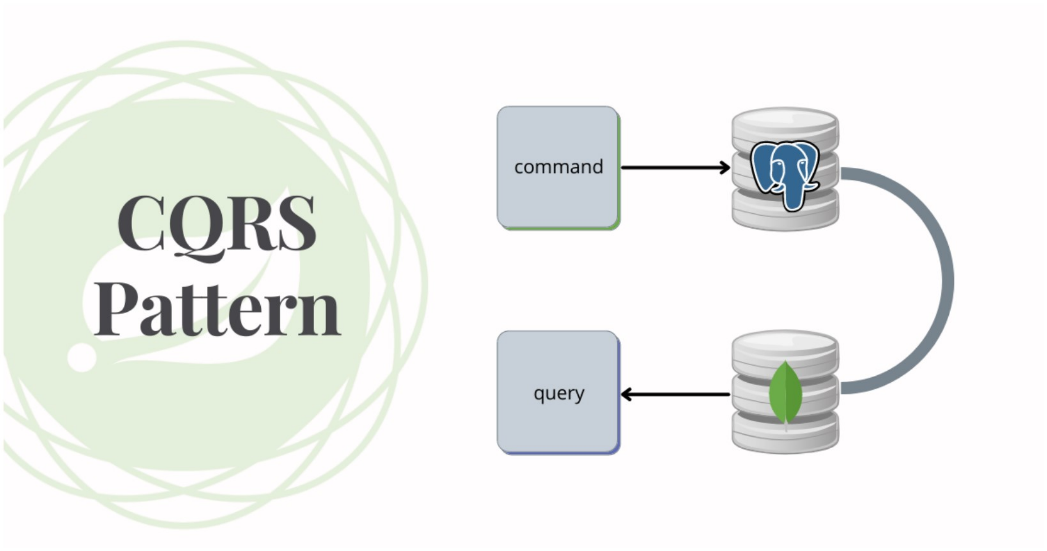
Vinsguru



CQRS Pattern With Spring Boot

16 Comments / Architectural Design Pattern, Architecture, Articles, Best Practices, Design Pattern, Framework, Java, Kafka, Kubernetes Design Pattern, MicroService, Spring, Spring Boot, Spring WebFlux / By vIns / December 14, 2020

Overview:

In this tutorial, I would like to demo **CQRS Pattern with Spring Boot** which is one of the **Microservice Design Patterns** to independently scale **read** and **write** workloads of an application & have well optimized data schema.

CQRS Pattern:

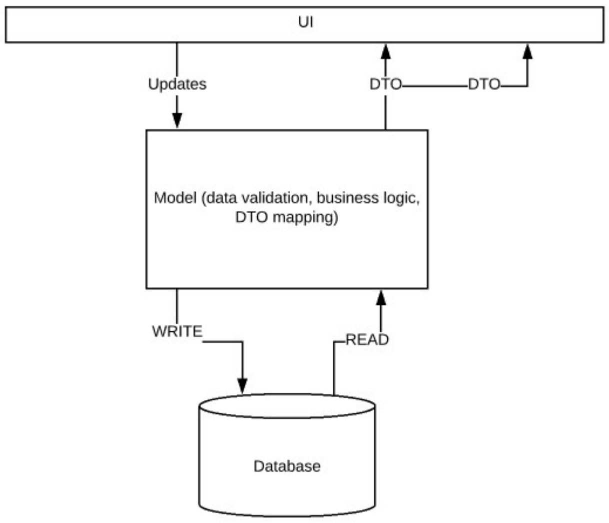
1. Read vs Write Models:

Most of the applications are CRUD in nature. When we design these applications, we create **entity** classes and corresponding **repository** classes for **CRUD** operations. We use the same model classes for all the CRUD operations. However these applications might have completely different **READ** and **WRITE** requirements!

For example: Let's consider an application in which we have 3 tables as shown here.

- **user**
- **product**
- **purchase_order**

All these tables have been normalized. Creating a new user or a product or an order go to the appropriate tables quickly and directly. But if we consider READ requirements, we would not simply want all the users, products, just orders. Instead we would be interested in knowing all the orders details of an user, state wise total sale, state and product wise sale etc. A lot of aggregate information which involves multiple tables join. All these join READ operations might require corresponding DTO mapping as well.



More normalization we do, it is more easier to write and but it is more difficult to read which in turn affects the overall read performance.

Even though we say WRITE is easy, before we insert the record into the database, we might do business validation. All these logic might be present in the model class. So we might end up creating a very complex model which is trying to support both READ and WRITE.

> *An application can have completely different READ and WRITE requirements. So a model created for WRITE might not work for READ. To solve this problem, we can have separate models for READ and WRITE.*

2. Read vs Write Traffic:

Most of the web based applications are read heavy. Let's take Facebook/Twitter for example. Whether we post any new updates or not, we all check these applications very often in a day. Of course, we keep getting updates in those applications which are due to some inserts in the DB. But it is a lot of READ than WRITE. Also, consider a flight-booking application. Probably less than 5% of the users might book a ticket while majority of the application users would keep searching for the best flight meeting their needs.

> *Applications have more requests for READ operations compared to WRITE operations. To solve this problem, We can even have separate Microservices for READ and WRITE. So that they can be scaled in/out independently depends on their needs.*

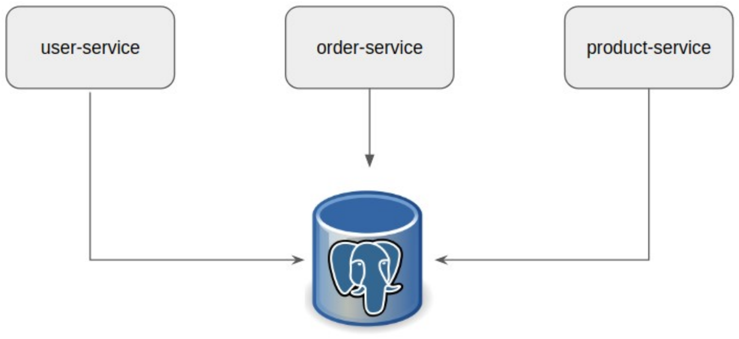
This is what **CQRS Pattern** is about which stands for **Command Query Responsibility Segregation Pattern**.

- **Command:** modifies the data and does not return anything **(WRITE)**
- **Query:** does not modify but returns data **(READ)**

That is separating **Command** (write) and **Query** (read) models of an application to scale read and write operations of an application independently. We can solve above 2 problems using **CQRS Pattern.** Lets see how it can be implemented.

Sample Application:

Lets consider a simple application in which we have 3 services as shown below. (Ideally all these services should have different databases. Here just for this article, I am using same db). We are interested only in the order-service related functionalities for now for this article.



We have 3 tables for this application as shown here.

- user
- product
- purchase_order

```
CREATE TABLE users(
   id serial PRIMARY KEY,
   firstname VARCHAR (50),
   lastname VARCHAR (50),
   state VARCHAR(10)
);

CREATE TABLE product(
   id serial PRIMARY KEY,
   description VARCHAR (500),
   price numeric (10,2) NOT NULL
);

CREATE TABLE purchase_order(
    id serial PRIMARY KEY,
    user_id integer references users (id),
    product_id integer references product (id),
    order_date date
);
```

Lets assume that we have an interface for the order service for the read and write operations as shown below.

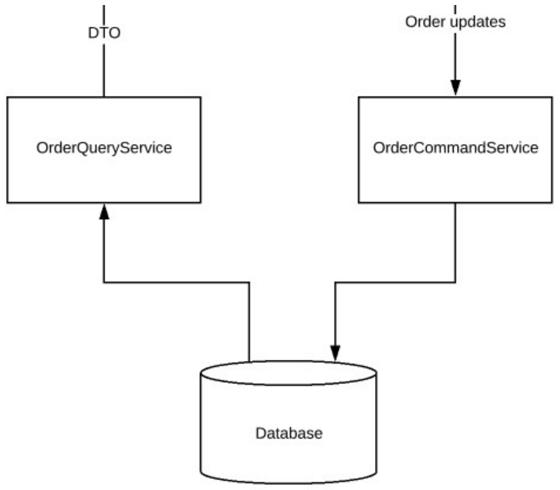
```
public interface OrderService {
    void placeOrder(int userIndex, int productIndex);
    void cancelOrder(long orderId);
    List<PurchaseOrderSummaryDto> getSaleSummaryGroupByState();
    PurchaseOrderSummaryDto getSaleSummaryByState(String state);
    double getTotalSale();
}
```

- It has multiple responsibilities like placing the order, cancelling the order and querying the table which produces different types of results.
- Cancelling the order might involve additional business logic like order date should be within 30 days and partial refund calculation etc.

CQRS Pattern – Read & Write Interfaces:

Instead of having 1 single interface which is responsible for all the READ and WRITE operations, Lets split them into 2 different interfaces as shown here.

- **Query Service** handles all the READ requirements
- **Command Service** handles all other requirements which modifies the data



Order Query Service:

```
public interface OrderQueryService {
    List<PurchaseOrderSummaryDto> getSaleSummaryGroupByState();
    PurchaseOrderSummaryDto getSaleSummaryByState(String state);
    double getTotalSale();
}
```

Order Command Service:

```
public interface OrderCommandService {
    void createOrder(int userIndex, int productIndex);
    void cancelOrder(long orderId);
}
```

CQRS Pattern – Read & Write Implementations:

Once we have separated command and query interfaces, Lets implement the operations.

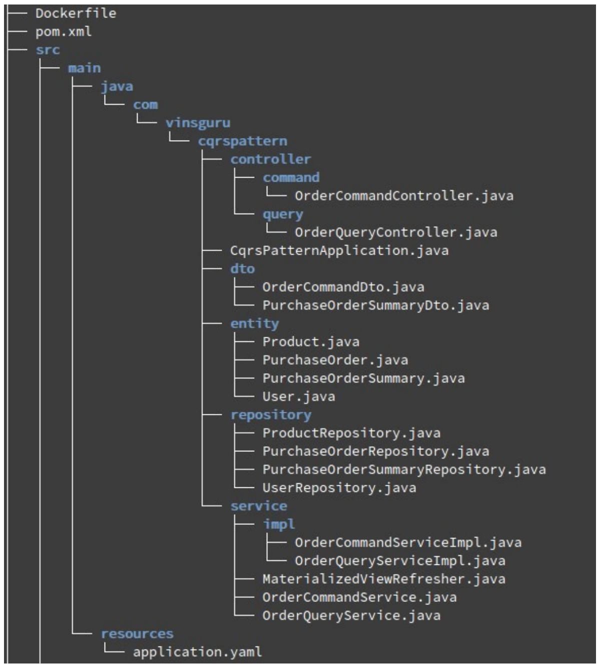
- **Order Query Service**

```
@Service
public class OrderQueryServiceImpl implements OrderQueryService {

    @Autowired
    private PurchaseOrderSummaryRepository purchaseOrderSummaryRepository;

    @Override
    public List<PurchaseOrderSummaryDto> getSaleSummaryGroupByState() {
        return this.purchaseOrderSummaryRepository.findAll()
                .stream()
                .map(this::entityToDto)
                .collect(Collectors.toList());
    }

    @Override
    public PurchaseOrderSummaryDto getSaleSummaryByState(String state) {
        return this.purchaseOrderSummaryRepository.findByState(state)
                        .map(this::entityToDto)
                        .orElseGet(() -> new PurchaseOrderSummaryDto(state, 0));
    }

    @Override
    public double getTotalSale() {
        return this.purchaseOrderSummaryRepository.findAll()
                        .stream()
                        .mapToDouble(PurchaseOrderSummary::getTotalSale)
                        .sum();
    }
    private PurchaseOrderSummaryDto entityToDto(PurchaseOrderSummary purchaseOrderSummary){
        PurchaseOrderSummaryDto dto = new PurchaseOrderSummaryDto();
        dto.setState(purchaseOrderSummary.getState());
        dto.setTotalSale(purchaseOrderSummary.getTotalSale());
        return dto;
    }
}
```

- Order Command Service
  - This is simple insert into the **purchase_order** table with its business logic which modifies the data – it does not return anything.

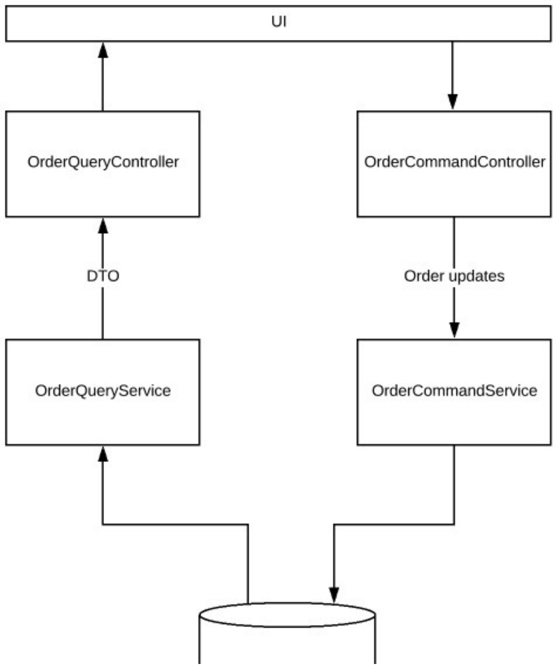
Command vs Query – Controllers:

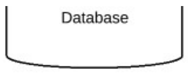
Overall the project structure looks like this. If you take a closer look at this, You could notice that I have 2 different controllers for Order Service.



```
├── Dockerfile
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com
│   │   │       └── vinsguru
│   │   │           └── cqrspattern
│   │   │               ├── controller
│   │   │               │   ├── command
│   │   │               │   │   └── OrderCommandController.java
│   │   │               │   └── query
│   │   │               │       └── OrderQueryController.java
│   │   │               ├── CqrsPatternApplication.java
│   │   │               ├── dto
│   │   │               │   ├── OrderCommandDto.java
│   │   │               │   └── PurchaseOrderSummaryDto.java
│   │   │               ├── entity
│   │   │               │   ├── Product.java
│   │   │               │   ├── PurchaseOrder.java
│   │   │               │   ├── PurchaseOrderSummary.java
│   │   │               │   └── User.java
│   │   │               ├── repository
│   │   │               │   ├── ProductRepository.java
│   │   │               │   ├── PurchaseOrderRepository.java
│   │   │               │   ├── PurchaseOrderSummaryRepository.java
│   │   │               │   └── UserRepository.java
│   │   │               └── service
│   │   │                   ├── impl
│   │   │                   │   ├── OrderCommandServiceImpl.java
│   │   │                   │   └── OrderQueryServiceImpl.java
│   │   │                   ├── MaterializedViewRefresher.java
│   │   │                   ├── OrderCommandService.java
│   │   │                   └── OrderQueryService.java
│   │   └── resources
│   │       └── application.yaml
```

We have dedicated controllers for **Query** (READ) and **Command** (WRITE).





We can even control if the app should work in READ mode or WRITE mode based on a property value.

Order Query Controller:

My order query controller is implemented as shown here which has only GET requests. It does not do anything which modifies the data.

```
@RestController
@RequestMapping("po")
@ConditionalOnProperty(name = "app.write.enabled", havingValue = "false")
public class OrderQueryController {

    @Autowired
    private OrderQueryService orderQueryService;

    @GetMapping("/summary")
    public List<PurchaseOrderSummaryDto> getSummary(){
        return this.orderQueryService.getSaleSummaryGroupByState();
    }

    @GetMapping("/summary/{state}")
    public PurchaseOrderSummaryDto getStateSummary(@PathVariable String state){
        return this.orderQueryService.getSaleSummaryByState(state);
    }

    @GetMapping("/total-sale")
    public Double getTotalSale(){
        return this.orderQueryService.getTotalSale();
    }

}
```

Order Command Controller:

```
@RestController
@RequestMapping("po")
@ConditionalOnProperty(name = "app.write.enabled", havingValue = "true")
public class OrderCommandController {

    @Autowired
    private OrderCommandService orderCommandService;

    @PostMapping("/sale")
    public void placeOrder(@RequestBody OrderCommandDto dto){
        this.orderCommandService.createOrder(dto.getUserIndex(), dto.getProductIndex());
    }

    @PutMapping("/cancel-order/{orderId}")
    public void cancelOrder(@PathVariable long orderId){
        this.orderCommandService.cancelOrder(orderId);
    }
}
```

App properties:

My application.yaml looks like this to run this application in READ mode.

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/productdb
    username: vinsguru
    password: admin
app:
  write:
    enabled: false
```

Of course, we can change the below property to run the application in WRITE mode.

```
app.write.enabled=true
```

CQRS Pattern – Scaling:

Now we have successfully split the READ and WRITE models. Now we need the ability to scale our system independently. Lets see how we can achieve that.

Creating Spring Bean At Run Time:

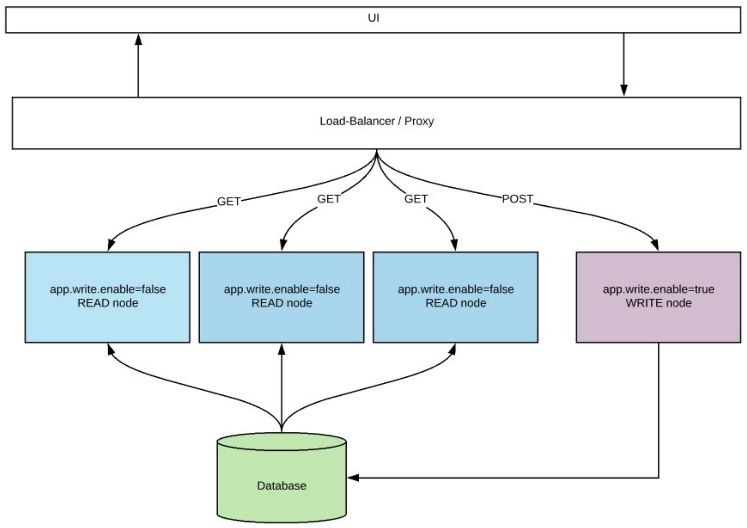
On the **Query** and **Command** controllers, I have added a condition for Spring Boot whether to create this controller or not. That is, below annotation will help creating the controller only if the **app.write.enabled** is set to **true**. Otherwise It would not create this controller bean.

```
//for WRITE controller
@ConditionalOnProperty(name = "app.write.enabled", havingValue = "true")
```

If the value is set to **false**, It will create this controller.

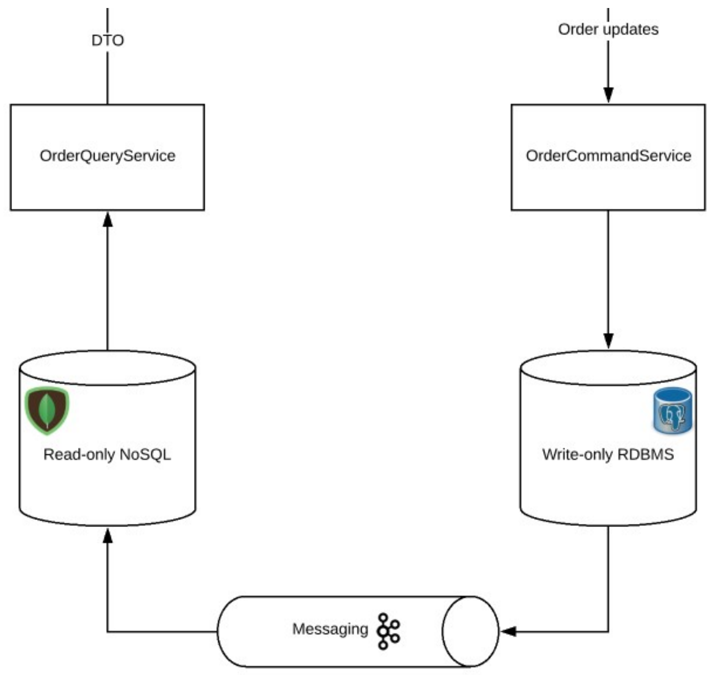
```
//for READ controller
@ConditionalOnProperty(name = "app.write.enabled", havingValue = "false")
```

So based on a property, we change if the app is going to behave like a **read-only** node or **write-only** node. It gives us the ability to run multiple instances of an app with different modes. I can have 1 instance of my app which does the writing while I can have multiple instances of my app just for serving the read requests. They can be scaled in-out independently. We can place them behind a load balancer / proxy like **nginx** – so that READ / WRITE requests could be forwarded to appropriate instances using path based routing or some other mechanism.

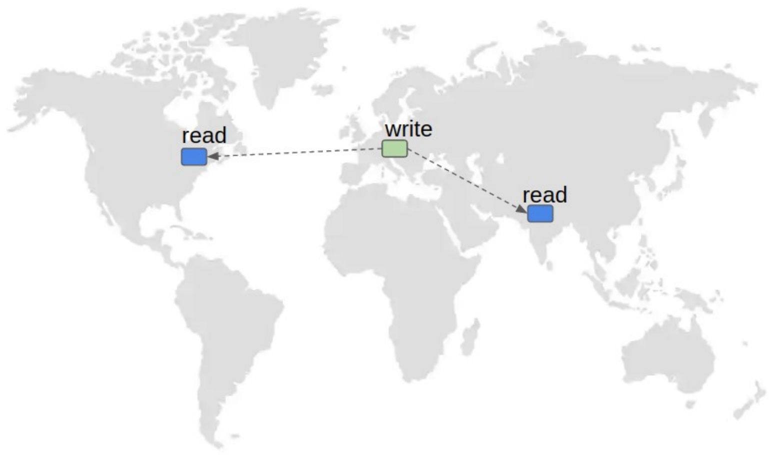

UI
Load-Balancer / Proxy
GET
GET
GET
POST
app.write.enable=false READ node
app.write.enable=false READ node
app.write.enable=false READ node
app.write.enable=true WRITE node
Database

Command vs Query DB:

In the above example we had used the same DB. We can even go one level further by having separating databases for **READ** and **WRITE** as shown here. That is, any write operations will push the changes to the read database via event sourcing.



CQRS Pattern brings additional benefits like using different DBs for the same application with well optimized data schema. However do note that data will be eventually consistent in this approach! As we can run multiple instances of read nodes with its own DB, we can even place them in different regions to provide user experience with minimal latency.



Summary:

Other than easy maintenance and scaling benefits, **CQRS Pattern** provides few other benefits like parallel development. That is, 2 different developers/team could work together on a Microservice. When one person could work on **Query** side while other person can work on the **Command** side.

Read more about **Microservice Design Patterns**.

- Materialized View PostgreSQL – Microservice Design Patterns
- Microservice Pattern – Orchestration Saga Pattern With Spring Boot + Kafka
- Bulkhead Pattern – Microservice Design Patterns

The source code is available here

Happy learning 🙂

Share This: