Reactive Microservice Scalability

Article · January 2019

CITATIONS
3

READS
267

1 author:

Ahmed Ali
University of the Cumberlands
18 PUBLICATIONS **10** CITATIONS

SEE PROFILE

Reactive Microservice Scalability

Author: Ahmed Ali

ABSTRACT

Microservice complexity is hard to define, but the general concept is associated with many components and parts that interact in timeframes to become a system. Complex systems are a subject of research in many prestigious institutions, and a microservices architecture is a perfect case of a complex system. It not only means a shift in technology, but also the way that components are connected, the way that they communicate, and how people collaborate to keep everything running. Reactive web programming is great for applications that have streaming data, and clients that consume it and stream it to their users. It isn't great for developing CRUD apps. If you want to develop a CRUD API, stick with Spring MVC and be happy about it. However, if you're developing the reactive applications with lots of data, a reactive API might be just what you're looking for if the application complexity is medium.

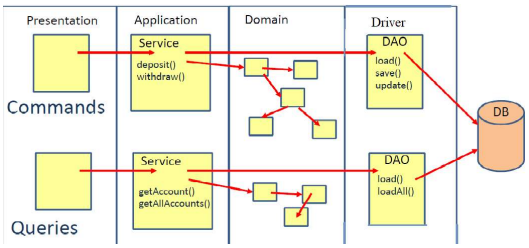
ONE MODEL FOR ALL

One way to support complex views and reporting, the domain model becomes complex Internal state needs to be exposed and so aggregates are merged for view requirements, Repositories often contain many extra methods to support presentation needs such as paging, querying, and free text searching the and the result will be single model that is full of compromises, below example of complex aggregates

```
public class Customer
{
    // ...
    public ContactDetails ContactDetails { get; private set; }
    public LoyaltyStatus LoyaltyStatus { get; private set; }
    public Money GiftCertBalance { get; private set; }
    public IEnumerable<Address> AddressBook { get; private set; }
}
```

CQRS

One of the most popular practices in event-driven architectures today is called CQRS, Command Query Responsibility Segregation CQRS is a style of architecture that allows you to use different models to update and read domain data. Separates the querying from command processing by providing two models instead of one. One model is built to handle, and process Commands One model is built for presentation needs (queries). CQRS is popular for event-driven architectures because domain events as inputs are structurally different. We simply use domain model is used for commands while View model is used for queries.



The separation aspect of CQRS is achieved by grouping query operations in one layer and commands in another layer. Each layer has its own data model (note that we say model, not necessarily a different database) and is built using its own combination of patterns and technologies. More importantly, the two layers can be within the same tier or microservice, as in the example (ordering microservice) used for this guide. Or they could be implemented on different microservices or processes, so they can be optimized and scaled out separately without affecting one another. Because queries can have much more relaxed consistency model (eventually consistent) we can cache the result. So, there is a performance boost. Not to mention that when you look at the traffic they are much more abandoned. CQRS good fit for distributed systems, queries run under relaxed consistency. So that can be called asynchronously with much less thinking about transactions, state change, error.

COMPOSED MODEL MICROSERVICE

Every business Service has a container configuration file, which describes required infrastructure services

including it is API-gateway, which would be deployed as well in a container. Every service would be deployed in a container. Common and infrastructure services would be deployed in container-in-container.

Business Service container can have only once business service but can have one or many instances of this business service, Infrastructure services should work in cluster active-active models like zookeeper and Cassandra so, it works in a highly distributed manner, which support scalability.

SOLUTION

Using Microservices Chassis which provides most of the cross-cutting concerns services with client agents Creating Composite Docker Containers with Docker Compose so every business service container has infrastructure & common services' agent inside its container.

REACTIVE REST WITH SPRING WEBFLUX

Reactive Streams provide non-blocking response in Message-driven environment typically there are different types of Implementations the most popular are, JavaRx (Netflix) and Reactor (Pivotal) Spring WebFlux framework is part of Spring 5 and provides reactive programming support for web applications

REACTOR

Mono<T>: for handling 0 or 1 element
Flux<T>: for handling N elements
You can subscribe to a Mono or a Flux Run some code when an object arrives in the Mono or Flux, below is a code snapshot of Mono implementation, Whenever the data arrives in the mono, print it out the Callback method, The Callback method here is printResult, we used delayElement just to show how it work but in the real application we should not use delay.

Dependency for spring-boot-starter-webflux dependency, which pulls in all other required dependencies:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

```
public static void reactor() throws InterruptedException {
    System.out.println(LocalDateTime.now());
    Mono<String> mono = Mono.just("send Something ").delayElement(Duration.ofSeconds(3));
    mono.subscribe(s->printResult(s));
    Thread.sleep(10000);
}

public static void printResult(String something) {
    System.out.print(LocalDateTime.now()+" : ");
    System.out.println(something);
}
```

The printed results

```
2019-07-06T09:50:40.429
2019-07-06T09:50:43.764 : send Something
```

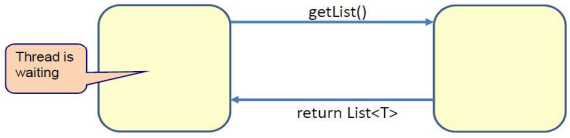
FLUX

In the sameway we could use Flux for reactor and Callback response

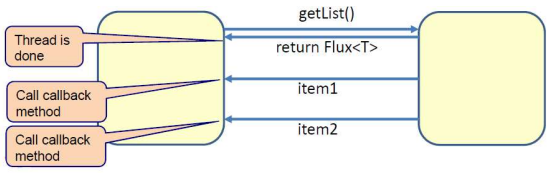
```
Flux<String> flux = Flux.just("A001", "A002", "A003", "A004").delayElements(Duration.ofSeconds(3));
flux.subscribe(s->printResult(s));
Thread.sleep(60000);
```

```
2019-07-06T10:00:14.627 : A001
2019-07-06T10:00:17.634 : A002
2019-07-06T10:00:20.635 : A003
2019-07-06T10:00:23.636 : A004
```

SYNCHRONOUS, BLOCKING



ASYNCHRONOUS, NON-BLOCKING



Advantage
Better Performance; Fast response, no need to wait till all results are available,
Scaling Architecture, less threads needed
Disadvantage
The whole calling stack needs to be reactive;
Client, controller and data access
Harder to debug

REFERENCES

[1] Apply simplified CQRS and DDD patterns in a microservice – Microsoft - apply-simplified-microservice-cqrs-ddd-patterns.

[2] Guide to Spring 5 WebFlux - spring-webflux