MINI PROJET

Développement à base de Composants

CQRS MICROSERVICES

Rihab Cheberli
Anas Hamrouni

MICROSERVICES

Microservices is an **architectural style** that structures an application as a collection of services that are :

- Highly ***maintainable*** and ***testable.***
- ***Loosely*** coupled.
- ***Independently*** deployable.
- ***Organized*** around business capabilities.
- Owned by a ***small team.***

The microservice architecture enables the **rapid**, **frequent** and **reliable** delivery of large, *complex applications*. It also enables an **organization** to evolve its technology stack.

COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS)

CQRS is one of the important **design pattern** when *querying* between microservices. We can use it in order to avoid complex queries by **separating** read and update operations.

In *monolithic* applications, we have 1 database that should respond both ***query*** and ***update*** operations. It is both working for complex join queries, and also perform CRUD operations. If the application goes more complex this query and crud operations will be also is going to be *un-manageable* situation.

CONTEXT

Applying the **Microservices architecture pattern** and the Database per service pattern. As a result, it is no longer straightforward to implement queries that join data from **multiple services**. Also, if you have applied the **Event sourcing** pattern then the data is no longer ***easily*** queried.

PROBLEM

How to implement a query that retrieves data from **multiple** services in a microservice architecture?

SOLUTION

Defining a **view database**, which is a read-only replica that is designed to support that query. The application keeps the ***replica*** up to data by subscribing to **Domain events** published by the service that own the data.

BENEFITS

- Supports ***multiple*** denormalized views that are ***scalable*** and ***performant.***
- Improved ***separation of concerns***.
- Necessary in an event sourced architecture

THE AIM OF THIS PROJECT :

Creating a solution that implements **CQRS** using the following:

- A **Read** service that makes calls to ElasticSearch.
- A **Write** service that makes calls to MongoDB
- Assert **coherence** between the contents of MongoDB & ElasticSearch

We decided to implement this solution using these technologies :

Spring Cloud provides tools for developers to quickly build some of the common patterns in *distributed* systems (***configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state***

It focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others.

- Distributed/versioned configuration.
- Service registration and discovery.
- Routing
- Service-to-service calls

And many other features.

Elasticsearch is a distributed, open-source search and analytics engine built on Apache Lucene and developed in Java.

It allows you to **store**, **search**, and **analyze** huge volumes of data quickly and in near real-time and give back answers in milliseconds.

It's able to achieve ***fast search responses*** because instead of searching the text directly, it searches an index. It uses a structure based on documents instead of tables and schemas and comes with extensive REST APIs for storing and searching the data.

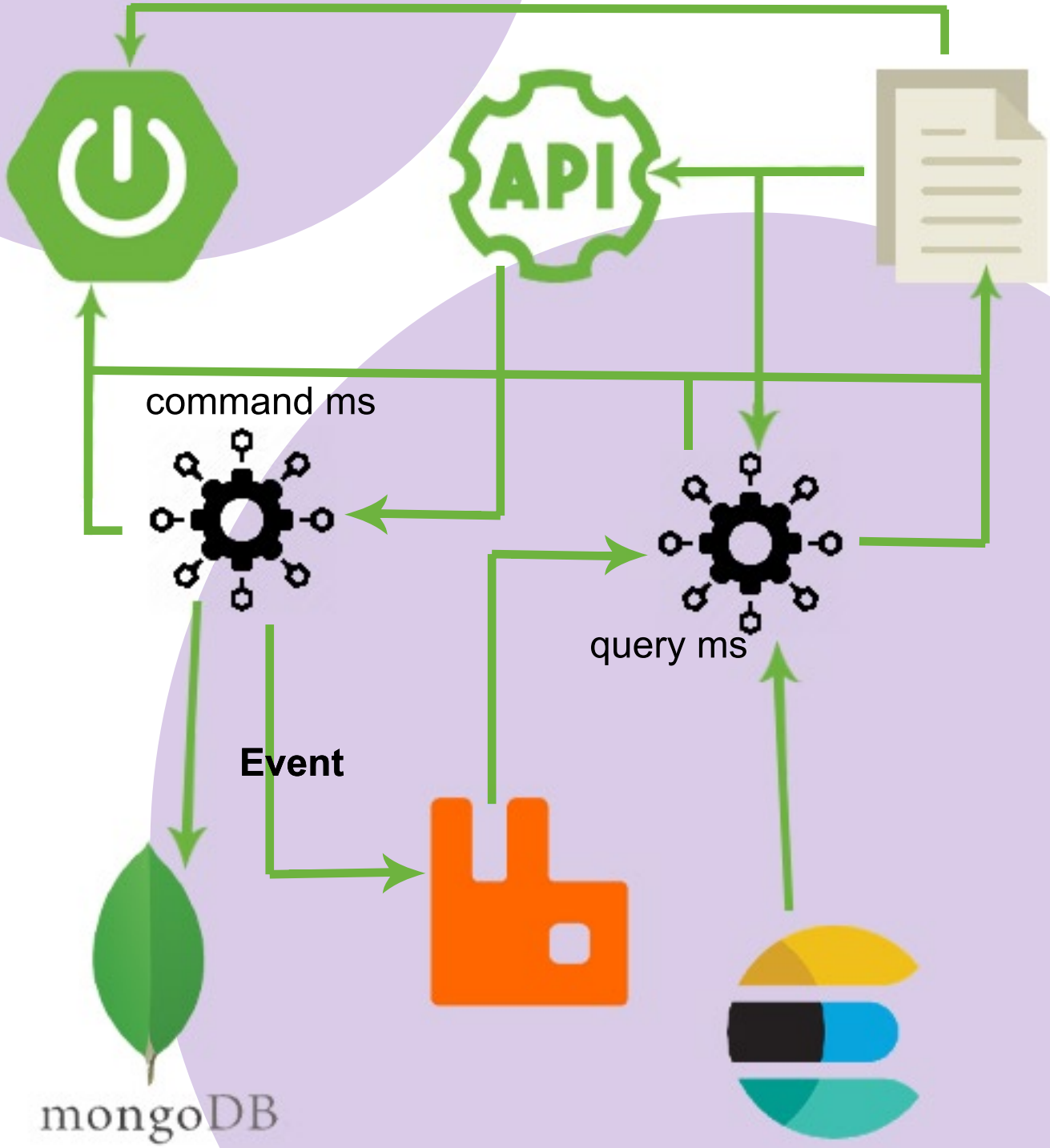
At its core, you can think of Elasticsearch as a server that can process JSON requests and give you back JSON data.

At a high level, **MongoDB** enables developers that use data to build **easily**, adapt **quickly**, and scale **reliably**. It gives :

- Flexible document schemas.
- Code-native data access.
- Change-friendly design.
- Powerful querying and analytics.
- Easy horizontal scale-out.

RabbitMQ is a **message-queueing** software also known as a message broker or queue manager. ***Queues*** are defined, to which applications connect in order to **transfer** a message or messages.

PROJECT ARCHITECTURE



We break down this architecture like it follows :

Service Registry : The services will register themselves directly when it's up.
Config Service : Provides the services config.
API Gateway: Handles incoming requests, redirects them to the corresponding registered internal services.
Command and Query Services : MicroServices implementing CQRS.
RabbitMQ : The message broker that synchronizes between both databases by providing events.
MongoDB : Command dababase.
ElasticSearch : Query database.