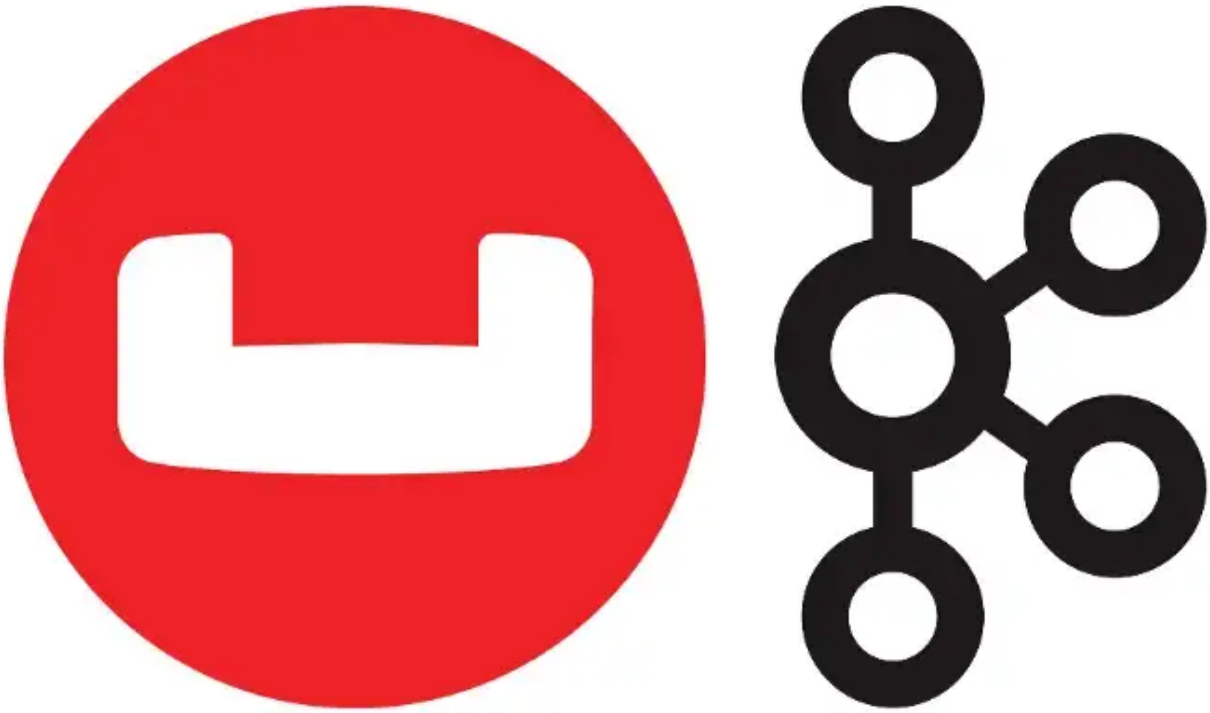
Published in Trendyol Tech

Mehmet Sezer Follow

Nov 22, 2022 · 8 min read · Listen



How did we implement Outbox Pattern by using Couchbase and Kafka?

Here are some steps we've experienced while implementing the Outbox Pattern in our projects.

What is Outbox Pattern?

Outbox Pattern **guarantees** that our **domain events** will be sent. It only ensures that domain events will be sent but **does not guarantee when it will happen.**

Why did we need Outbox Pattern?

As the Product International domain, we are sending domain events to Kafka about the products that can be sold in different countries. In the past, **the applications** were sending domain events to Kafka directly. This would cause us to be unable to provide service **in case of a possible problem in Kafka.** We've never encountered such a situation, but that doesn't mean we never will.

What were the requirements for our new domain events sender system?

- Should guarantee that it will send domain events to Kafka, if not immediately, but definitely.
- Even if there is a problem in Kafka, our services should continue to work without any error, and when the problem in Kafka is fixed, the events that were not sent should be sent successfully without breaking their order.
- Kafka domain events **contract(Kafka key, payload, headers) must not change** to ensure other teams are not affected by this development.
- Should be suitable for the technology stack we are using(Couchbase and Kafka).
- Must be scalable.

Implementation

Instead of sending Kafka events directly from our application, we added a domainEvents list to our AggregateRoot. If you are not familiar with Domain-Driven Design concepts, you might think that each database entity is an Aggregate Root.

```
abstract class AggregateRoot<A : AggregateRoot<A>>(
    @JsonIgnore val aggregateId: String,
    var createdBy: String,
    var createdDate: Instant
) {
  ...
  // we added new field to store domainEvents
  var domainEvents: MutableList<DomainEvent> = mutableListOf()
}
```

After that, we removed all Kafka producer code from projects.

Couchbase Eventing

Official Couchbase documentation defines the Couchbase Eventing as follows:

> *The Couchbase Eventing Service is a framework to operate on changes to data in real time. Events are changes to data in the Couchbase cluster.*

With Couchbase Eventing, we can write a **JavaScript function** that can be triggered as a result of changes(create, update, delete, expiry, etc.) on the cluster.

We can easily define this function from the interface of our Couchbase cluster.

Couchbase Kafka Connector

Official Couchbase documentation defines the Couchbase Kafka Connector as follows:

> *The Couchbase Kafka connector is a plug-in for the Kafka Connect framework. It provides source and sink components.*

The **source connector** streams documents from Couchbase Database Change Protocol (DCP) and publishes each document to a Kafka topic in near real-time.

The **sink connector** subscribes to Kafka topics and writes the messages to Couchbase.

We will only need to use a source connector for our system to listen to Couchbase mutations and send events to Kafka.

First of all, we saved our domain events to Couchbase over the related documents. What we need to do now is to move these events on the documents to a different Couchbase collection because we will connect the Kafka connector to the collections that store events. We can easily do this using Couchbase Eventing.

```
    if (domainEvents) {
            for (let domainEvent of domainEvents) {
                const domainEventId = create_UUID()
                events[domainEventId] = domainEvent;
                log("event moved into outbox", domainEventId, domainEvent);
            }
        }
    // clear the domainEvents list of the document
    doc.domainEvents = []
    realCollection[meta.id] = doc
}
```

As seen in the example above, we moved the events in the document to another collection named **events**. **realCollection** which is an alias of the current collection, we can define the aliases while creating a new eventing function. Since these events will take up too much space in the system after a while, we need to define Time-To-Live(TTL) and ensure that these documents are deleted after a certain time. We can enter this TTL value while creating the event collection.

Now we have a collection where our events are kept, but we need a Couchbase Source Kafka Connector to send them to Kafka. We deployed a connector by cloning the connector repository. This connector code **does not support custom Key and Headers,** so if we use directly this repository, we won't be able to send key and headers information. **Key** is important for the events sent for the same document to go in order to Kafka. **Headers** contain detailed information about the event, other teams may decide whether to listen to the event or not according to the headers information.

Solution for Providing Key and Headers to Every Domain Event

Since the connector code does not support keys and headers, we had to update the connector code ourselves. We updated our DomainEvent class to store headers information per domain event.

```
        fun getHeaders(): Map<String, String?> = constructMessageHeaders(getType()
    }
```

constructMessageHeaders is just a function to create the message header by using MDC and some static metadata such as API name. As you can see “x-kafka-headers" is a special variable to store headers information. We also needed to update the connector code. In the connector code, we read this “x-kafka-headers” value and started sending it as a Kafka header.

In order to solve the key problem, each domain event has to have an id field already, we updated the connector code so that it sends this **id field as the key of the event,** to provide order.

By configuring the connector, we enabled it to listen to the collections where we keep the events and send messages to the Kafka topics we specified.

An example domain event:

```
{
  "id": "abcabcabc",
  "payload": {
    "name": "Test"
  },
```



```
    "User-Agent": "unknown-user-agent",
    "X-PublishedAt": "2022-11-17T13:36:31.218813239Z",
    "X-AgentName": "test-api",
    "X-Type": "international.event.test"
  }
}
```

Unfortunately not everything was that easy.

Our first problem was how could we send deletion domain events for deleted documents. Couchbase Eventing **onUpdate** function only listen create and update events. The **onDelete** function needs to be used to listen for deletion operations but **onDelete function can not access the document.** This causes us to be unable to access the “domainEvents” field on the document so we won’t be able to send deleted domain events to Kafka.

```
function OnUpdate(doc, meta) {
    ...
}
// We can't access the doc
function OnDelete(meta) {
   ....
}
```


Solution for Delete Operations

To solve this problem, we first created a new collection for deleted documents.

And we revised the delete function in our code. This code is not production code, there are also system codes that handle operational errors that may occur due to Couchbase, but I removed them for simplicity.

```
override suspend fun remove(aggregate: T): Unit {
      val documentId = getDocumentId(aggregate.aggregateId)
      collection.remove(documentId)
```

```
        deletedCollection.upsert(documentId, aggregate as AggregateRoot<T>)
    }
```

As can be seen in the code above, for each deleted document, we create the deleted document in our newly created collection.

And we wrote a Couchbase Eventing function for this new deleted collection **to be able to send deletion events as we send other events.**

In eventing, we can create some alias to access collections, **realCollection** is the alias of the real collection. **deleted** is the alias of the deleted collection.

Example: If we delete a document from the system, it will be removed from the **realCollection** collection completely but it will create a new document on **deleted** collection.

```
        return uuid;
    }
```

As can be seen in the code above, we listen to this deletion collection and feed the event collection just like we did in creating and updating. The biggest difference is that **we delete the deleted document directly after the process is finished since we do not need it later.**

Everything seems fine now we can send deletion events too.

Let's do a load test.

During a detailed load test, we found that there was an inconsistency in the number of events sent. It was hard to find fault. The problem occurred in the code where we emptied the list after moving the events to the new collection. Since we updated it without using any optimistic lock-like solution, a race condition has occurred.

Solution of Race Condition

```
function OnUpdate(doc, meta) {
    const domainEvents = doc.domainEvents
    if (domainEvents) {
            for (let domainEvent of domainEvents) {
                const domainEventId = create_UUID()
                events[domainEventId] = domainEvent;
                log("event moved into outbox", domainEventId, domainEvent);
            }
        }
    // clear the domainEvents list of the document
    doc.domainEvents = []
    var result = couchbase.replace(realCollection, {"id": meta.id, "cas": meta.
    if (!result.success) {
        log('replace not succeed: result ',result, meta.id);
    }
}
```

As seen in the code above, instead of updating the original document directly, we used the **replace** method offered by Couchbase. This function takes CAS(Concurrent Document Mutations) variable as a parameter.

What is CAS in Couchbase?

Official Couchbase documentation defines the CAS as follows.

> *You can use the CAS value to control how concurrent document modifications are handled. It helps avoid and control potential race conditions in which some mutations may be inadvertently lost or overridden by mutations made by other clients.*

In summary, Couchbase offers a CAS value for documents, and by using this CAS value as a version, we can prevent race conditions without locking the document directly.

With this CAS value we used while updating the document, we prevented race conditions.

After solving all these problems, we have taken this development live and now we are able to send domain events to Kafka in production without any problem. From now on, our applications will not fail in a possible problem that may occur in Kafka.

Since our connectors keep their states, after the Kafka is up, the connectors will send the events that it could not send in the order.

All feedbacks are welcome, thank you.

References

Microservices Pattern: Transactional outbox

Application events A service command typically needs to update the database and send messages/events. For example, a...

microservices.io

Eventing Service: Fundamentals

The Couchbase Eventing Service is a framework to operate on changes to data in real time. Events are changes to data in...

docs.couchbase.com

bliki: DDD_Aggregate

Aggregate is a pattern in Domain-Driven Design. A DDD aggregate is a cluster of domain objects that can be treated as a...

martinfowler.com

Introduction

The Couchbase Kafka connector is a plug-in for the Kafka Connect framework. It provides source and sink components. The...

docs.couchbase.com

Concurrent Document Mutations

You can use the CAS value to control how concurrent document modifications are handled. It helps avoid and control...

docs.couchbase.com

Microservices Couchbase Kafka Outbox Pattern

Thanks to Sena Erdogan


Download on the
App Store


GET IT ON
Google Play