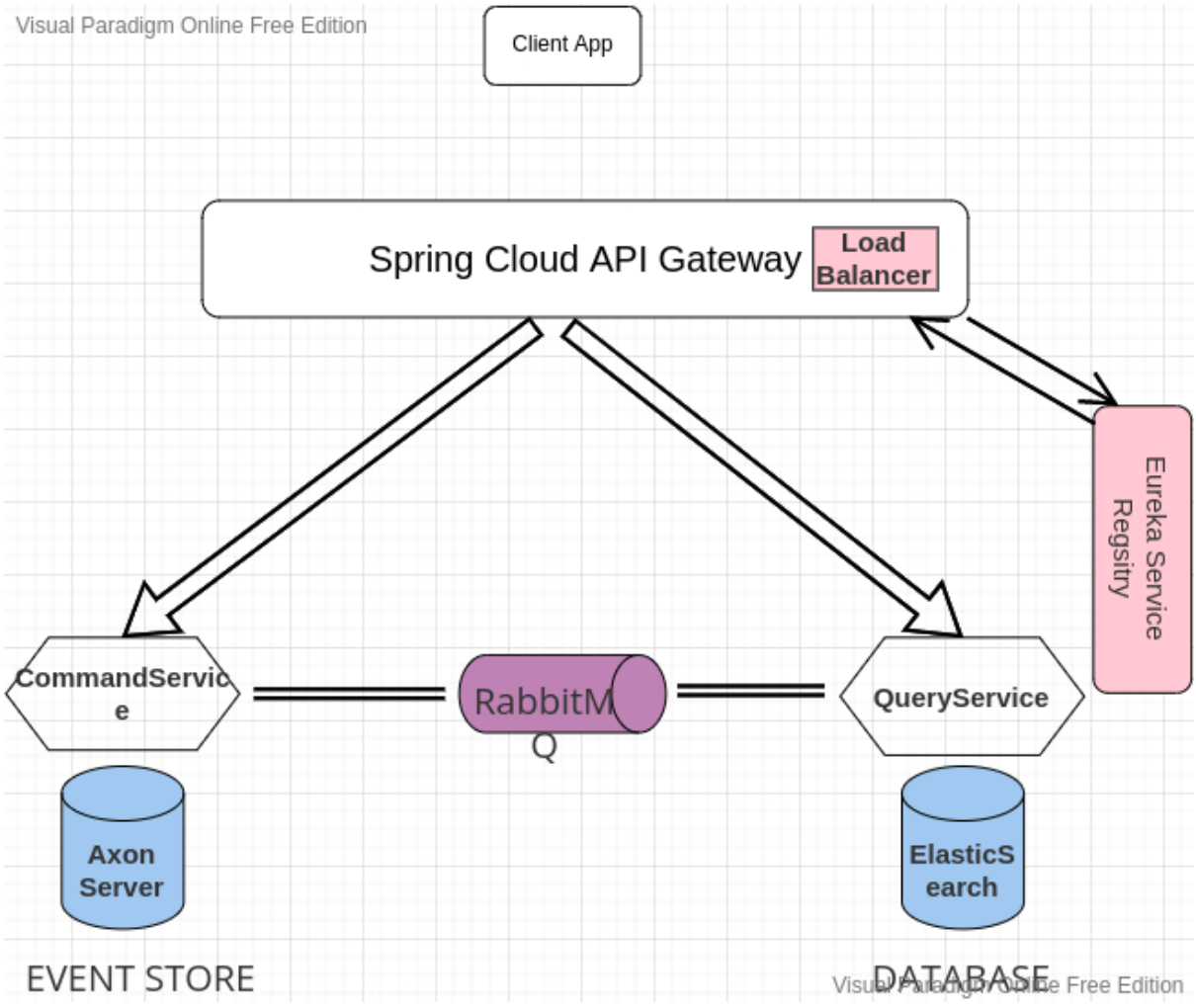
Walkthrough SpringBoot Microservices Project

The project architecture follows multiple patterns such as:

- CQRS
- Event-Sourcing
- Event-Driven Design

Overall Architecture:



After viewing the overall architecture of the project we will be walking through each piece of it separately

.

1. API Gateway
2. Eureka Service Registry
3. Command Service
4. Query Service

API Gateway:

- It is responsible to route and map incoming HTTP requests from the client to the specific destination micro-service automatically. We just need to do the right configuration in application.properties file :

```
spring.cloud.gateway.discovery.locator.enabled=true
```

- It has a built-in Load Balancer
- It is a Eureka Client (which means it should register within the discovery service, more on that later)

Concerned dependency :

```
<dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-gateway</artifactId>
  </dependency>
```

Eureka Server:

- Eureka Server is an application that holds information about all client-service applications. Every Microservice will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server.
- The code for main Spring Boot application class file is as shown below :

```
package com.tutorialspoint.eurekaserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class EurekaserverApplication {
   public static void main(String[] args) {
      SpringApplication.run(EurekaserverApplication.class, args);
   }
}
```

- The code for Maven user dependency is shown below

```
<dependency>
<groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

- By default, the Eureka Server registers itself into the discovery. We should add the below given configuration into application.properties file.

```
eureka.client.registerWithEureka = false
eureka.client.fetchRegistry = false
server.port = 8761
```

Command Service:

workflow of the command Service :

1. Receives commands to update the database (via rest API)
2. create an event out of it

3. publish the event to the event store
4. publish the event to rabbitmq broker

The command service is built following **Event-Driven Architecture.**

- Event-driven architecture is a software architecture and model for application design. With an event-driven system, the capture, communication, processing, and persistence of events are the core structure of the solution. This differs from a traditional request-driven model.

> ***Why did I choose Event-driven Architecture?***
>
> - An event-driven architecture is loosely coupled because event producers don’t know which event consumers are listening for an event, and the event doesn’t know what the consequences are of its occurrence.
> - Event-driven architecture can help organizations achieve a flexible system that can adapt to changes and make decisions in real time.
> - This design, along with **Axon Framework** will make the handling of events very easy and is designed to enable event sourcing and to persist events automatically in the event store Axon Server.

> ***Why Axon Framework?***
>
> - Based on architectural principles, such as Domain-Driven Design (DDD) and Command-Query Responsibility

> Separation (CQRS), Axon Framework provides the building blocks that CQRS requires and helps to create scalable and extensible applications while maintaining application consistency in distributed systems.

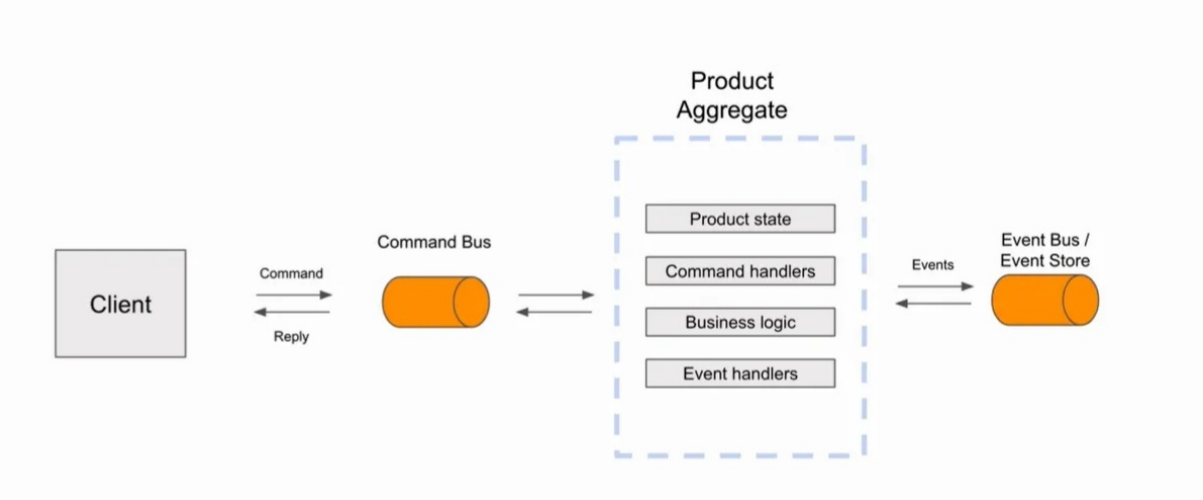
Proven technology for DDD / event-sourcing / CQRS systems	Focus on business logic	Integrates with Spring Boot and other frameworks	Open Source

> ***Why Axon Server?***
>
> - Zero-configuration message router and event store for Axon Framework
> - Axon Server makes it significantly easier for the user to setup and maintain the environment.

Reduce configuration management	Real-time insight	Scalable event sourcing	Real-time insight

Command Service Architecture:

Here is the final architecture of the CommandService:



Query Service:

workflow of the Query Service :

1. Receives read query via rest api
2. Query the database and return the result
3. All along, listen to published events in rabbitmq
4. Handle received events and updates the elasticsearch database

> **Why Elasticsearch?**
>
> Search has become a central idea in many fields with ever-increasing data. As most applications become data-intensive, it is important to search through a large volume of data with speed and flexibility. ElasticSearch offers both.

RabbitMQ:

RabbitMQ message broker is used to assure an asynchronous communication between the Command microservice and the Query microservice.

Run using ready-to-use docker image:

```
docker run --rm -it -p 15672:15672 -p 5672:5672 rabbitmq:3-management
```

Configuration file for RabbitMQ:

```
@Configuration
public class MQConfig {


    public static final String QUEUE = "event_queue";
```

```
    @Bean
    public Queue queue(){
        return new Queue(QUEUE);
    }

    @Bean
    public Jackson2JsonMessageConverter messageConverter(){
        return new Jackson2JsonMessageConverter();
    }

    @Bean
    public  AmqpTemplate template(ConnectionFactory connectionFactory){
        RabbitTemplate template = new RabbitTemplate(connectionFactory);
        template.setMessageConverter(messageConverter());
        return template;
    }
}
```