**Name**: Jacques Desire Graham

**Role**: Java developer

**Summary of my experience as microservices developer**

I worked on a Java 8 / Spring 5 / Spring boot 2 project to improve the automation process for an Ecommerce platform. Parcel sorting, packing and delivery,

We built a lot of microservices. e.g : Parcel loading, Parcel sorting, Parcel packing, Parcel unloading, and Parcel delivery.

We use Dockers containers, Kubernetes and Kafka for microservices communications (messaging, event, logging). We use Jenkins for Continuous Integration.

We use Junit, Cucumber and Mockito for tests.

**Microservices architecture /Design:**

The goal is to builda set of loosely coupled, collaborating, scalable and reusable services.  If there is an issue in one microservice, only that microservice will be affected. The other microservices will continue to work as expected. So when we want to update the application, it will be sufficient to directly target the microservice responsible for the concerned feature.

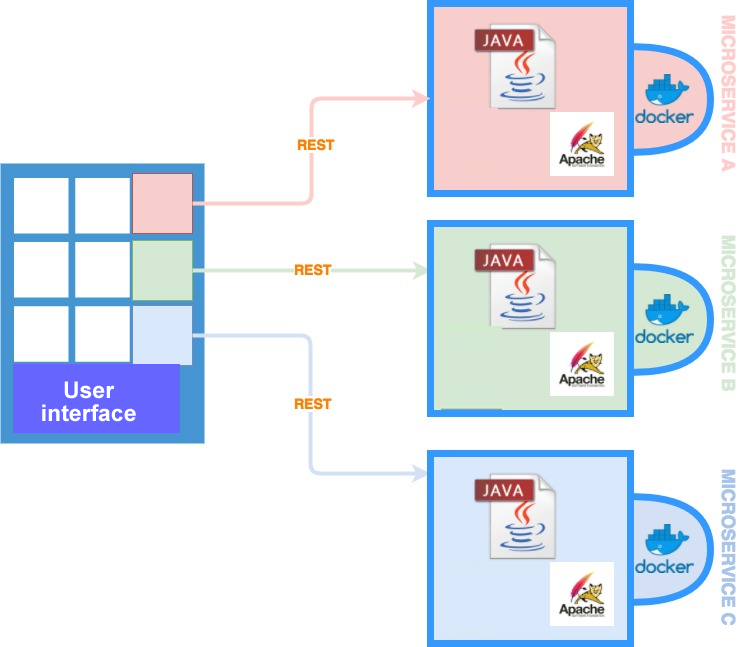
We use a combination of **Event driven, Domain Driven** and the **Single Responsibility Principle (SRP)** to divide the software into microservices.

We use a communication backbone (Kafka) for messaging between the microservices.

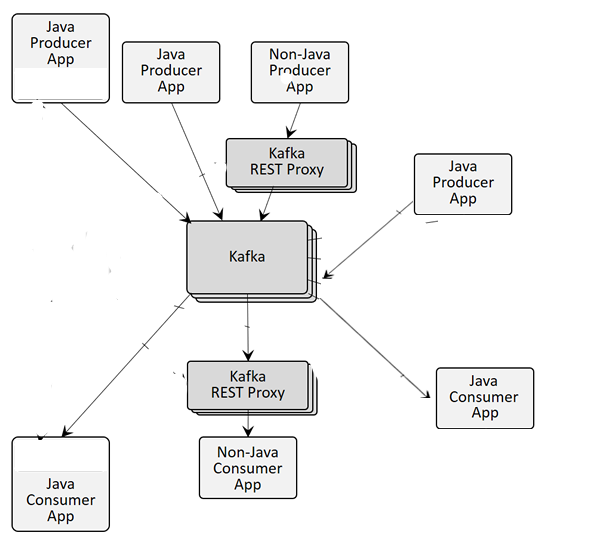
Some of our microservices make call of Restful service.

Our system is like the below image.

**Scenario from User to different Microservices**

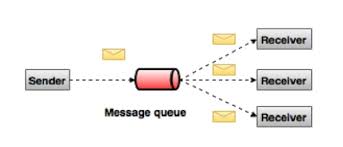


**Scenario from Microservice to Kafka**

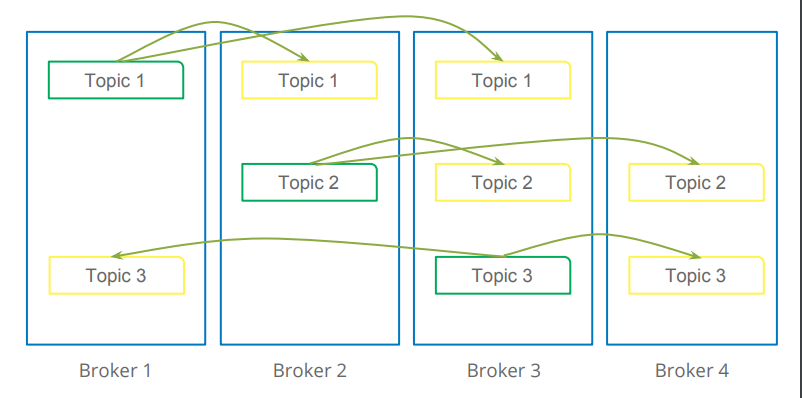


**KAFKA**

Kafka allows the consumption of messages in real time and in batch mode. We use publish-subscribe (pub-sub patterns) messaging system. In the pub-sub system, messages are persisted in a topic. Each topic corresponds to a category of data. Systems that publish data in Kafka topics are Producers. The systems that read the data of the topics are consumers. Producers send messages consisting of an optional key and a value. Several Producers can send data to the same topic. Consumers can subscribe to one or more topic and consume all the messages in that topic. Kafka records are immutable.



Kafka offer distributed, partitioned, replicated, parallelism and fault tolerance. Kafka is used to store messaging, events, metrics, logs. This data is replicated to multiple brokers.



* Each **stream partition** is a totally ordered sequence of data records and maps to a Kafka **topic partition**.
* A **data record** in the stream maps to a Kafka **message** from that topic.
* The **keys** of data records determine the partitioning of data in both Kafka and Kafka Streams, i.e., how data is routed to specific partitions within topics.

Kafka uses ZooKeeper to coordinate (topic partition leadership election) the brokers/cluster topology. ZooKeeper is a file system for configuration information. ZooKeeper sends changes of the topology to Kafka, so each node in the cluster knows when a new broker joins, a Broker dies, a topic was removed or a topic was added.

A Kafka cluster is made up of multiple Kafka Brokers. Each Kafka Broker has a unique ID (number). Kafka Brokers contain topic log partitions.

**REST:**

We implemented REST in order to meet the **"Loose Coupling", “Stateless Communication”, “Explicit caching”** and **"Open/Close principle"** requirements. We use **Spring HATEOAS** to build an API that describes in its responses how it can be used, by providing URLs to other allowed actions.

HATEOAS reduce the need for configuring URL endpoints and other brittle configurations.

The HTTP verbs we use:

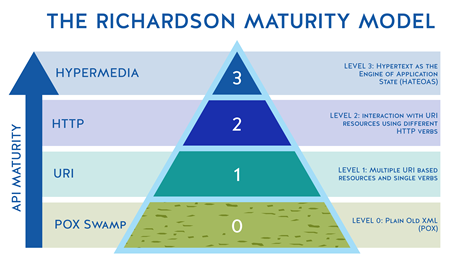
**GET:** to find (get) an existing resource. **POST**: to create a new resource.

**PUT:** to update an existing resource. **PATCH**: to partially update a resource.

**DELETE:** to delete a resource.

**REST versioning:** For the webservice versioning we use the “**Request Parameter versioning**” but we know there are other options like “URI Versioning”, “Headers versioning” and Media type versioning (Accept Header).

By this way we meet the level 3 of REST maturity model.

****