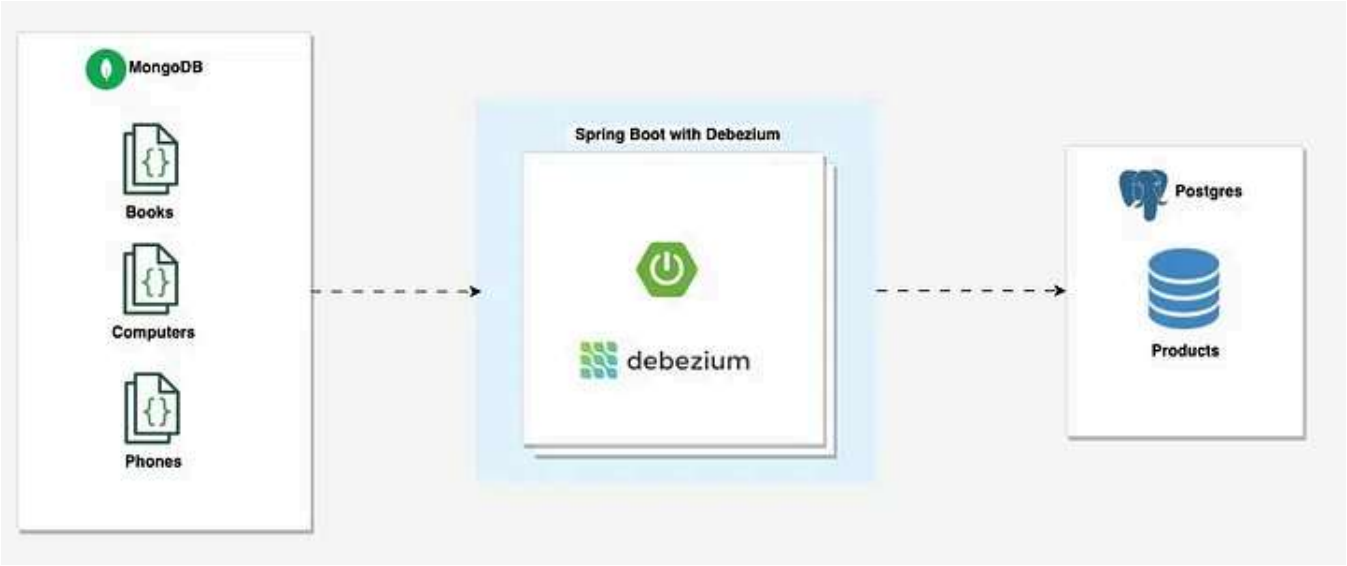
Imagine you have an e-commerce platform where all the transactional data is stored in multiple MongoDB collections. This could include user profiles, product catalogs, orders, and reviews.

However, for analytical purposes, you want to maintain a real-time analytics dashboard that requires complex queries and joins across these collections. MongoDB is not very efficient for such operations. So, you decide to use PostgreSQL, which is more suitable for complex queries and joins.

In this tutorial, We'll demonstrate how to synchronize two databases, MongoDB and Postgres, using Debezium. Whenever an operation is performed on MongoDB, it will trigger a synchronization process in the Postgres database.



For simplicity, this tutorial we will create three MongoDB collections representing three different products: phones, PCs, and books. These collections will be synchronized with the PostgreSQL products table.

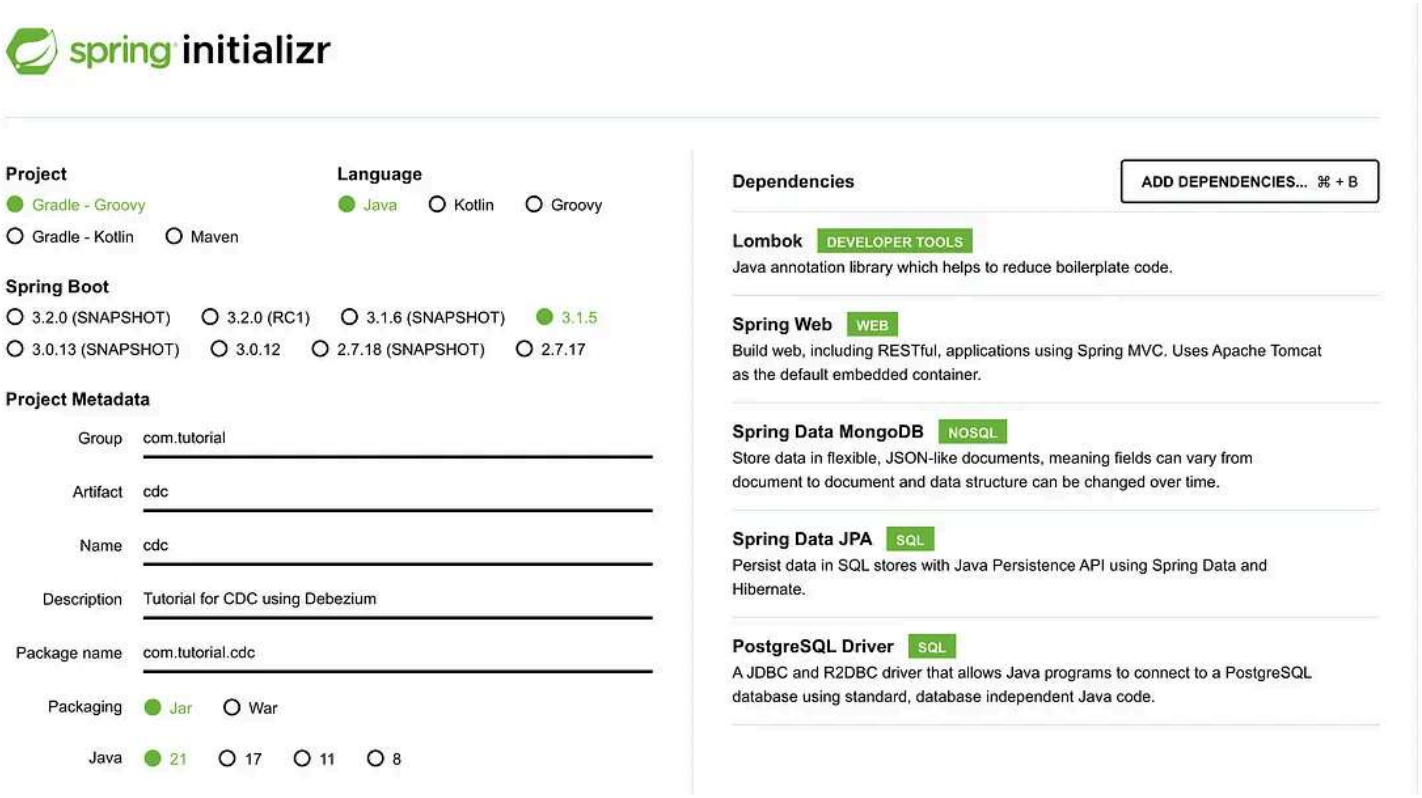


Generating our application

We will generate our project using the ***Spring Initializr***.

Theses are the dependencies that we need to add from the ***Initializr*** :

- Lombok
- Spring web
- Spring Data MongoDB
- Spring Data JPA
- PostgreSQL Driver



Generating the project on Spring Initializr

After generating the project, add it to your favorite IDE 😎.

Now we need to add Debezium's dependencies:

```
//Debezium dependencies
implementation 'io.debezium:debezium-api:2.1.2.Final'
implementation 'io.debezium:debezium-embedded:2.1.2.Final'

//Debezium connector for MongoDB
implementation 'io.debezium:debezium-connector-mongodb:2.1.2.Final'

//Debezium offset storage
implementation 'io.debezium:debezium-storage-jdbc:2.3.0.Final'
```

So our dependencies will look like this :

```
dependencies {
 implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
 implementation 'org.springframework.boot:spring-boot-starter-data-mongodb'
 implementation 'org.springframework.boot:spring-boot-starter-web'
```

```
 compileOnly 'org.projectlombok:lombok'
 annotationProcessor 'org.projectlombok:lombok'
 testImplementation 'org.springframework.boot:spring-boot-starter-test'
 implementation 'io.debezium:debezium-api:2.1.2.Final'
 implementation 'io.debezium:debezium-embedded:2.1.2.Final'
 implementation 'io.debezium:debezium-connector-mongodb:2.1.2.Final'
 implementation 'io.debezium:debezium-storage-jdbc:2.3.0.Final'
 runtimeOnly 'org.postgresql:postgresql'
}
```

Now let's create the **DebeziumConnectorConfig** class :

```
@Configuration
public class DebeziumConnectorConfig {

  @Value("${spring.data.mongodb.uri}")
  private String mongoDbUri;

  @Value("${spring.data.mongodb.username:}")
  private String mongoDbUsername;

  @Value("${spring.data.mongodb.password:}")
  private String mongoDbPassword;

  @Value("${spring.datasource.url}")
  private String postgresUrl;

  @Value("${spring.datasource.username}")
  private String postgresUsername;

  @Value("${spring.datasource.password}")
  private String postgresPassword;

  @Value("${database.include.list}")
  private String databaseIncludeList;

  @Value("${collection.include.list}")
  private String collectionIncludeList;

  @Value("${spring.profiles.active}")
  private String activeProfile;

  @Bean
  public io.debezium.config.Configuration mongodbConnector() {

    Map<String, String> configMap = new HashMap<>();

    //This sets the name of the Debezium connector instance. It's used for logg
```

```
        configMap.put("name", "cdc-mongodb");
        //This specifies the Java class for the connector. Debezium uses this to cr
        configMap.put("connector.class", "io.debezium.connector.mongodb.MongoDbConn
        //This sets the Java class that Debezium uses to store the progress of the
        // In this case, it's using a JDBC-based store, which means it will store t
        configMap.put("offset.storage", "io.debezium.storage.jdbc.offset.JdbcOffset
        //This is the JDBC URL for the database where Debezium stores the connector
        configMap.put("offset.storage.jdbc.url", postgresUrl);
        configMap.put("offset.storage.jdbc.user", postgresUsername);
        configMap.put("offset.storage.jdbc.password", postgresPassword);
        //This is the MongoDB connection string that Debezium uses to connect to yo
        configMap.put("mongodb.connection.string", mongoDbUri);
        //This prefix is added to all Kafka topics that this connector writes to.
        configMap.put("topic.prefix", "sbd-mongodb-connector");
        //This is a comma-separated list of MongoDB database names that the connect
        configMap.put("database.include.list", databaseIncludeList);
        //This is a comma-separated list of MongoDB collection names that the conne
        configMap.put("collection.include.list", collectionIncludeList);
        //When errors.log.include.messages set to true, then any error messages res
        // are also written to the log.
        configMap.put("errors.log.include.messages", "true");

        //Use MongoDB without credentials and without SSL for local and test profil
        if (!"local".equals(activeProfile) && !"test".equals(activeProfile)) {
          configMap.put("mongodb.user", mongoDbUsername);
          configMap.put("mongodb.password", mongoDbPassword);
          configMap.put("mongodb.ssl.enabled", "true");
        }

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

Then add used properties to the **application.yml** file :

```
spring:
  profiles:
    active: local
  datasource:
    url: jdbc:postgresql://localhost:5432/postgres
    username: postgres
    password: root
  data:
    mongodb:
      uri: "mongodb://localhost:27017/"
      database: test
      password:
      username:
```

```
    jpa:
      hibernate:
        ddl-auto: create-drop
  database:
    include:
      list: test
  collection:
    include:
      list: "test.products"
```

Then the **DebeziumSourceEventListener** class :

First we need to inject these beans to our class :

```
//This will be used to run the engine asynchronously
private final Executor executor;

//DebeziumEngine serves as an easy-to-use wrapper around any Debezium connector
private final DebeziumEngine<RecordChangeEvent<SourceRecord>> debeziumEngine;
```

Then we should add the constructor :

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

        // Create a new DebeziumEngine instance.
        this.debeziumEngine = DebeziumEngine.create(ChangeEventFormat.of(Connec
                        .using(mongodbConnector.asProperties())
                        .notifying(this::handleChangeEvent)
                        .build();
    }
```

Let's explain the **debeziumEngine** instance step by step:

```
DebeziumEngine.create(ChangeEventFormat.of(Connect.class))
```

This line is creating a new instance of the Debezium Engine. The `ChangeEventFormat.of(Connect.class)` part specifies the format of the change events that the engine will produce. In this case, it's using the `Connect` class, which means it will produce events in Kafka Connect's data format.

```
.using(mongodbConnector.asProperties())
```

This line is configuring the engine with the properties of a MongoDB connector. The `mongodbConnector.asProperties()` method should return a `Properties` object that contains all the necessary configuration for connecting to a MongoDB database.

```
.notifying(this::handleChangeEvent)
```

This line is specifying a method that will be called whenever a change event is produced by the connector.

To start the debezium engine we need to pass the **debeziumEngine** to the **executor** :

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

And to stop it we need to close the engine.

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

Now Let's now the `handleChangeEvent` method :

```
    private void handleChangeEvent(RecordChangeEvent<SourceRecord> sourceRecord
        SourceRecord sourceRecord = sourceRecordRecordChangeEvent.record();
        Struct sourceRecordKey = (Struct) sourceRecord.key();
        Struct sourceRecordValue = (Struct) sourceRecord.value();
        if (sourceRecordValue != null) {
            try {

                String operation = HandlerUtils.getOperation(sourceRecordValue)

                String documentId = HandlerUtils.getDocumentId(sourceRecordKey)

                String collection = HandlerUtils.getCollection(sourceRecordValu

                Product product = HandlerUtils.getData(sourceRecordValue);

                log.info("Collection : {} , DocumentId : {} , Operation : {}",

            } catch (IOException e) {
                throw new RuntimeException(e);
            }

        }

    }
```

The `handleChangeEvent` will extract useful data from the event.

Now we need to write **HandlerUtils**

```
    /**
     * Extracts the document ID from the given Struct object.
     *
     * @param key The Struct object containing the document information.
     * @return The extracted document ID, or null if not found.
     */
    public static String getDocumentId(Struct key) {
      String id = key.getString("id");
      Matcher matcher = Pattern.compile("\"\\$oid\":\\s*\"(\\w+)\"").matcher(id);
      return matcher.find() ? matcher.group(1) : null;
    }

    /**
```

```
     * Extracts the collection name from the source record value.
     *
     * @param sourceRecordValue The Struct object representing the source record.
     * @return The name of the collection.
     */
    public static String getCollection(Struct sourceRecordValue) {
      Struct source = (Struct) sourceRecordValue.get("source");
      return source.getString("collection");
    }

    /**
     * Deserializes the 'after' field of the source record value into a Product o
     *
     * @param sourceRecordValue The Struct object representing the source record.
     * @return The deserialized Product object.
     * @throws IOException If there is an error during deserialization.
     */
    public static Product getData(Struct sourceRecordValue) throws IOException {
      var source = sourceRecordValue.get("after").toString();
      ObjectMapper mapper = new ObjectMapper();
      mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
      return mapper.readValue(source, Product.class);
    }

    /**
     * Retrieves the operation type from the source record value.
     *
     * @param sourceRecordValue The Struct object representing the source record.
     * @return The operation type, such as "insert", "update", or "delete".
     */
    public static String getOperation(Struct sourceRecordValue) {
      return sourceRecordValue.getString("op");
    }
  }
```

Now the debezium part is almost ready let's now prepare the product part that will deal with Postgres.

Starting by the `Product` entity :

```
@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Product {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;
    private String name;
    private Float price;
    private String description;
    private String sourceCollection;
    private String mongoId;
}
```

Then the `ProductRepository` :

```
@Repository
public interface ProductRepository extends JpaRepository<Product, String> {
    void removeProductByMongoId(String mongoId);
}
```

Then the `ProductService` :

```
public interface ProductService {
    void handleEvent(String operation, String documentId, String collection, Pr
}
```

The **handleEvent** method will be called by the **handleChangeEvent** method.

```
@Service
@AllArgsConstructor
public class ProductServiceImpl implements ProductService {

    private final ProductRepository productRepository;

    @Override
    @Transactional
    public void handleEvent(String operation, String documentId, String collect

        // Check if the operation is either CREATE or UPDATE
        if (Envelope.Operation.CREATE.code().equals(operation) || Envelope.Oper
```

```
                // Set the MongoDB document ID to the product
                product.setMongoId(documentId);
                // Save the updated product information to the database
                productRepository.save(product);

                // If the operation is DELETE
            } else if (Envelope.Operation.DELETE.code().equals(operation)) {
                // Remove the product from the database using the MongoDB document
                productRepository.removeProductByMongoId(documentId);
            }
        }
    }
```

Now we need to update the **DebeziumSourceEventListener** by injecting the **ProductService :**

```
@Slf4j
@Service
public class DebeziumSourceEventListener {

    //This will be used to run the engine asynchronously
    private final Executor executor;

    //DebeziumEngine serves as an easy-to-use wrapper around any Debezium conne
    private final DebeziumEngine<RecordChangeEvent<SourceRecord>> debeziumEngin

    //Inject product service
    private final ProductService productService;


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

        //Create a new DebeziumEngine instance.
        this.debeziumEngine =
                DebeziumEngine.create(ChangeEventFormat.of(Connect.class))
                        .using(mongodbConnector.asProperties())
                        .notifying(this::handleChangeEvent)
                        .build();

        //Set the product service.
        this.productService = productService;
    }
```

```
        private void handleChangeEvent(RecordChangeEvent<SourceRecord> sourceRecord
            SourceRecord sourceRecord = sourceRecordRecordChangeEvent.record();
            Struct sourceRecordKey = (Struct) sourceRecord.key();
            Struct sourceRecordValue = (Struct) sourceRecord.value();
            if (sourceRecordValue != null) {
                try {

                    String operation = HandlerUtils.getOperation(sourceRecordValue)

                    String documentId = HandlerUtils.getDocumentId(sourceRecordKey)

                    String collection = HandlerUtils.getCollection(sourceRecordValu

                    Product product = HandlerUtils.getData(sourceRecordValue);

                    productService.handleEvent(operation, documentId, collection, p

                    log.info("Collection : {} , DocumentId : {} , Operation : {}",

                } catch (IOException e) {
                    throw new RuntimeException(e);
                }

            }

        }

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

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

That's it, Our application is now ready to run 😎

But before we run it, we need to have a running MongoDB and Postgres.

By the way, we need to have a replica set of MongoDB not a simple instance.

> *Debezium's MongoDB connector tracks a MongoDB replica set or a MongoDB sharded cluster for document changes in databases and collections, recording those changes as*

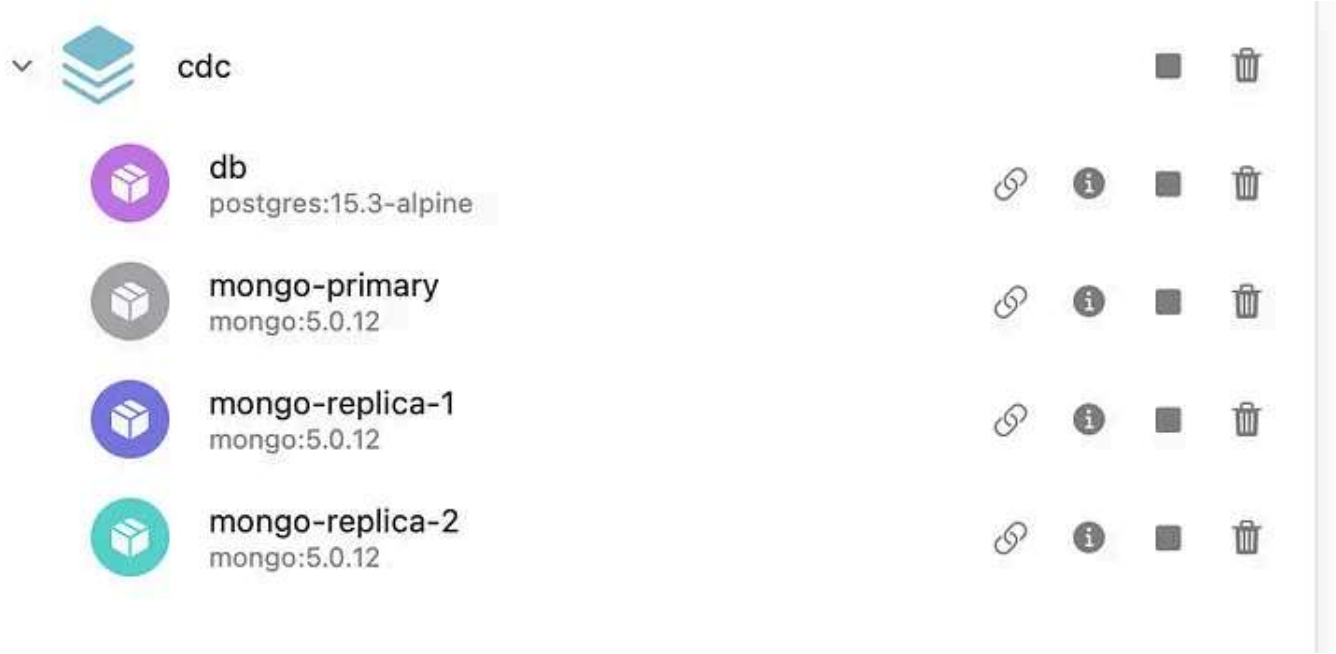
> *events in Kafka topics.*

To keep thing easy, I have prepared a **docker-compose.yml** that will configure a replica set of MongoDB (1 primary and 2 replicas) and a Postgres DB

```
version: '3.8'
networks:
    mongo-net:
        name: mongo-net
services:
    mongo-replica-1:
        hostname: mongo-replica-1
        container_name: mongo-replica-1
        image: mongo:5.0.12
        command: mongod --replSet rs --journal --bind_ip_all
        ports:
            - "27018:27017"
        restart: always
        networks:
            - mongo-net
    mongo-replica-2:
        hostname: mongo-replica-2
        container_name: mongo-replica-2
        image: mongo:5.0.12
        command: mongod --replSet rs --journal --bind_ip_all
        ports:
            - "27019:27017"
        restart: always
        networks:
            - mongo-net

    mongo-primary:
        hostname: mongo-primary
        container_name: mongo-primary
        depends_on:
            - mongo-replica-1
            - mongo-replica-2
        image: mongo:5.0.12
        command: mongod --replSet rs --journal --bind_ip_all
        ports:
            - "27017:27017"
        links:
            - mongo-replica-1
            - mongo-replica-2
        restart: always
        networks:
            - mongo-net
        healthcheck:
            test: test $$(echo "rs.initiate({_id:'rs',members:[{_id:0,host:\"mo
```

```
            interval: 10s
            start_period: 30s

    db:
        image: postgres:15.3-alpine
        container_name: postgres-db
        ports:
            - "5432:5432"
        environment:
            POSTGRES_USER: postgres
            POSTGRES_PASSWORD: root
```

Now we run this using : `docker-compose up -d`



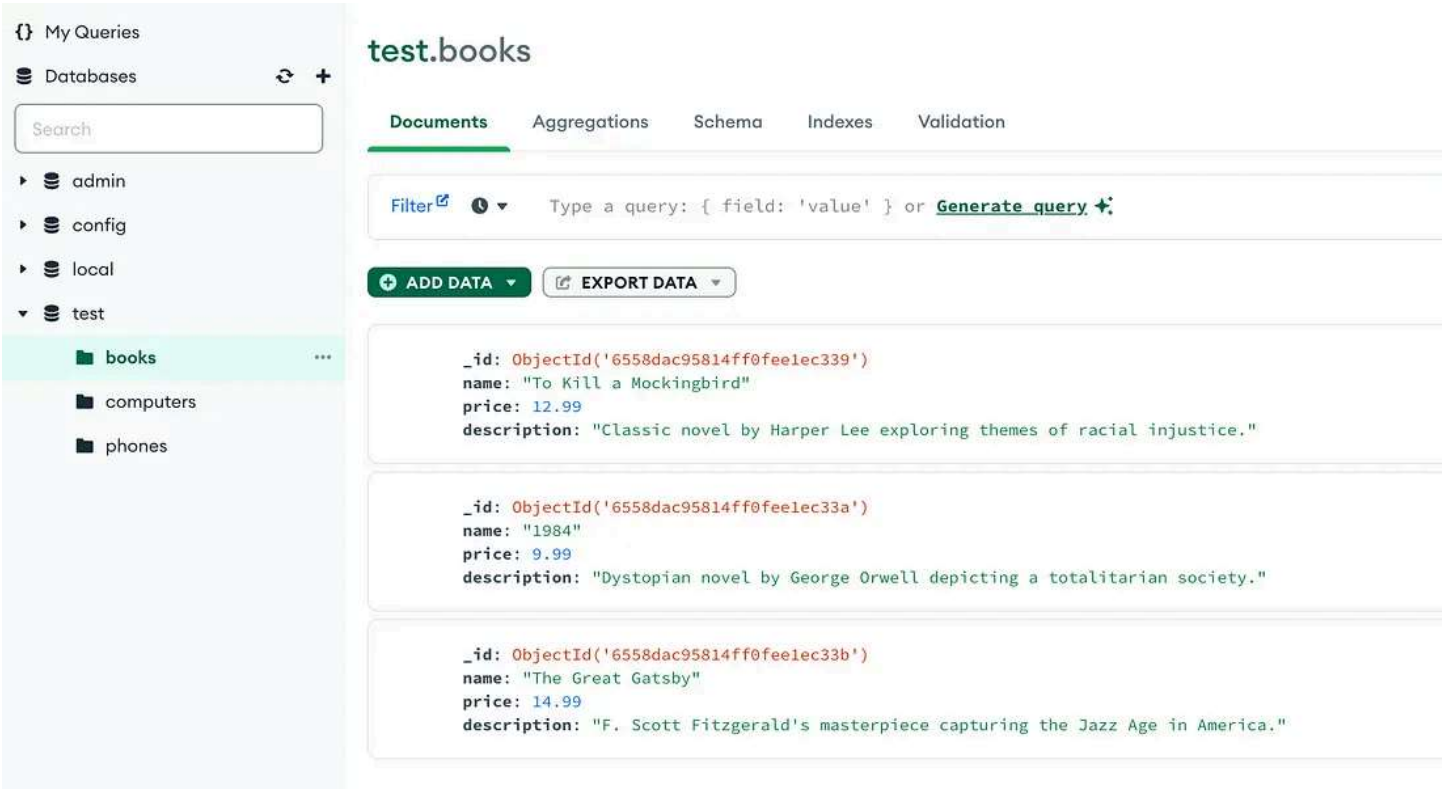
Running containers

Now connect to the primary replica and insert the following data :

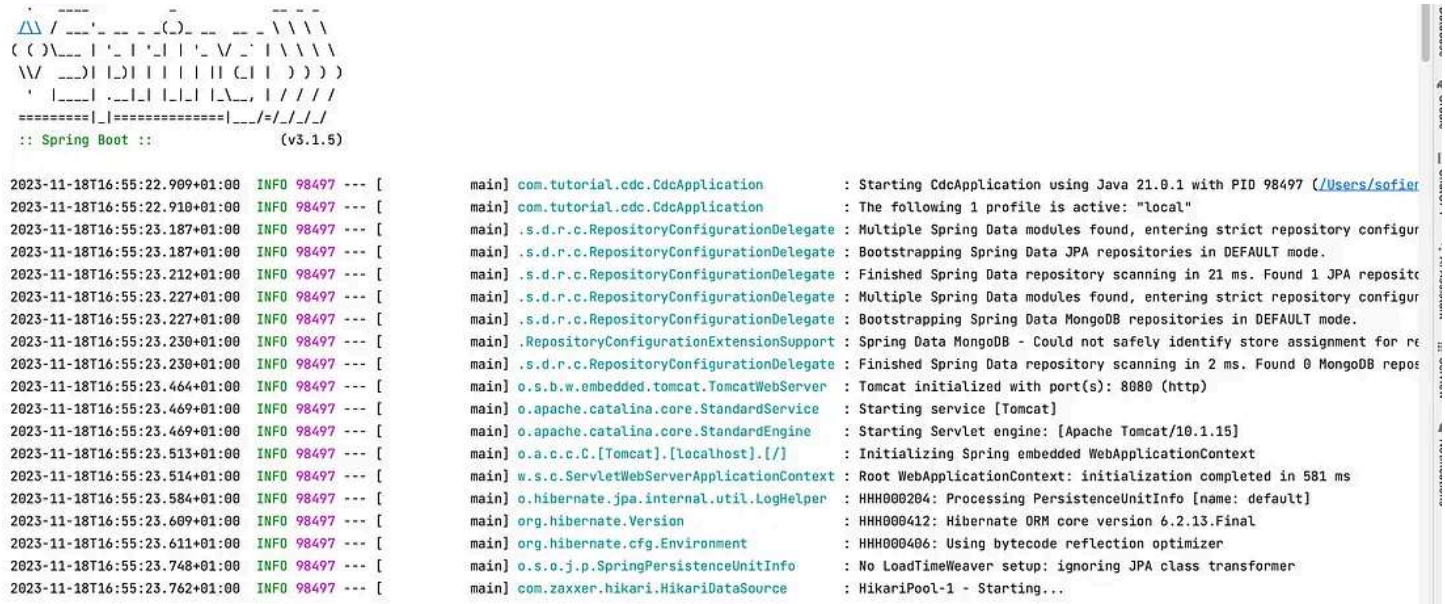
```
db = db.getSiblingDB('test');

db.phones.insertMany([
    {
        "name": "IPhone 15",
        "price": 999,
        "description": "Premium Apple smartphone with powerful features."
    },
    {
        "name": "Samsung Galaxy S23",
        "price": 899,
```

```
            "description": "Premium Android smartphone with powerful features."
        },
        {
            "name": "Google Pixel 6",
            "price": 799,
            "description": "Flagship phone with top-notch camera and performance."
        }
    ])

    db.computers.insertMany([
        {
            "name": "MacBook Pro",
            "price": 1499,
            "description": "High-performance laptop for professionals."
        },
        {
            "name": "Dell XPS 15",
            "price": 1299,
            "description": "Powerful laptop with stunning display and long battery
        },
        {
            "name": "HP Spectre x360",
            "price": 1099,
            "description": "Versatile 2-in-1 laptop with impressive design and perf
        }
    ])


    db.books.insertMany([
        {
            "name": "To Kill a Mockingbird",
            "price": 12.99,
            "description": "Classic novel by Harper Lee exploring themes of racial
        },
        {
            "name": "1984",
            "price": 9.99,
            "description": "Dystopian novel by George Orwell depicting a totalitari
        },
        {
            "name": "The Great Gatsby",
            "price": 14.99,
            "description": "F. Scott Fitzgerald's masterpiece capturing the Jazz Ag
        }
    ])
```

In my case I'm going to use MongoDB Compass



Now let's start our Spring boot application



At the end of log you will see this :

```
Collection : phones , DocumentId : 6558dac95814ff0fee1ec333 , Operation : r
Collection : phones , DocumentId : 6558dac95814ff0fee1ec334 , Operation : r
Collection : phones , DocumentId : 6558dac95814ff0fee1ec335 , Operation : r
Collection : computers , DocumentId : 6558dac95814ff0fee1ec336 , Operation : r
```

```
Collection : computers , DocumentId : 6558dac95814ff0fee1ec337 , Operation : r
Collection : computers , DocumentId : 6558dac95814ff0fee1ec338 , Operation : r
Collection : books , DocumentId : 6558dac95814ff0fee1ec339 , Operation : r
Collection : books , DocumentId : 6558dac95814ff0fee1ec33a , Operation : r
Collection : books , DocumentId : 6558dac95814ff0fee1ec33b , Operation : r
```

this happen when you run debezium for the first time, it read the entire database and then calculate a resume token to detect changes starting from that point.

This resume token will be saved into Postgres db as we use **JdbcOffsetBackingStore**

Debezium supports many other storage tools available **here**

Let's take a look into our Postgres db and see what we have:

PgAdmin UI

	price real	id [PK] bigint	description character varying (255)	mongo_id character varying (255)	name character varying (255)	source_collection character varying (255)
1	999	1	Premium Apple smartphone with powerful features.	6558dac95814ff0fee1ec333	IPhone 15	phones
2	899	2	Premium Android smartphone with powerful features.	6558dac95814ff0fee1ec334	Samsung Galaxy S23	phones
3	799	3	Flagship phone with top-notch camera and performance.	6558dac95814ff0fee1ec335	Google Pixel 6	phones
4	1499	4	High-performance laptop for professionals.	6558dac95814ff0fee1ec336	MacBook Pro	computers
5	1299	5	Powerful laptop with stunning display and long battery life.	6558dac95814ff0fee1ec337	Dell XPS 15	computers
6	1099	6	Versatile 2-in-1 laptop with impressive design and performance.	6558dac95814ff0fee1ec338	HP Spectre x360	computers
7	12.99	7	Classic novel by Harper Lee exploring themes of racial injustice.	6558dac95814ff0fee1ec339	To Kill a Mockingbird	books
8	9.97	8	Dystopian novel by George Orwell depicting a totalitarian society.	6558dac95814ff0fee1ec33a	1984	books
9	14.99	9	F. Scott Fitzgerald's masterpiece capturing the Jazz Age in Ameri…	6558dac95814ff0fee1ec33b	The Great Gatsby	books

We have one table called product that contains all Mongodb collections data.

Now let's update a book document :

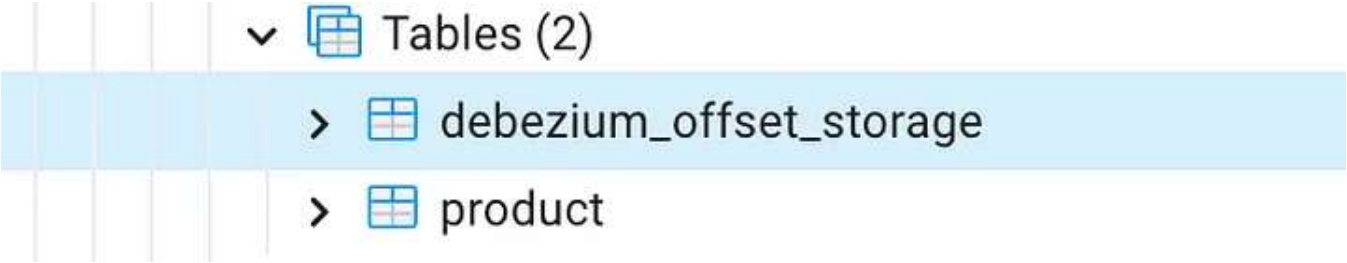
```
db.books.updateOne(
  { "_id" : ObjectId("6558dac95814ff0fee1ec33a") },
  { $set: { "price" : 15 } }
)
```

In the application logs we get this :

```
2023-11-18T17:49:26.559+01:00  INFO 818 --- [pool-2-thread-1] i.d.connector.com
2023-11-18T17:49:26.669+01:00  INFO 818 --- [pool-2-thread-1] c.t.c.l.DebeziumS
```

The first line means that debezium start to use the **resume_token** and the second line tell us that debezium detected and update operation for a specific document.

Let's see postgres tables again :



As you can see, now we have a new table called **debezium_offset_storage** table. This table is used by debezim to store and retreive the resume token.

	id character varying (36)	offset_key character varying (1255)	offset_val character varying (1255)
1	f5f0c49d-e217-4a7a-8003-1ee7abd70817	["cdc-mongodb",{"rs":"rs","server_id":"sbd-mongodb-connector...	{"sec":1700331187,"ord":1,"transaction_id":null,"resume_token":"826558FEB3000000012B022C0100296E5A1004BCC44D8D7CBE4E0

> *A Resume Token is a mechanism used to keep track of the position in the MongoDB oplog (operation log) when streaming changes to a Kafka topic. The oplog is a capped collection in MongoDB that records all write operations.*
>
> *Here's how it works:*
>
> ***Change Tracking:** Debezium monitors the MongoDB oplog for any changes in databases and collections.*
>
> ***Resume Token:** As Debezium processes these changes, it generates a unique identifier known as the Resume Token. This token represents a specific position in the oplog.*
>
> ***Fault Tolerance:** The Resume Token is crucial for fault tolerance. If the Debezium connector or Kafka Connect process is interrupted, it can use the Resume Token to resume*

> *streaming from the last processed position in the oplog, ensuring no data loss.*
>
> *It is essential to carefully select the Debezium storage to avoid losing the resume token.*

After the update operation we made, the data inside postgres must be updated :

	id [PK] bigint	description character varying (255)	mongo_id character varying (255)	name character varying (255)	price real	source_collection character varying (255)
1	52	Premium Apple smartphone with powerful features.	6558dac95814ff0fee1ec333	IPhone 15	999	phones
2	53	Premium Android smartphone with powerful features.	6558dac95814ff0fee1ec334	Samsung Galaxy S23	899	phones
3	54	Flagship phone with top-notch camera and performance.	6558dac95814ff0fee1ec335	Google Pixel 6	799	phones
4	55	High-performance laptop for professionals.	6558dac95814ff0fee1ec336	MacBook Pro	1499	computers
5	56	Powerful laptop with stunning display and long battery life.	6558dac95814ff0fee1ec337	Dell XPS 15	1299	computers
6	57	Versatile 2-in-1 laptop with impressive design and performance.	6558dac95814ff0fee1ec338	HP Spectre x360	1099	computers
7	58	Classic novel by Harper Lee exploring themes of racial injustice.	6558dac95814ff0fee1ec339	To Kill a Mockingbird	12.99	books
8	59	Dystopian novel by George Orwell depicting a totalitarian society.	6558dac95814ff0fee1ec33a	1984	15	books
9	60	F. Scott Fitzgerald's masterpiece capturing the Jazz Age in Ameri…	6558dac95814ff0fee1ec33b	The Great Gatsby	14.99	books

Let’s add a new document inside the phones collection :

```
db.phones.insertOne({
  "name": "OnePlus 10 Pro",
  "price": 800,
  "description": "Flagship OnePlus smartphone with advanced features and Hassel
})
```

```
> db.phones.insertOne({
    "name": "OnePlus 10 Pro",
    "price": 800,
    "description": "Flagship OnePlus smartphone with advanced features and Hasselblad camera technology."
  })
< {
    acknowledged: true,
    insertedId: ObjectId("655903225814ff0fee1ec33c")
  }
```

Now if we look to the application logs we will see a create operation

```
Collection : phones , DocumentId : 655903225814ff0fee1ec33c , Operation : c
```

and new row appears on products table

	id [PK] bigint	description character varying (255)	mongo_id character varying (255)	name character varying (255)	price real	source_collection character varying (255)
1	52	Premium Apple smartphone with powerful features.	6558dac95814ff0fee1ec333	IPhone 15	999	phones
2	53	Premium Android smartphone with powerful features.	6558dac95814ff0fee1ec334	Samsung Galaxy S23	899	phones
3	54	Flagship phone with top-notch camera and performance.	6558dac95814ff0fee1ec335	Google Pixel 6	799	phones
4	55	High-performance laptop for professionals.	6558dac95814ff0fee1ec336	MacBook Pro	1499	computers
5	56	Powerful laptop with stunning display and long battery life.	6558dac95814ff0fee1ec337	Dell XPS 15	1299	computers
6	57	Versatile 2-in-1 laptop with impressive design and performance.	6558dac95814ff0fee1ec338	HP Spectre x360	1099	computers
7	58	Classic novel by Harper Lee exploring themes of racial injustice.	6558dac95814ff0fee1ec339	To Kill a Mockingbird	12.99	books
8	59	Dystopian novel by George Orwell depicting a totalitarian society.	6558dac95814ff0fee1ec33a	1984	15	books
9	60	F. Scott Fitzgerald's masterpiece capturing the Jazz Age in America.	6558dac95814ff0fee1ec33b	The Great Gatsby	14.99	books
10	61	Flagship OnePlus smartphone with advanced features and Hasselblad camera technolo…	655903225814ff0fee1ec33c	OnePlus 10 Pro	800	phones

Now let's delete this phone from the phones collection:

```
db.phones.deleteOne({ "_id": ObjectId("655903225814ff0fee1ec33c") })
```

```
> db.phones.deleteOne({ "_id": ObjectId("655903225814ff0fee1ec33c") })
< {
    acknowledged: true,
    deletedCount: 1
  }
```

Now if we look to the application logs we will see the delete operation

```
Collection : phones , DocumentId : 655903225814ff0fee1ec33c , Operation : d
```

and now the row disappears from products table

	id [PK] bigint	description character varying (255)	mongo_id character varying (255)	name character varying (255)	price real	source_collection character varying (255)
1	52	Premium Apple smartphone with powerful features.	6558dac95814ff0fee1ec333	IPhone 15	999	phones
2	53	Premium Android smartphone with powerful features.	6558dac95814ff0fee1ec334	Samsung Galaxy S23	899	phones
3	54	Flagship phone with top-notch camera and performance.	6558dac95814ff0fee1ec335	Google Pixel 6	799	phones
4	55	High-performance laptop for professionals.	6558dac95814ff0fee1ec336	MacBook Pro	1499	computers
5	56	Powerful laptop with stunning display and long battery life.	6558dac95814ff0fee1ec337	Dell XPS 15	1299	computers
6	57	Versatile 2-in-1 laptop with impressive design and performance.	6558dac95814ff0fee1ec338	HP Spectre x360	1099	computers
7	58	Classic novel by Harper Lee exploring themes of racial injustice.	6558dac95814ff0fee1ec339	To Kill a Mockingbird	12.99	books
8	59	Dystopian novel by George Orwell depicting a totalitarian society.	6558dac95814ff0fee1ec33a	1984	15	books
9	60	F. Scott Fitzgerald's masterpiece capturing the Jazz Age in Ameri…	6558dac95814ff0fee1ec33b	The Great Gatsby	14.99	books

Conclusion

In this article, we've explored the concept of Change Data Capture (CDC) and learned how to integrate Debezium with Spring Boot and MongoDB to enable real-time data streaming. By following the steps outlined in this guide, you can easily set up CDC in your applications, allowing you to react to data changes as they happen and build responsive, data-driven solutions.

All the project sources are available on my GitHub page :

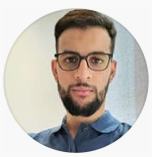
GitHub - sofieneBK/Hands-on-debezium-cdc-with-springboot: Code for Change Data Capture Made Easy...

Code for Change Data Capture Made Easy: Debezium Integration with Spring Boot Tutorial - GitHub ...

github.com

Thank you for reading! 🙏 🙏

Java Spring Boot Debezium Docker Mongodb



Follow

Written by Sofiene Ben Khemis

413 Followers

Software Engineer & AWS Community Builder I'm not perfect. still running after my dreams. going to the moon 🌚

More from Sofiene Ben Khemis

Sofiene Ben Khemis in Javarevisited

Hands-on Hexagonal Architecture With Spring Boot

Hexagonal architecture, or port and adapter architecture, is an architectural pattern used in software design. It aims to create systems...

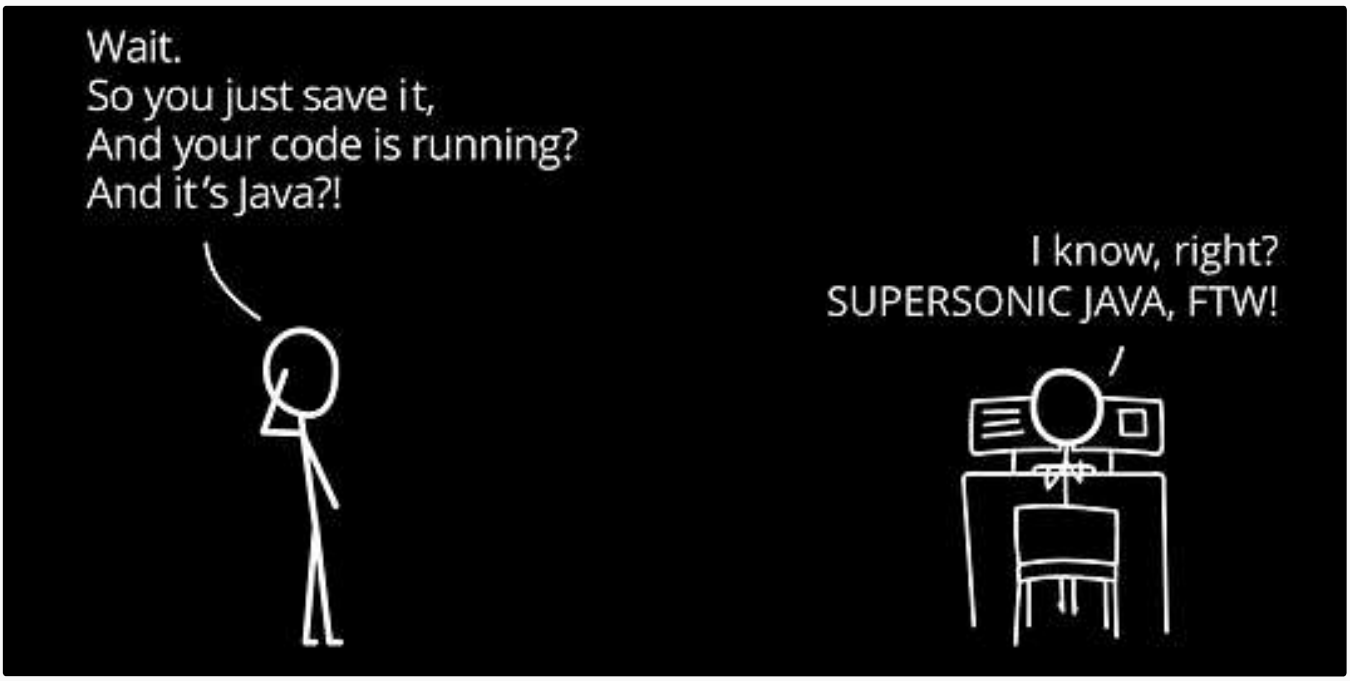
Mar 29, 2021 532 8

Sofiene Ben Khemis

Optimistic locking in java

When it comes to enterprise applications, it is essential to properly manage concurrent access to a database.

Feb 10, 2020 16



Sofiene Ben Khemis in The Startup

Is Quarkus the future of Java?

A Kubernetes Native Java stack tailored for OpenJDK HotSpot and GraalVM, crafted from the best of breed Java libraries and standards.

May 1, 2020 533 3

Sofiene Ben Khemis in Javarevisited

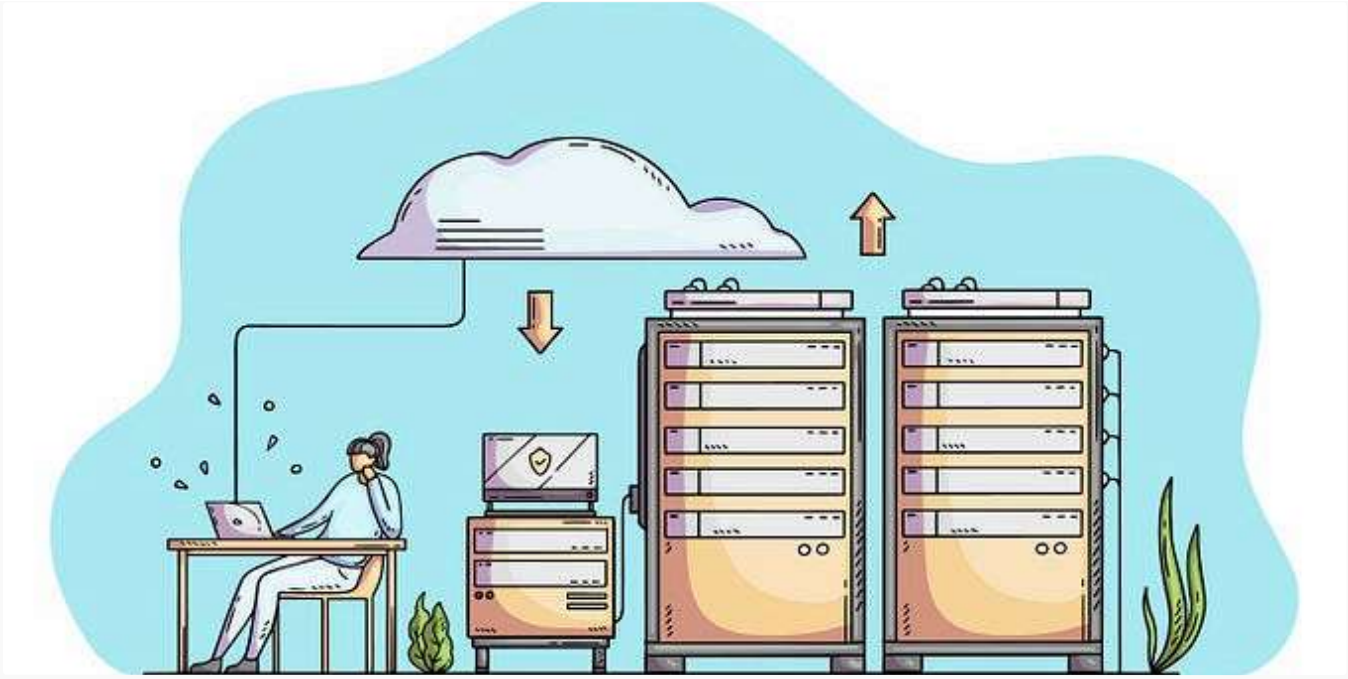
Hands-on Consul With Spring boot

Learn how to easily integrate Consul with Spring Boot for dynamic service discovery and configuration management

Jan 23, 2023 53 3

See all from Sofiene Ben Khemis

Recommended from Medium



Nithidol Vacharotayan

Create a PostgreSQL database server with docker.

PostgreSQL is an open-source database. Developers can create their database servers for any purpose. PostgreSQL provides an image for...

May 17 4

Rabinarayan Patra

Integrating MySQL with Spring Boot: A Comprehensive Guide: All in ONE

Dive into integrating MySQL with Spring Boot: from setup, entity mapping, to advanced JPA configurations unlock the full potential of data.

Feb 24 15

Lists

Coding & Development

11 stories · 754 saves

General Coding Knowledge

20 stories · 1502 saves

data science and AI

40 stories · 223 saves

Natural Language Processing

1662 stories · 1234 saves

Vasko Jovanoski

Spring Boot and Testcontainers—A love test story

Hi everyone!

Feb 25 12

Kareem Mohllal

Change Data Capture (CDC) with Debezium, Kinesis, and EventBridge

Capture data changes in source databases and deliver them to downstream systems in real-time using an event-driven approach.

Apr 14 26

NGU in Level Up Coding

Spring Boot 3.3: Hidden Gems Features Must-Try

Boost your development and debugging efficiency

Aug 16 118 1

Catherine Edelveis

Using Liquibase with Spring Boot Tutorial

Any application relying on a database will benefit from a database management tool, and for enterprise projects, it is a must. Two most...

Apr 8 41 2

See more recommendations