Performance Benchmarking of Reactive Programming with MongoDB:

Introduction

We will discuss reactive programming, specifically used while creating REST APIs to serve the clients' requests over HTTP. We will look at the performance improvement using this new approach and comment on why and when to use it. I will demonstrate using Java 8, Spring boot 2.7.13 with the latest MongoDB atlas. I will also use Jmeter 5.6.1 to perform load tests. The Project details are at the end of the document.

Why Reactive Programming?

So, What's wrong with the traditional approach? I will call this traditional approach Synchronous because it is blocking. Blocking in the sense that the thread would wait to return the result while one call is made to the database.

We have a beautiful concept of multiple threads while working with synchronous code. We are allowed to use multiple threads simultaneously. I have implemented a GREP command in Java manually and used this concept. The results were impressive; my implementation with multiple threads worked faster than the in-built implementation of the GREP command.

Manual Implementation with threads	Default command from a terminal
7.6 seconds	78.7 seconds

Steps to run GREP Implementation:

Here is the link for the implementation:

https://github.com/MrSMS0000/PROJECT_CODE_RESULTS/tree/master/Grep_Implementation

I have implemented and tested it using different I/O methods, like Buffered IO, Random Access File, and NIO.

To demonstrate, one can clone the `Grep_Implementation` in any IDE and run the `main()` method of the `GREP` class in the `buffered_with_threads` package with the command line arguments “word_to_find” and “path_to_directory_or_file”; I have got these results with `test_large` folder containing five identical files with 1 GB size each and the word `namespace` appearing many times in those files.

Also, check with the terminal by using the command:

```
time grep -r any_word_like_namespace path_to_folder/test_large
```

Coming back to the question, Why Reactive?

The problem with this multi-threaded concept is “Threads are limited.” We can scale the performance to some extent using multi-threading, but after reaching the limit of threads, there is no way other than horizontal scaling to improve the performance or to serve more requests. In the next section, we will see how reactive programming helps to achieve higher performance beyond the limit of threads.

How does Reactive Programming work?

Instead of waiting for the result in Reactive programming, the thread calls the database and moves ahead. It is ready to take other requests. Once the database sends the results, the thread takes it and returns it to the client.

Restaurant Analogy

We can understand it by the restaurant analogy. Assume that the manager takes the order and sends it to the cook. Once an order is ready, the manager will serve it to the customer. Here, the manager works like a server, and the cook works like a database. Food preparation time is the time the database takes to return the result.

In a synchronous way of coding, the manager takes the order, sends it to the cook, and waits for it to be ready. After the order is prepared, the manager will serve it and take another order.

While in the reactive way of coding, the manager is not waiting for the order to be ready. The manager can take other orders while one order is being prepared. So, the manager's work comprises taking and serving orders rather than waiting for orders to be ready. Provided the cook can handle multiple orders, now the restaurant as a whole can serve a higher number of orders at the same time.

Illustration

This animated GIFs illustrate the working of reactive programming compared to traditional synchronous programming.

Synchronous:

Reactive:

Note that the red light shows the thread is busy. In reactive programming, it just receives a request and sends a response back. For further comparison, we assume that the time taken doing this work is negligible compared to the time taken by the database. If the database is too fast to serve the response, this time cannot be negligible. Also, note that the server is generally divided into controller and service parts. This GIFs are just for illustration.

How much fast?

So with this, reactive programming can handle parallel requests with limited numbers of threads also. Even with one thread only, it can send multiple requests to the database, which is impossible in synchronous programming. This makes a difference in the performance of handling multiple requests at a time. From our understanding and assumption, reactive programming can handle all the requests in parallel.

Assume we have a REST endpoint named `getMessage`, which takes t seconds to execute database operations, and we have R number of requests to access that endpoint.

	Time to complete all the requests Seconds	Throughput Requests/second
Synchronous	$R * t$	$1/t$
Reactive	t	R/t

Now, We will test the endpoint and check if the result matches our assumptions.

Testing - without database

Let us test the REST endpoint, which contains no database calls. Instead, we will put a delay element in our code to test the performance. I have made a simple endpoint named, which waits for 2 seconds and then returns a string in both reactive and synchronous programming. We will call this endpoint for both by different numbers of users.

Steps to run Implementation:

Here is the link for the implementation containing code for both endpoints with the Jmeter test file and results.

https://github.com/MrSMS0000/PROJECT_CODE_RESULTS/tree/master/Delay

In this folder, We can find two projects, `delay-reactive` and `delay-synchronous`. To start the server, we need to run `DelaySynchronousApplication.java` and `DelayReactiveApplication.java`.

If you have Jmeter installed on your machine, you can run `delay-jmeter.jmx` or see my results in an Excel file `delay-results.xlsx` with screenshots of the results in `delay-result-screenshots`.

Service code

Synchronous:

```
public String getMessage(Integer delay) {

    // waiting for delay milliseconds in synchronous way.
    try {
        Thread.sleep(delay);
    }
    catch (InterruptedException e) {
        System.out.println("Error in Thread.sleep()");
        throw new RuntimeException(e);
    }

    return "Hello! after "+delay+" milliseconds.";
}
```

Reactive:

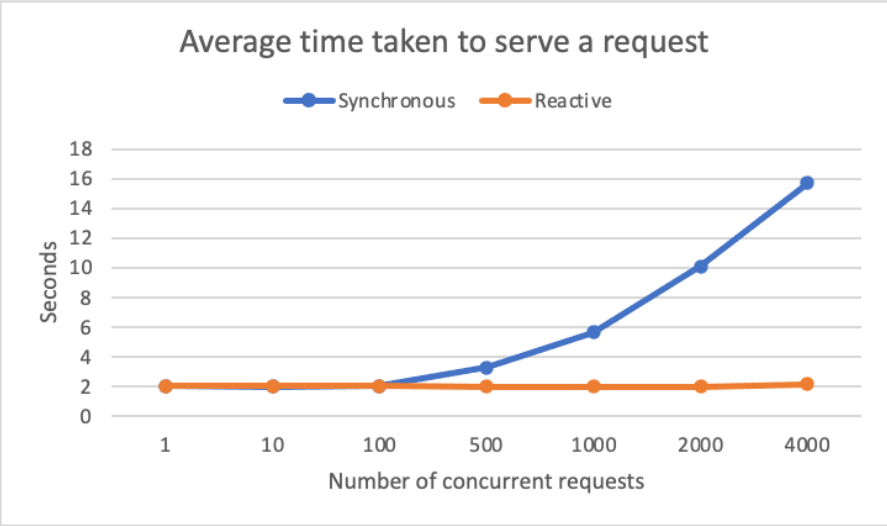
```
public Mono<String> getMessage(Integer delay) {
    return Mono.just( data: "Hello! after "+delay+" milliseconds.")
            // waiting for delay milliseconds in reactive way.
            .delayElement(Duration.ofMillis(delay));
}
```

Results

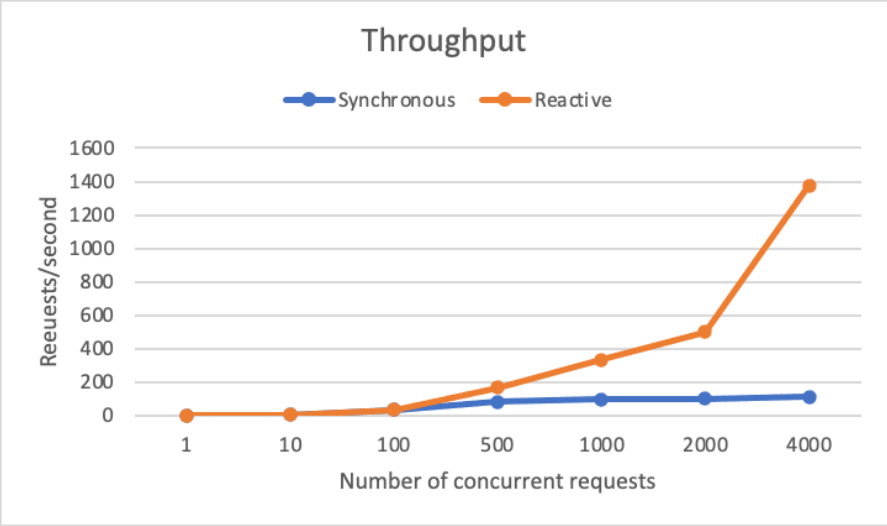
Samples	Average time seconds	Average time seconds	Throughput Req/sec	Throughput Req/sec	Error	Error
	Synchronous	Reactive	Synchronous	Reactive	Synchronous	Reactive
1	2.004	2.005	0.50	0.50	0.00%	0.00%
10	2.002	2.004	3.44	3.44	0.00%	0.00%
100	2.003	2.003	33.38	33.40	0.00%	0.00%
500	3.284	2.002	80.65	166.67	0.00%	0.00%
1000	5.643	2.002	97.13	332.89	0.00%	0.00%
2000	10.115	2.002	98.86	499.38	0.00%	0.00%
4000	15.729	2.155	110.94	1375.99	13.88%	0.00%

Graphs

Average time:



Throughput:



Comments

The results shown above clearly support our understanding of reactive programming. Every time it almost took only. $t = 2$ seconds to serve the requests. Throughput is also following the increasing pattern. However, it does not match exactly with the formula because it is calculated as the total time/number of requests, and the total time might disregard our assumption.

On the other hand, synchronous programming takes the same time as reactive programming, up to 100 requests. This is where threading comes in. Tomcat server, by default, allows a maximum of 200 threads. So, all those requests are executed parallelly and served in just $t = 2$ seconds. After increasing the number of requests to more than 200, every 200 requests execute parallelly, another 200, another 200, and so on. To serve 1000 requests, it did not take $1000 * 2$ seconds but only took near $(1000/200) * 2$ seconds. Also, note that there is a 13.88% error in synchronous programming. This is because handling these many requests in synchronous programming took too long.

This leads to updating the idle formulas. Suppose we have T threads together serving requests.

	Time to complete all the requests Seconds	Throughput Requests/second
Synchronous	$\lceil R/T \rceil * t$	$R/(\lceil R/T \rceil * t)$
Reactive	t	R/t

Look at the fact that, ideally, having T number of threads does not affect the performance of Reactive programming because we have considered the work of the thread in the reactive programming as negligible.

Testing - with database

Once we have confirmed results, let us move further and replace that delay with some database call to get actual performance improvement in real scenarios. For that, I made a JSON document containing a long string of 1 MB (I took such a big document so that it follows our assumption). I also created an endpoint to add that document to the MongoDB atlas database. Then for the testing, I am fetching that document using normal and reactive MongoTemplate by the `getMessage` endpoint.

Steps to run Implementation:

Here is the link for the implementation containing code for both endpoints with the Jmeter test file and results.

https://github.com/MrSMS0000/PROJECT_CODE_RESULTS/tree/master/Database

In this folder, We can find two projects, `database-reactive` and `database-synchronous`. To start the server, we need to run `DatabaseReactiveApplication.java` and `DatabaseReactiveApplication.java`.

If you have Jmeter installed on your machine, you can run `database-jmeter.jmx` or see my results in an Excel file `database-results.xlsx` with screenshots of the results in `database-results-screenshots`. You can also find the document used to being fetch from the database here `data.json`.

Service code

Synchronous:

```
public Entity getMessage(String id){
    return template.findById(id, Entity.class, AppConfig.collectionName);
}
```

Reactive:

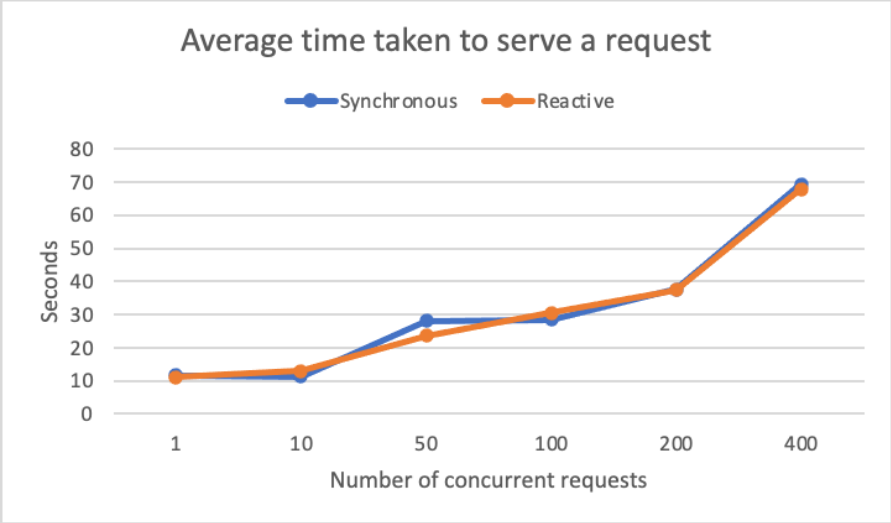
```
public Mono<Entity> getMessage(String id) {
    return template.findById(id, Entity.class, AppConfig.collectionName);
}
```

Results

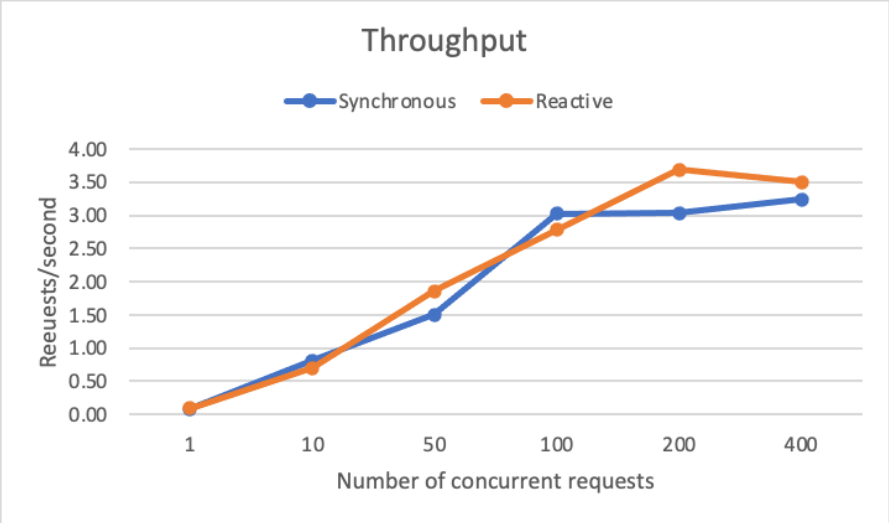
Samples	Average time seconds	Average time seconds	Throughput Req/sec	Throughput Req/sec	Error	Error
	Synchronous	Reactive	Synchronous	Reactive	Synchronous	Reactive
1	11.698	11.082	0.09	0.09	0.00%	0.00%
10	11.271	12.944	0.81	0.70	0.00%	0.00%
50	28.206	23.601	1.50	1.86	0.00%	0.00%
100	28.492	30.599	3.03	2.78	0.00%	0.00%
200	37.661	37.491	3.04	3.69	0.00%	0.00%
400	69.355	67.815	3.24	3.50	0.00%	0.00%

Graphs

Average time:



Throughput:



Comments

The results are not complying with the concept of reactive programming. Both of them gave similar performances. Even when the number of requests is 400, beyond the number of threads being used, both gave almost identical results.

It was hard to find that the default maximum connection limit to mongoDB is 100 connections only. That means that we can connect only 100 parallel requests from our Java application. This is the reason behind the poor performance of the reactive programming and the difference between the results we tested without a database and with a database. After 100 connections, reactive programming will take requests in but cannot send them to the database; that will be bottlenecked and won't increase the performance. Whether we use MongoDB atlas or MongoDB local, it only allows 100 connections by default, which can be configured to 500 at maximum.

This, again, leads to a change in our formulas. Suppose N is the maximum number of parallel connections to the database.

	Time to complete all the requests Seconds	Throughput Requests/second
Synchronous	$\lceil R/min(N,T) \rceil * t$	$R/(\lceil R/min(N,T) \rceil * t)$
Reactive	$\lceil R/N \rceil * t$	$R/(\lceil R/N \rceil * t)$

Testing - with database fixed

To fix the issue of limited connections, I setted it to 500 , but the machine was not able to make To fix the issue of limited connections, I set it to 500, but the machine could not make 500 connections from each application simultaneously. So, I increased it to a maximum of 200 connections. Still, the testing wouldn't give actual results because we have 200 threads in synchronous programming, as shown above, and 200 connections in reactive programming. In real life, a server can connect by multiple connections to the database and not only by 200, but it will certainly have a limited number of threads. We can limit the number of threads in both applications to mimic this situation of numerous connections. It will give us insights into the performance comparison when we would have allowed to have multiple connections.

So, I made the maximum connections to 200, and the number of threads allowed is 10 in both applications. Now, we can test the same endpoint with everything else remaining the same.

Steps to run Implementation:

Here is the link for the implementation containing code for both endpoints with the Jmeter test file and results.

https://github.com/MrSMS0000/PROJECT_CODE_RESULTS/tree/master/Database Fixed

In this folder, We can find two projects, `database-reactive-fixed` and `database-synchronous-fixed`. To start the server, we need to run `DatabaseReactiveFixedApplication.java` and `DatabaseSynchronousFixedApplication.java`.

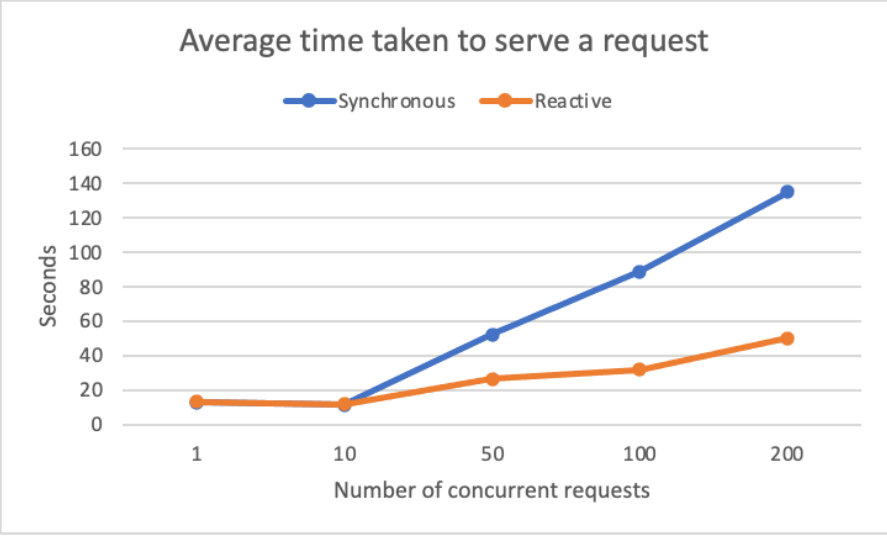
If you have Jmeter installed on your machine, you can run `database-fixed-jmeter.jmx` or see my results in an Excel file `database-fixed-results.xlsx` with screenshots of the results in `database-fixed-screenshots`.

Results

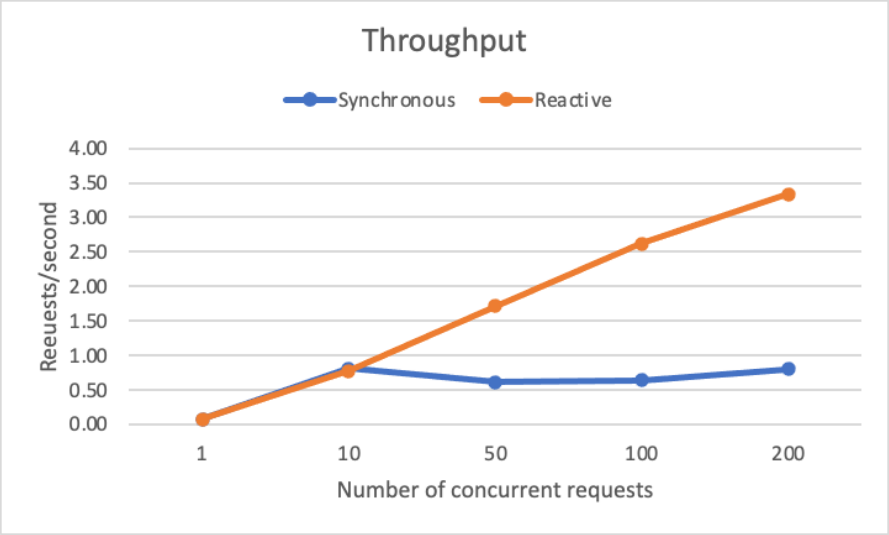
Samples	Average time seconds	Average time seconds	Throughput Req/sec	Throughput Req/sec	Error	Error
	Synchronous	Reactive	Synchronous	Reactive	Synchronous	Reactive
1	12.785	13.231	0.08	0.08	0.00%	0.00%
10	11.351	11.848	0.80	0.78	0.00%	0.00%
50	52.352	26.562	0.61	1.71	0.00%	0.00%
100	88.659	31.903	0.64	2.62	0.00%	0.00%
200	135.014	50.021	0.80	3.34	0.00%	0.00%

Graphs

Average time:



Throughput:



Comments

Since the number of connections was limited, reactive programming gave us the same results and was not affected by the number of threads as derived in the formulas. But limiting the number of threads from 200 to 10 changed the results of synchronous programming. So, using limited resources, reactive programming gives better throughput than synchronous one. These results support our derived formulas for the time a request takes and the server's throughput.

Further Modification

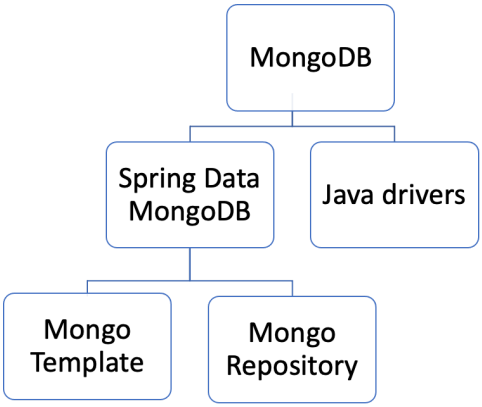
We have tested a simple REST endpoint with just a single database call. But it can have multiple database calls and some complex logic also. Let's assume another endpoint has x numbers of independent database calls. Independents in the sense that it is unnecessary to complete any other database call before it. In this case, Reactive programming will make $R * x$ parallel calls instead of just R calls. But, in synchronous programming, it must wait and complete all the x calls sequentially. Let us further modify our equations for this.

	Time to complete all the requests Seconds	Throughput Requests/second
Synchronous	$\lceil R/min(N,T) \rceil * t * x$	$R/(\lceil R/min(N,T) \rceil * t * x)$
Reactive	$\lceil R * x/N \rceil * t$	$R/(\lceil R * x/N \rceil * t)$

Notice that x is treated in both equations differently. This is because even though having extra threads, it cannot execute those x calls parallelly in synchronous programming. At this point, we should remember that we have neglected the time taken by the thread in reactive programming. Here, for the x parallel calls from the same endpoint will take some time because, for i^{th} call, the time passed to send it to the database is accumulated by all the previous calls. This will make a difference in the results from our ideal formulas. Also, we have a limited number of connections here. So, reactive programming will work parallelly only till (N/x) number of requests, where N is the number of maximum connections allowed.

Comparison for Mongo Connections

There are three ways to connect our Java application to the MongoDB database. All of them can be used with reactive and synchronous both.



I have tested some of the standard CRUD methods in all six ways.

Here are the links for the code:

- Reactive Java Driver
- Reactive Mongo Repository
- Reactive Mongo Template
- Synchronous Java Driver
- Synchronous Mongo Repository
- Synchronous Mongo Template

You can find the results and the Jmeter test file of the different testing here: Mongo Connections

Comparison

Using MongoDB Java drivers, we can connect our Java application to MongoDB irrespective of using the Spring framework. This is the standard way and gives more flexibility to write different queries. The result shows that if one is flexible with Java drivers, this is the best way to connect to MongoDB for performance. However, code may become more complex.

Using Mongo Template of Spring Data MongoDB is a simple yet flexible way to connect with MongoDB using the Spring framework. It is more convenient and readable to use this.

Finally, Mongo Repository of Spring Data MongoDB is the easiest way to write queries. We can declare our queries with methods without any implementation. MongoRepository provides all the basic functionality in a much simpler way. However, for custom or complex queries, I recommend to use Mongo Template. Performance-wise, both of them were almost identical.

Comments

For the results, again, we have a limited number of connections. So, we won't get performance improvement using reactive programming. If I had put the limit on the threads, the performance of the synchronous programming would have been degraded. Also, Jmeter cannot send a request with more than 60-70 KBs of the request body data, which leads to faster database operation(a few tens or hundreds of milliseconds) and inaccurate results.

I also noticed that for a collection with multiple documents, the `findAll()` and `findByQuery()` methods provided by the ReactiveMongoTemplate and ReactiveMongoRepository are very slow compared to synchronous methods. I tried to search for this on the internet and found the same issue by other developers. Looking for a new release of Spring Data MongoDB with this fix. Also, found a hack to solve is to use these methods with pagination and sort.

Conclusion

With all these analyses and testing, I believe that reactive programming performs better than the traditional synchronous approach. We can apply this new programming paradigm and achieve massive scalability in some real-life scenarios.

Video streaming is the best example of reactive programming. Apart from this performance improvement, reactive programming has some concepts of the publisher, subscriber, backpressure, and error handling. Netflix uses this reactive programming for video streaming, successfully serving millions of customers. Since video streaming contains heavy database calls and follows to react on events like play and pause, it works better with reactive programming.

To serve some requests, we need to call multiple databases. In the microservices-based architecture, it is very often that one microservice calls other microservices for data. In those cases, parallel calls would greatly help, serving more requests. Today, many organizations are moving to establish microservices-based architecture and reactive programming can be used for inter-communication between microservices.

For another example, reactive programming can be used for the application showing the data of cricket matches. At every ball, we need to update the score. Also, we need to update the player's record. We also need to post a commentary for that ball. We can do all these tasks in parallel with reactive programming. Millions of users simultaneously access the data for this type of application.

With the benefits of reactive programming, one must pass the learning curve to write the code in a reactive way. It needs some practice and time to get used to it. Also, sometimes it is challenging to debug where our code is getting blocked and bottlenecking the whole flow. To use reactive programming, we must make every API reactive from end to end; otherwise, it would block the thread. Since other APIs we use have yet to be reactive, using reactive programming in a massive project is a big challenge.

Project details

Name: Sarthak Sonagara

Mentor: Sudhanshu Bansal

Manager: Sandeep Maheshwari

Timeline

This is the outcome of my 8 weeks of internship project at Sprinklr.

- Week 1: Introduction and basics of Java
- Week 2: Core concepts of Java
- Week 3: Spring Core, Spring MVC and Spring Boot along with MongoDB basics
- Week 4: REST with MongoDB and Spring Boot, Implementation of GREP with threads
- Week 5: Basic concepts of Reactive Programming, Spring Data Reactive MongoDB
- Week 6: Comparison of Different ways to connect with MongoDB
- Week 7: Testing with and without Database
- Week 8: Construction of formulas and documentation