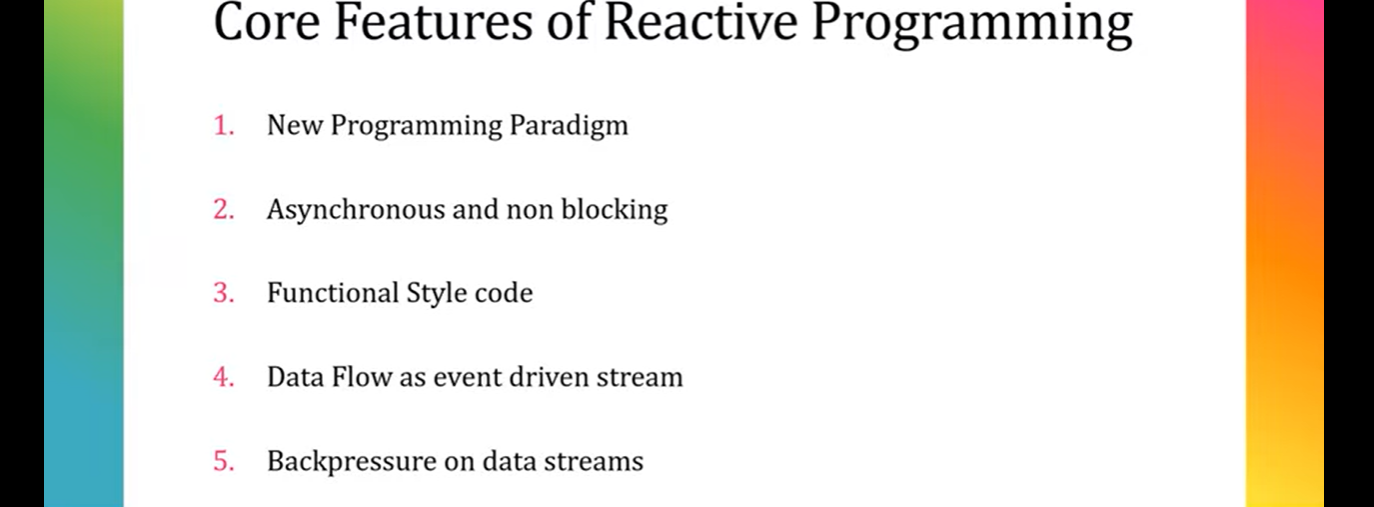
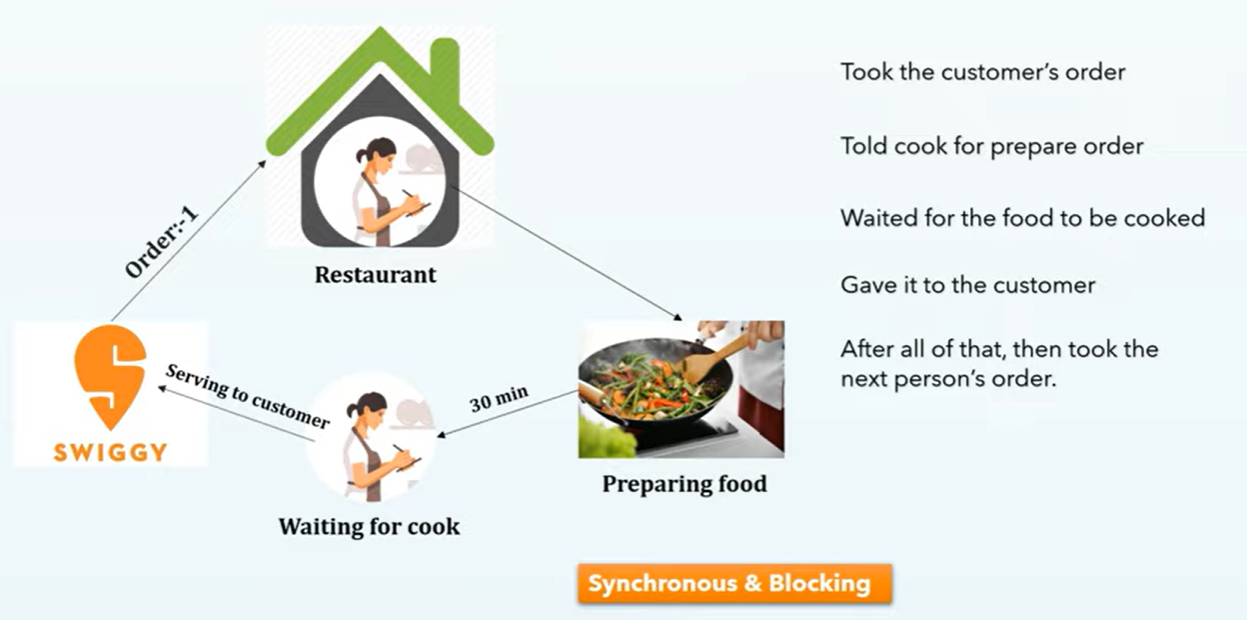
* **Spring Boot - Introduction to Reactive Programming**

**--------------------------------------------------------------**

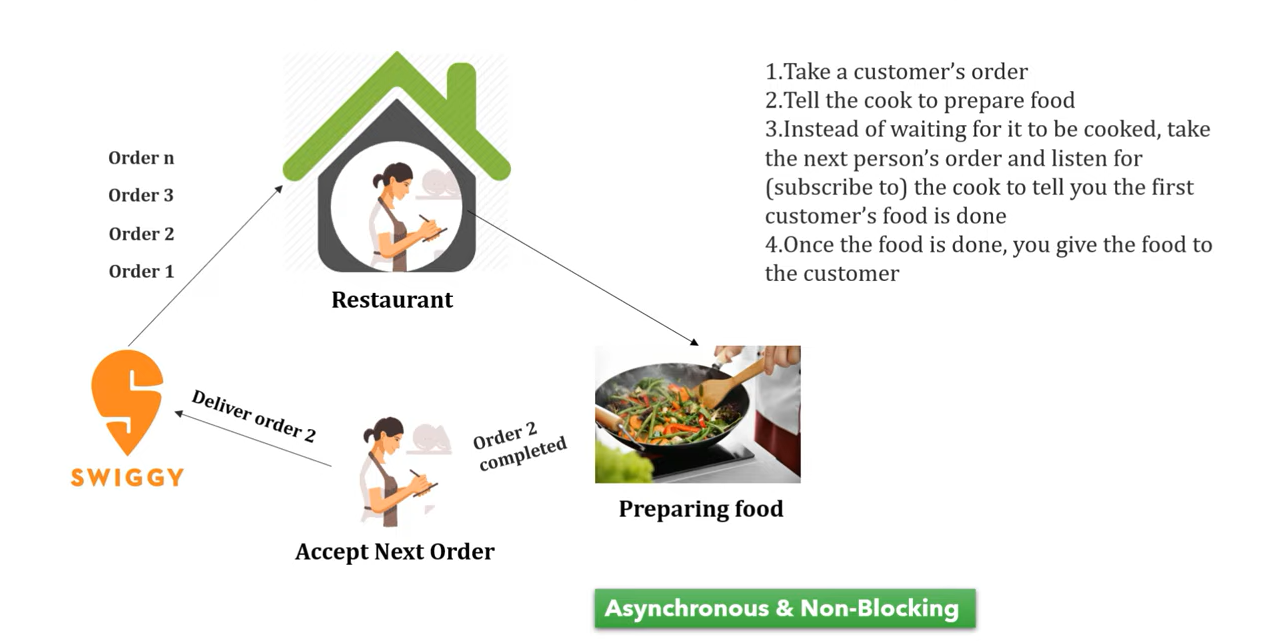


We will discuss about reactive programming. Basically, we will cover **what** is reactive programming, **why** we need this reactive programming and **when** to use this reactive programming.

Basically, these are the core features. This is a new programming paradigm, and it is going to a new complete shift the way we are designing our application previously. Also, as per the documentation, it is asynchronous and non-blocking. If you are familiar with java 8 stream API, you may need to write lot of functional style of code. It supports data flow as event driven stream. Also, it supports backpressure on data streams. Don't worry, each core features will understand in detail with example. So, let's begin with what is asynchronous and non-blocking.



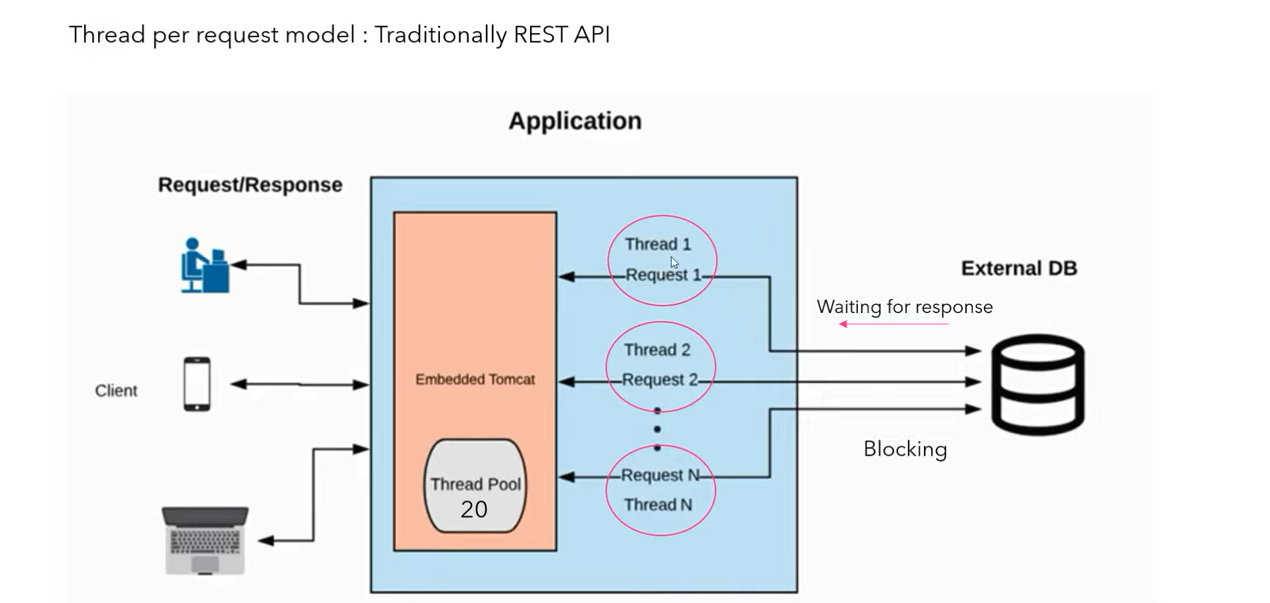
Let's assume I am the owner of one restaurant, and I can accept order from any third party. It can be swiggy, it can be zomato or I can directly accept the order from customer itself. Let's assume I accept the order from swiggy, that is order 1.Then I pass that order to my cook for prepare the food. Now I just wait for food to be cooked. After few minutes, my cook updates me, the order is ready. It took around 30 minutes. Once I receive the order, I will serve it to the customer. Then again, once I serve, I will accept another order, let's say order 2. I will wait for food to be cooked. Once I got the food, I will give it back to the customer. After all of that, I will take next person's order. So, if you observe, the flow of execution is one after another, which is nothing but synchronous flow. I will accept the order, then I will follow all the 4 steps, then I will accept another order. That is what called synchronous and blocking flow. Because my restaurant is completely blocked for the cook, until I will not get response back from the cook, I am not accepting the next order, which is completely synchronous and blocking. And also, if I will continue my restaurant in same pattern, then I could not deliver more than 50 orders per day. And you would have a line out the door and customers waiting for long, which is not in a good way. Let's optimize this in a reactive programming manner, which is asynchronous and non-blocking.



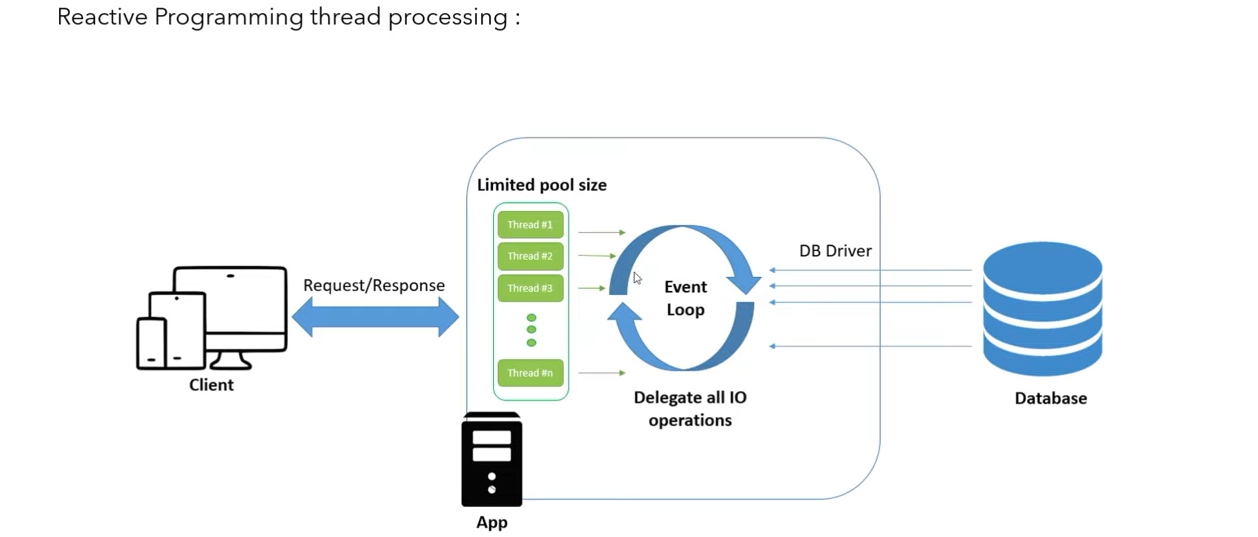
Let's optimize this in a reactive programming manner, which is asynchronous and non-blocking. I accept the order from swiggy, which is order 1. Then I pass it to my cook for preparing the food. Now instead of waiting for it to be cooked, I will take the next person's order and I will tell to my cook, you just let me know once you are ready with the first order. Then again, I will accept the next order. I will again not wait for my cook to prepare the food. Then again, I will accept another order. I will accept n number of order. My cook can continue cooking the food based on the order priority. Let's assume cook is ready with any of the order. It can be order 2, order 1 or order 3. Once I listen from the cook that order is ready, I will serve it to the customer. So, in this approach, what we are doing, we are taking the customer's order. Then we are telling the cook to prepare the food. Then instead of waiting for it to be cooked completely, we are accepting the next person's order and we are waiting for our cook to tell you the first customer's food is done. Once we receive the order confirmation from our cook, we are returning back it to the customer or we are serving to the customer, right? Which is completely asynchronous and non-blocking. Now my restaurant is not blocking with this cook. I am ready to accept n number of order per day where I will just inform to my cook, you just let me know once you are done with the order rather than waiting for it, isn't it? Which is completely asynchronous and non-blocking.

If you compare in this asynchronous and non-blocking, you can achieve a lot of advantages. For example, we can handle more requests with the same amount of employee by using a publisher and subscriber model. So here the restaurant owner is a subscriber and cook is a publisher. I just subscribe to this cook, and I just inform him, you just let me know with the event when you are ready with your cook or food. So, this cook will be publisher. He will publish the event, I'm done with the food, you can deliver it. So as a subscriber, I can hear that. That is what and the next we can say you get more work done during the time you would normally be waiting for the cook to cook the other order. That is what we understood in our previous slide.

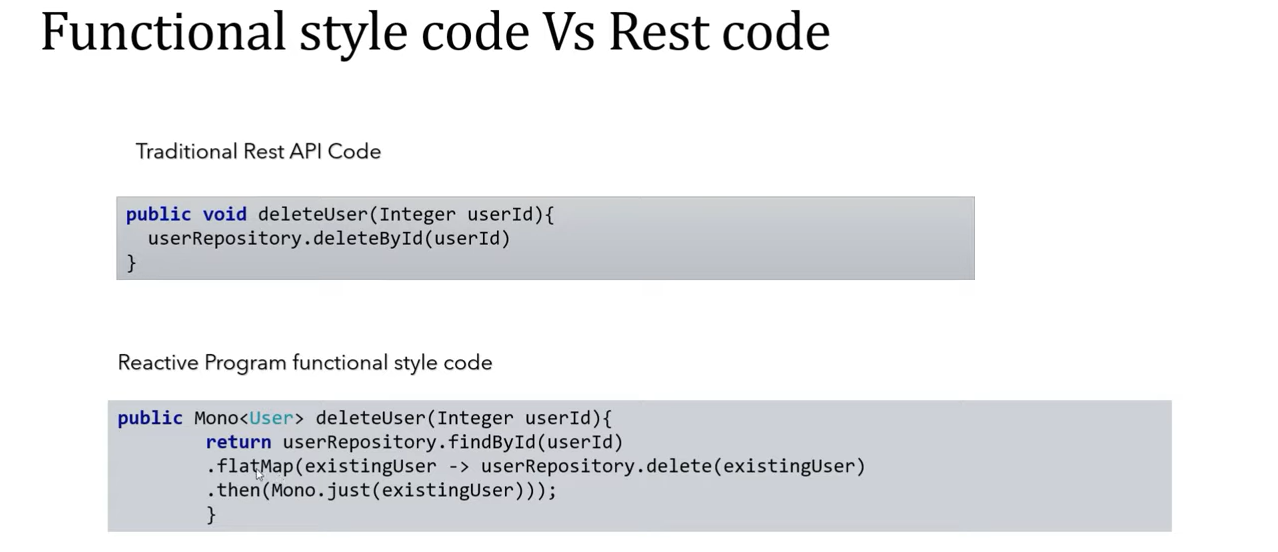
You don't need to wait for the cook to be prepared the food, just accept the order, and process it. This is what the advantages in case of asynchronous and non-blocking. Hope this is clear for you. Now let's link this concept to our application-level understanding.



Usually, traditional REST API work done thread per request model. If you can see this diagram, let's assume client send request to my application. As you already know, each request will be served by a separate thread. So here we can consider request one assigned to thread one. Now this thread one goes to DB and try to fetch the database. Now here until thread one gets response back from DB, he is completely blocked. So, he's waiting for the response. Similarly, let's assume we received another request to application, and it is serving by thread two and again it goes to DB and waits for its completion until then it is blocked for a long. Similarly, we can process n number of requests, but in real time there is a thread pool limitation. Let's consider my application thread pool limit is 20.It means my application can handle only 20 concurrent requests at a time. If my application received 21 request count, then that request need to be wait until any of existing thread is free because all my thread is blocked by this database driver for getting the response, which completely give us poor performance as it seems blocking and synchronous flow because all the thread is already occupied with this DB driver to get the response. That's the reason 21 count request is not immediately handled by any of the thread. So that's where reactive programming came into picture, and it eliminate use of thread per request concept.

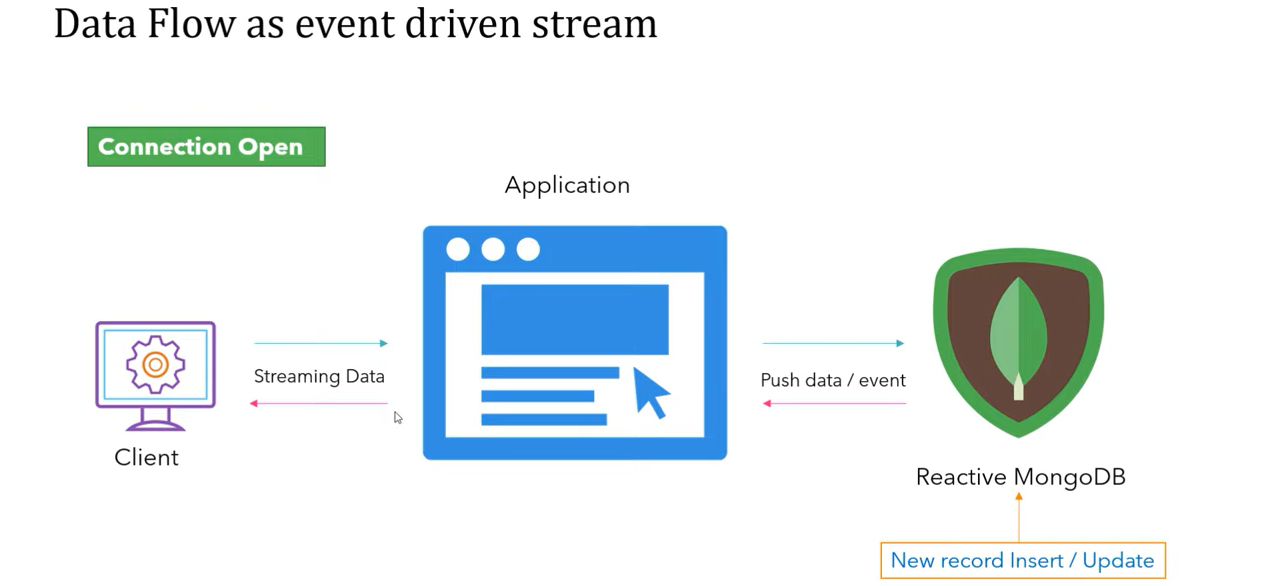


Now let's understand how it works on reactive programming. In reactive programming, let's say request came to my application and it assigned to thread one. Now thread one will go to database for fetching the data, but the advantages here, he will not wait to get response back. Instead, he'll send an event to database, and he'll inform to database, hey, I'm not going to wait to you anymore. You just do your job. Whenever you will be ready with response, assign that response to available thread and just publish me a complete event. Now in that case, my thread is completely free. He can happily accept any number of requests because not a single thread is blocking in this event loop approach. With this approach, you can handle tons of concurrent requests with very less thread. That is what asynchronous and non-blocking request assigned to the particular thread and thread is not going to wait for database driver to get the response. He will continue his process. He’ll accept a number of requests simultaneously. That is what all about asynchronous and non-blocking.

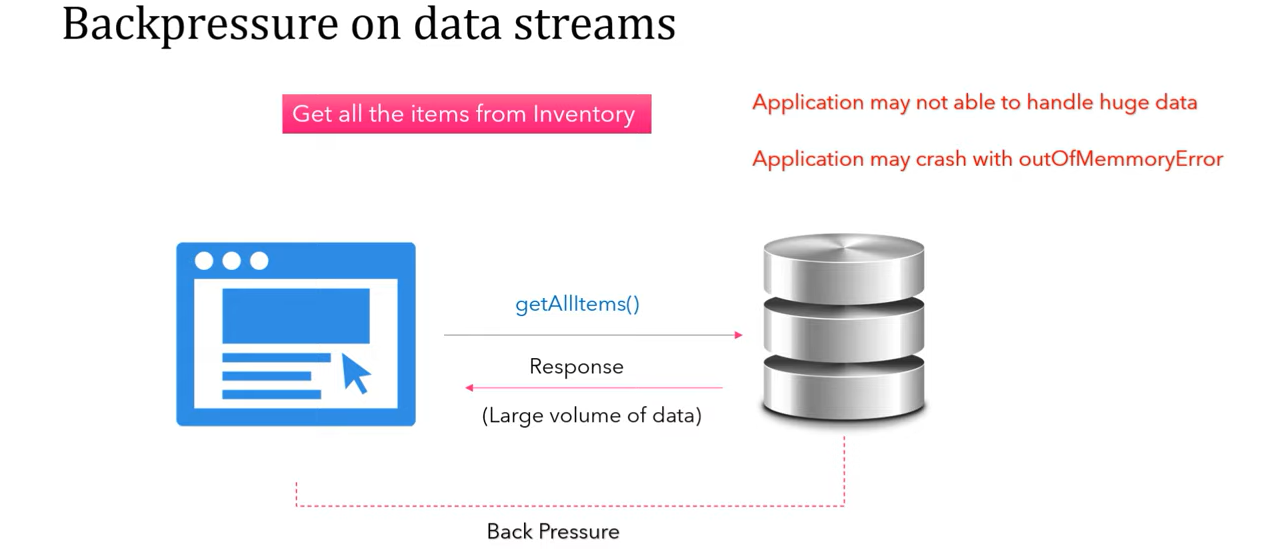


Let's move into the next core feature that is functional style code. In traditional REST API code, let's say I just want to delete a user from database. I can write something like this. Delete user, you can pass the argument. Then you just call your database.deleteById and pass the argument. It will delete the object from database.

But in case of reactive programming, the syntax is bit different. There is a data type called **Mono** who can handle single object. Also, there is another data type called **Flocks**. He can handle zero to n number of objects. Then you can write delete user, then pass the argument, then get the existing user, then use the **.flatMap** and there is a syntax called **Mono.just**. So, the syntax in reactive programming, it is similar Java 8 stream API. If you are not familiar with Java 8 stream API functional interface uses, you can check out my Java 8 playlist. So that it will be easy for you to understand the reactive programming syntax, which is nothing functional style code.



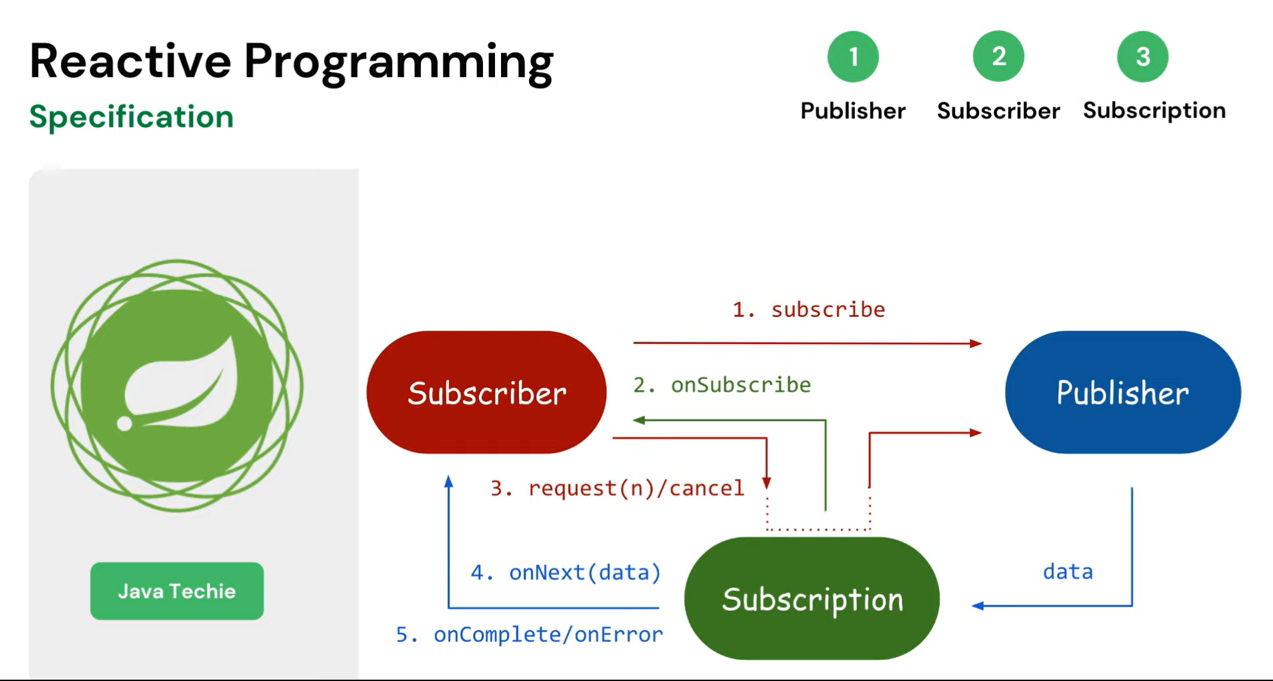
Now let's move into the next core feature of reactive programming, which is data flow as event driven stream. So, if you can see this diagram, typically in traditional REST approach, clients send request to application, then request goes to database and process the data. Then finally it returned response back to the client. But what if there is a few records inserted or modified in middle? Now client don't have those update information. To get that update information, again, he may need to do a separate REST API call. But how it works in reactive programming? If there are any changes done in database, then immediately it will fire or it will publish an event. Hey, there is a new data found. Now whoever subscribe, they can stream the data. In this case, if you observe my client is a subscriber and my database driver is a publisher. If any changes happen to my DB, we'll publish an event. As my client is consumed to my application, he can easily stream the data. And also in this scenario, your connection will be always open state. That's the reason we can perform this publisher subscriber operation. We can consider any streaming platform that work in reactive programming manner. The best example you can consider cricket live score. We're seeing the cricket live score, right? Any update happened in the application, we can see the current score with the updated details. So, this is how the data will be transferred as an event on the publisher and subscriber model. That is what dataflow as event-driven stream, which was supported by reactive programming.



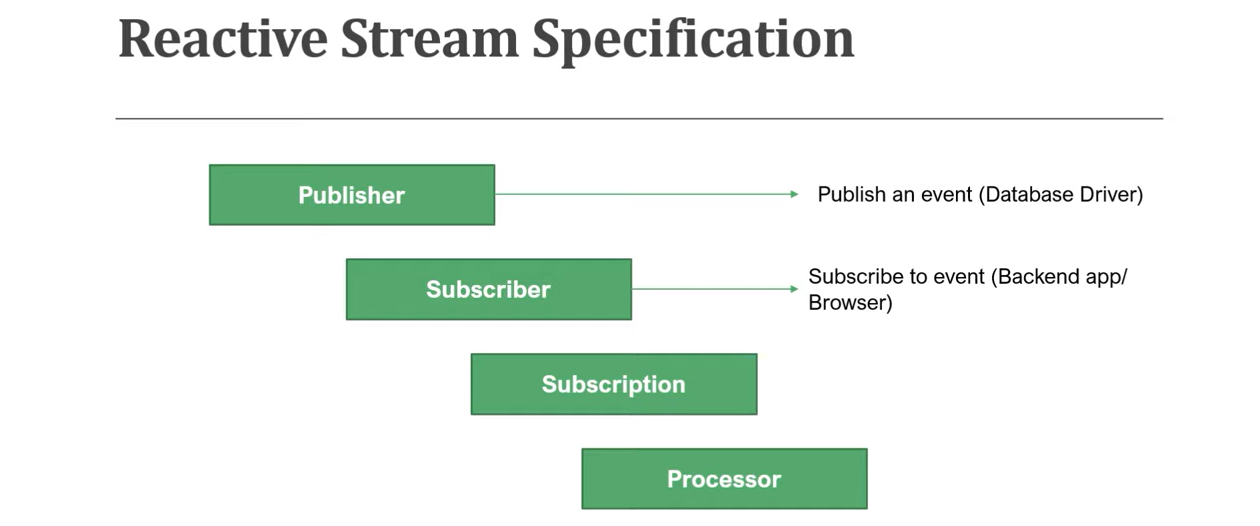
Let's move into the next core feature of reactive programming. That is backpressure on data streams. Let's assume I just want to get all the items from inventory. Now my application send request to database and database return response as a large volume of data. In that case, my application may not be able to handle huge amount of data, or my application may crash with out of memory because no backpressure feature was available for traditional REST API. But in case of reactive programming, there is a concept called backpressure. If your database gives huge data and your application not able to handle it, you can simply tell your database, hey, please slow down the response. Let me process with whatever data I have with me. Then you can push other data. So, this backpressure support only applicable for reactive programming, where you can add a limitation on database driver, how much data you are expecting. You don't want to load all the data. You can inform to your database driver. He can slow down the data based on your configuration. That is what the advantages of using this backpressure. So far, we cover all the theoretical terms of this core features of reactive programming. Don't think too much about its internal details of how it works. So far, we just understand the advantages of reactive programming. We can achieve proper CPU utilization and there is no application downtime, and the flow of execution can be concurrent because it supports asynchronous and non-blocking. Also, it's streaming the data as an event using publisher and subscriber model.

* **Spring Boot WebFlux | Reactive Streams Specification and workflow | Project Reactor**

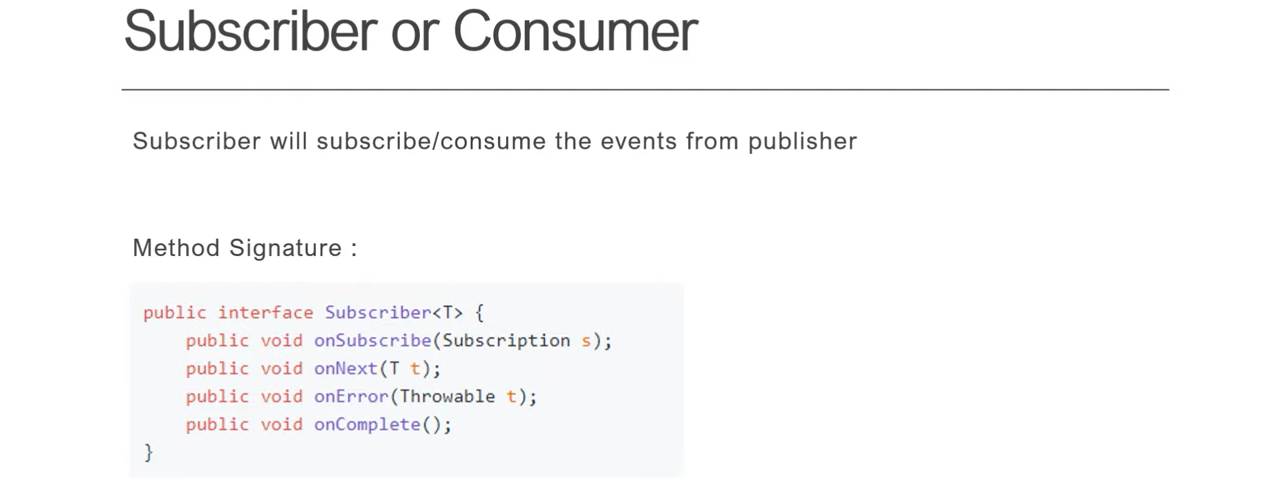
**---------------------------------------------------------------------------------------------------**

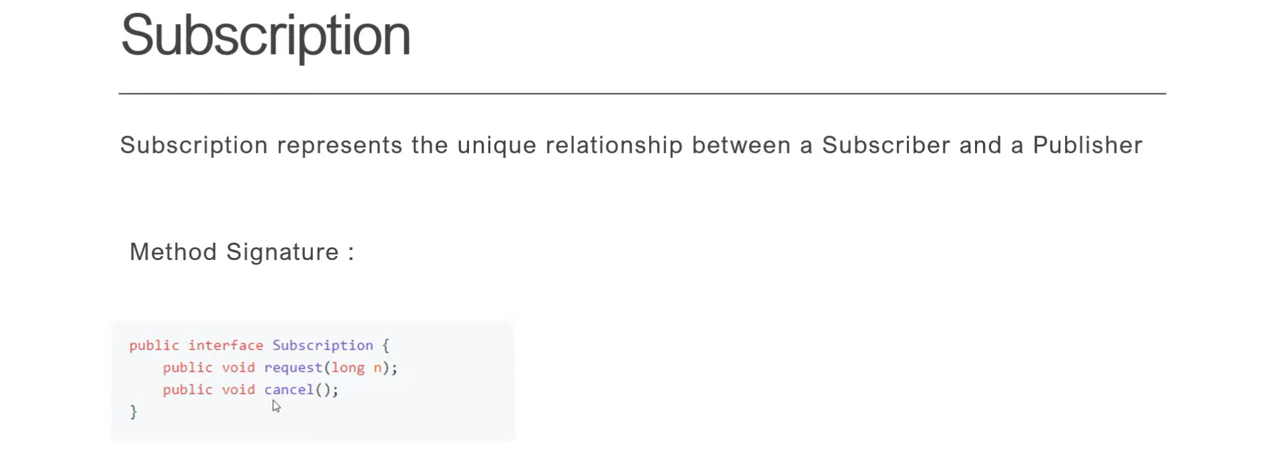


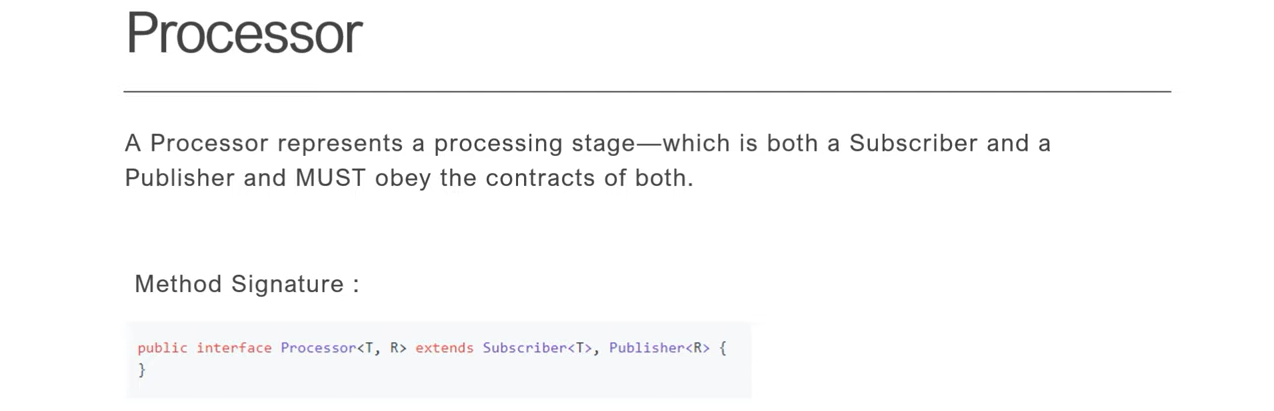
In this tutorial, let's deep dive into its specification or rules. Basically, reactive stream specification is nothing, it's just a rule to design a reactive programming. The way we have rule to design an object-oriented programming language, similar way we need to follow certain rule in reactive programming. Specification is very simple, it has only four interfaces, **publisher**, **subscriber**, **subscription**, and **processor**. These are the four interfaces which will be pillar of reactive stream specification or reactive programming. Each interface has couple of abstract methods.



1. The first interface is **Publisher** or you can pronounce it as a producer. **Publisher** is a data source who will always publish an event. If you observe in this publisher interface, we have only one method which is **subscribe**(). Now, subscriber need to call this subscribe method to register into publisher. We will understand this flow in our upcoming slide.
2. Now, the next interface is **Subscriber** or you can call it as a **Consumer**. Subscriber will subscribe or consume the event from publisher. If you observe the method signature, it has four abstract methods, **onsubscribe()**, **onnext()**, **onError()**, **oncomplete()**. If there is no error in your event processing, then you will find this oncomplete event. If there is any error, you will find this onerror event and in each data transfer from publisher to subscribe, you will find onnext event.
3. Let's assume your publisher is publishing 10 events, then you will find 10 onnext call.
4. Now, the next interface is **Subscription**. Subscription represents the unique relationship between a Subscriber and Publisher. So, if you observe the method signature, it contains two methods, **request()** and **cancel()**.
5. So, **Subscriber** will call this **request**() method to get the details from the publisher, to get the data from the publisher and there is also a **cancel()** method. So, you will understand all the methods in the workflow.
6. Next interface is **Processor**. A **Processor** represents a processing stage, which is both a Subscriber and a Publisher and it must obey the contract of both. So, if you observe the method signature, it extends from both **Subscriber** and **Publisher**. So, it should obey the contract of both publisher and subscriber. 

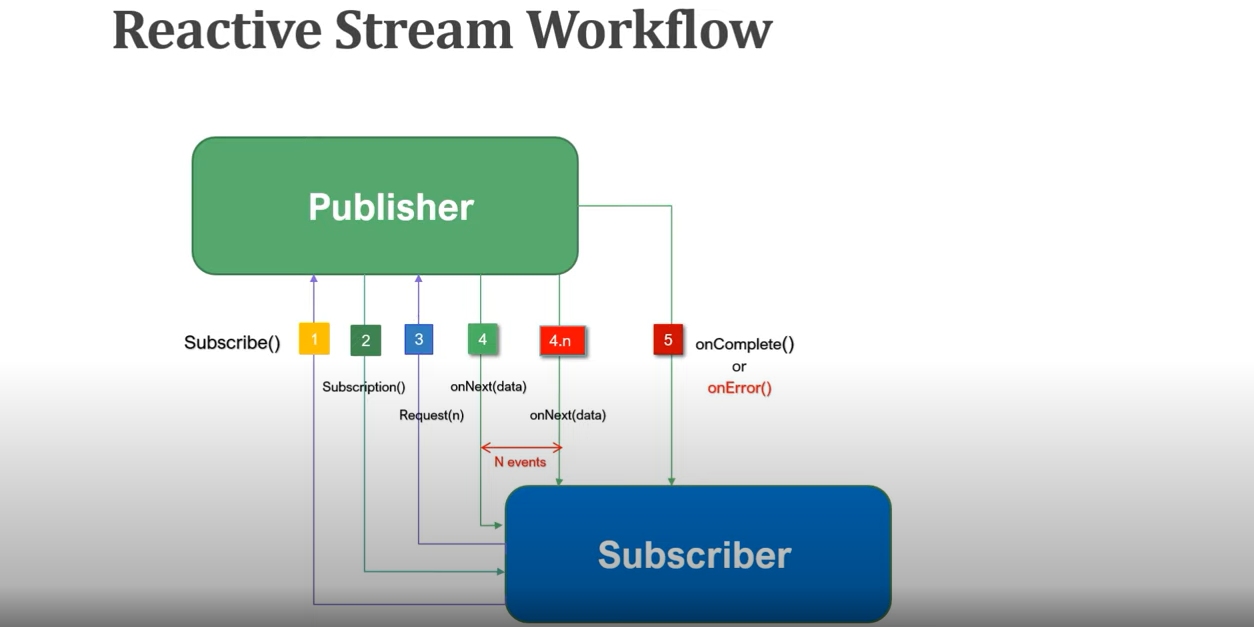






* Now, let's put all interfaces together and try to find out the workflow in reactive programming.

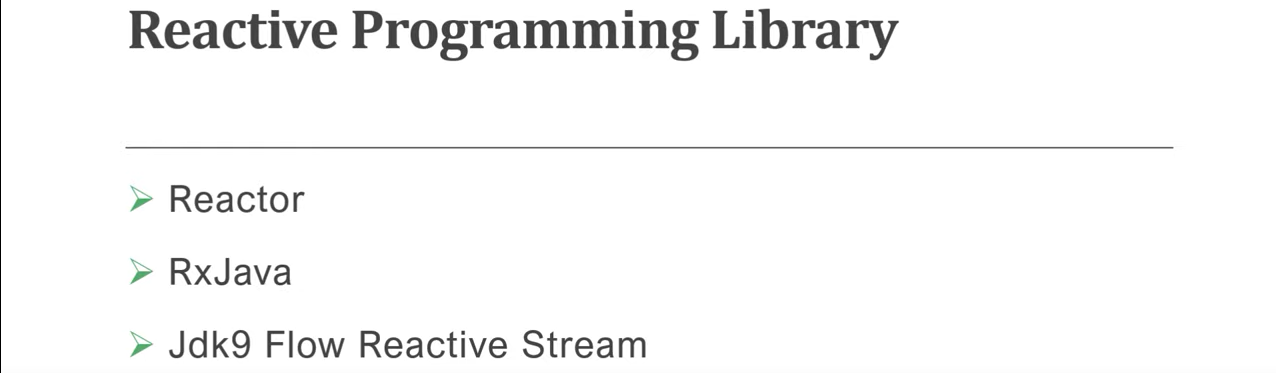
1. If you can observe, I have **Publisher** and **Subscriber**. Now, the first step, **Subscriber** will invoke **subscribe()** method of **Publisher** and pass the instance of subscriber instance as a input. Now, the next step, **Publisher** will going to send a subscription event to the Subscriber, confirming that your subscription is successful. Now, after that, Subscriber will call request() method from Subscription interface to get the data from Publisher.
2. So, if you observe the method signature is request n. n means **Subscriber** can request n number of data from Publisher. Now, next **Publisher** will send data stream to Subscriber by invoking **onNext()** method. But let's assume **Publisher** is returning 10 record. Then in that case, Publisher will fire 10 times **onnext()** event. If Publisher send n number of data, then there will be n time **onNext()** event process or it will execute **onNext()** event 10 times. Once all the record received by Subscriber, then publisher will invoke one onComplete() method of Subscriber to confirming, hey, Subscriber, I am done with my job. Your execution is successful.
3. If there is any error, then **Publisher** will fire **onError()** event. Also, there is an option for **Subscriber** to ask limited number of data from **Publisher**. let's say **Publisher** have 10 item and you want to fetch only 2 item. Then **Subscriber** can control that. If you can see this method signature request n. So, if I will pass your request 2, then I can fetch only 2 record from **Publisher**. That is what all about data backpressure in reactive stream programming. So far, we understand how this Publisher and Subscriber talk to each other through these are the interfaces.

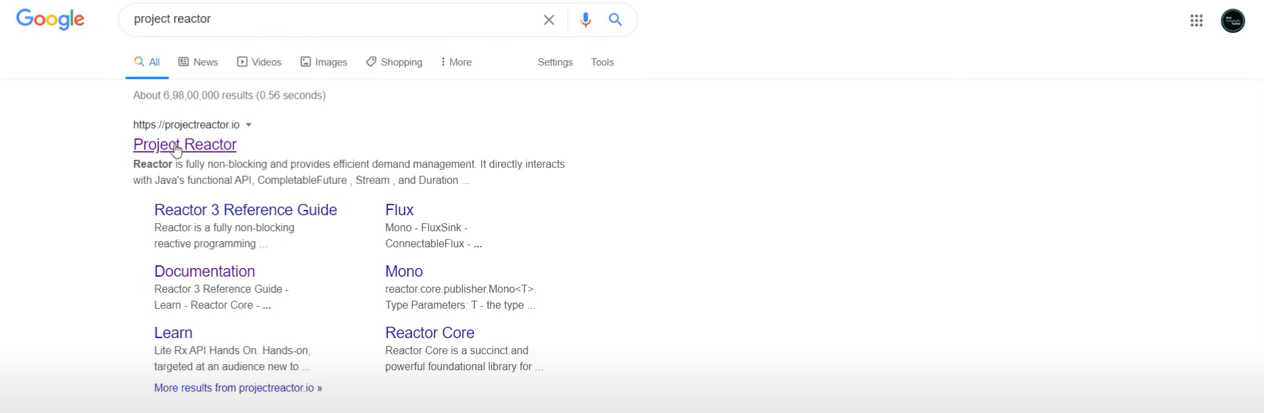


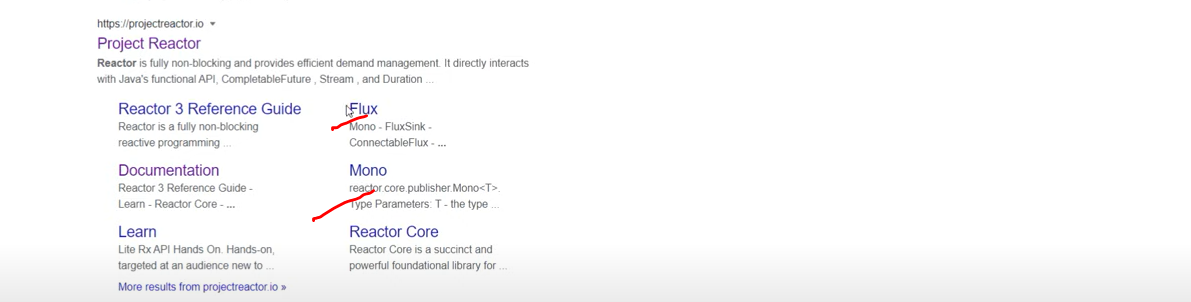
So, the first step, subscriber need to register to this publisher by call this subscribe method. Now, publisher need to send one subscription event to the subscriber. Then third step, subscriber need to get the data from publisher by requesting this request n method from subscription interface. Now, publisher will publish the data to subscriber by calling this onNext event method. Then as number of record or number of events that will be fired n times onNext event. Then on Completion, you will find onComplete event. If there is error, you will find onError event. Also, you understand how this backpressure support through this reactive programming. We understand what is reactive programming and what's its specification.

Now, let's understand how to build reactive programming. I mean, what all library or frameworks are available in market to design a reactive programming. So, if you can see this, there is three library through which we can design a reactive programming.

**Project Reactor**, **RxJava** and **ZDK9** itself provided one reactive stream implementation. In our entire series of course, we'll use **Project Reactor** because it is recommended library to work with Spring Boot framework. Let's do some quick review about this Project Reactor and its modules. So, if you go to the browser and if you type **Project Reactor**, if you go to the first link, then go inside documentation. So, in this particular course, we are going to cover **Reactor Core**, **Reactor Test** and **Reactor Netty**.







So, we understand what the modules are available in Project Reactor. Now, what are the data types we need to work with this reactive programming?

So, if you go to the search bar, you can see there is a two component, Flux and Mono. So, these are the two Project Reactor data type through which we can design our reactive programming and with this Flux and Mono, we need to play with Java 8 Stream API.