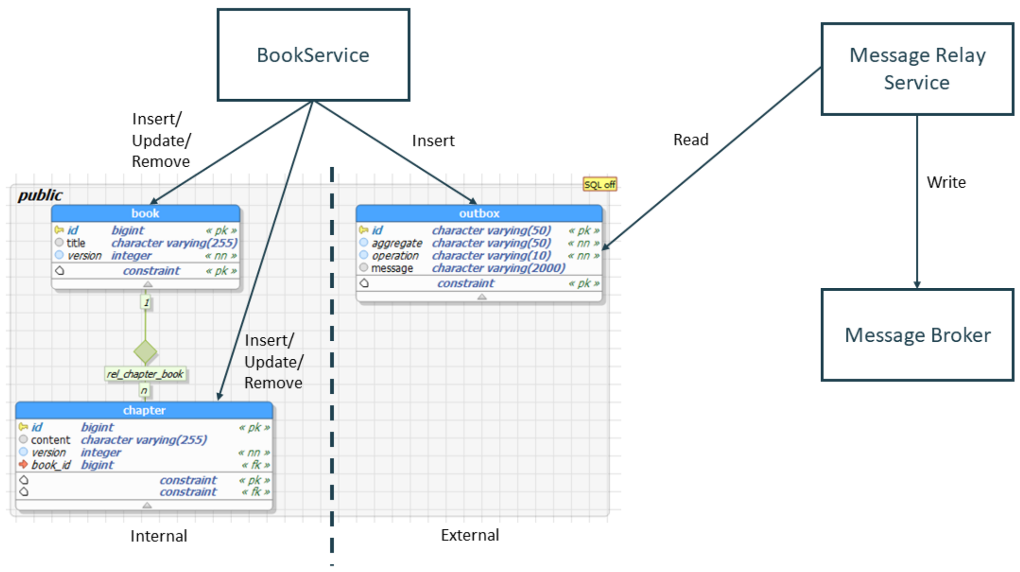
Implementing the Outbox Pattern with Hibernate

When you apply the Outbox pattern, you split the communication between your microservice and the message broker into multiple parts by providing an outbox table.



In the next step, you need an additional service that gets the messages from your outbox table and sends them to your message broker. This message relay service is the topic of another tutorial and I only want to mention your 2 main implementation options here:

1. You can use a tool like Debezium to monitor the logs of your database and let it send a message for each new record in the outbox table to your message broker. This approach is called Change Data Capture (CDC).
2. You can implement a service that polls the outbox table and sends a new message to your message broker whenever it finds a new record.

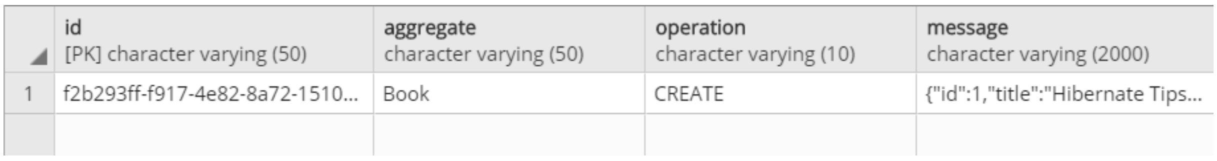
I prefer option 1, but both of them are a valid solution to connect your outbox table with your message broker.

The Structure of the Outbox Table

The outbox table is an external API of your service and you should treat it in the same way as any other externally available API. That means:

- You need to keep the structure of the table and the contained messages stable.
- You need to be able to change your microservice internally.
- You should try to not leak any internal details of your service.

To achieve all of this, most teams use a table that's similar to the following one. They use a UUID as the primary key, a JSON column that contains the payload of the message and a few additional columns to describe the message.



	id [PK] character varying (50)	aggregate character varying (50)	operation character varying (10)	message character varying (2000)
1	f2b293ff-f917-4e82-8a72-1510...	Book	CREATE	{"id":1,"title":"Hibernate Tips...

The message is often times based on the aggregate for which the message was created. So, if your microservice manages books, the aggregate root might be the book itself, which includes a list of chapters.

Whenever a book gets created or changed or when a chapter gets added, a new message for the book gets added to the outbox table.

The payload of the message can be a JSON representation of the full aggregate, e.g. a book with all chapters, or a message-specific subset of the aggregate. I prefer to include the full aggregate in the message, but that's totally up to you.

Write the Outbox Record Programmatically

Writing the record programmatically is relatively simple. You need to implement a method that transforms your aggregate into its JSON representation and inserts it, together with a few additional information, into the outbox table. You can then call this method from your business logic when you perform any changes on your aggregate.

But how do you write the record? Should use an entity or an SQL INSERT statement?

In general, I recommend using a simple SQL INSERT statement which you execute as a native query. Using an entity doesn't provide you any benefits because it's a one-time write operation. You will not read, update or remove the database record. You will also not map any managed association to it. So, there is no need to map the outbox table to an entity class or to manage the lifecycle of an entity object.

Here is an example of a *writeMessage* method which writes a message for the previously described book aggregate. Please pay special attention to the creation of the JSON document. As described earlier, I prefer to store the complete aggregate which includes the book and the list of chapters.

```
public class OutboxUtil {


    private static ObjectMapper mapper = new ObjectMapper();


    public static final void writeBookToOutbox(EntityManager em, Book book,
Operation op) throws JsonProcessingException {


        ObjectNode json = mapper.createObjectNode()
            .put("id", book.getId())
            .put("title", book.getTitle());


        ArrayNode items = json.putArray("chapters");


        for (Chapter chapter : book.getChapters()) {
            items.add(mapper.createObjectNode()
                        .put("id", chapter.getId())
                        .put("content", chapter.getContent())
            );
        }
        Query q = em.createNativeQuery("INSERT INTO Outbox (id, operation,
aggregate, message) VALUES (:id, :operation, :aggregate, :message)");
        q.setParameter("id", UUID.randomUUID());
        q.setParameter("operation", op.toString());
        q.setParameter("aggregate", "Book");
        q.setParameter("message", mapper.writeValueAsString(json));
        q.executeUpdate();
    }
}
```

In your business code, you can now call this method with an instance of the *Book* entity and an enum value that represents the kind of operation (create, update or remove) performed on the aggregate.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

Book b = new Book();
b.setTitle("Hibernate Tips - More than 70 solutions to common Hibernate
problems");
em.persist(b);

Chapter c1 = new Chapter();
c1.setContent("How to map natural IDs");
c1.setBook(b);
b.getChapters().add(c1);
em.persist(c1);

OutboxUtil.writeBookToOutbox(em, b, Operation.CREATE);

em.getTransaction().commit();
em.close();
```