Spring Boot Reactive Programming   11-03-2025

Spring Boot Reactive Programming is based on **Project Reactor** and enables building **non-blocking, event-driven** applications using the **Reactive Streams** specification. It is designed to handle large volumes of data and I/O-bound tasks with better scalability and lower resource consumption.

🚀 Key Concepts of Reactive Programming

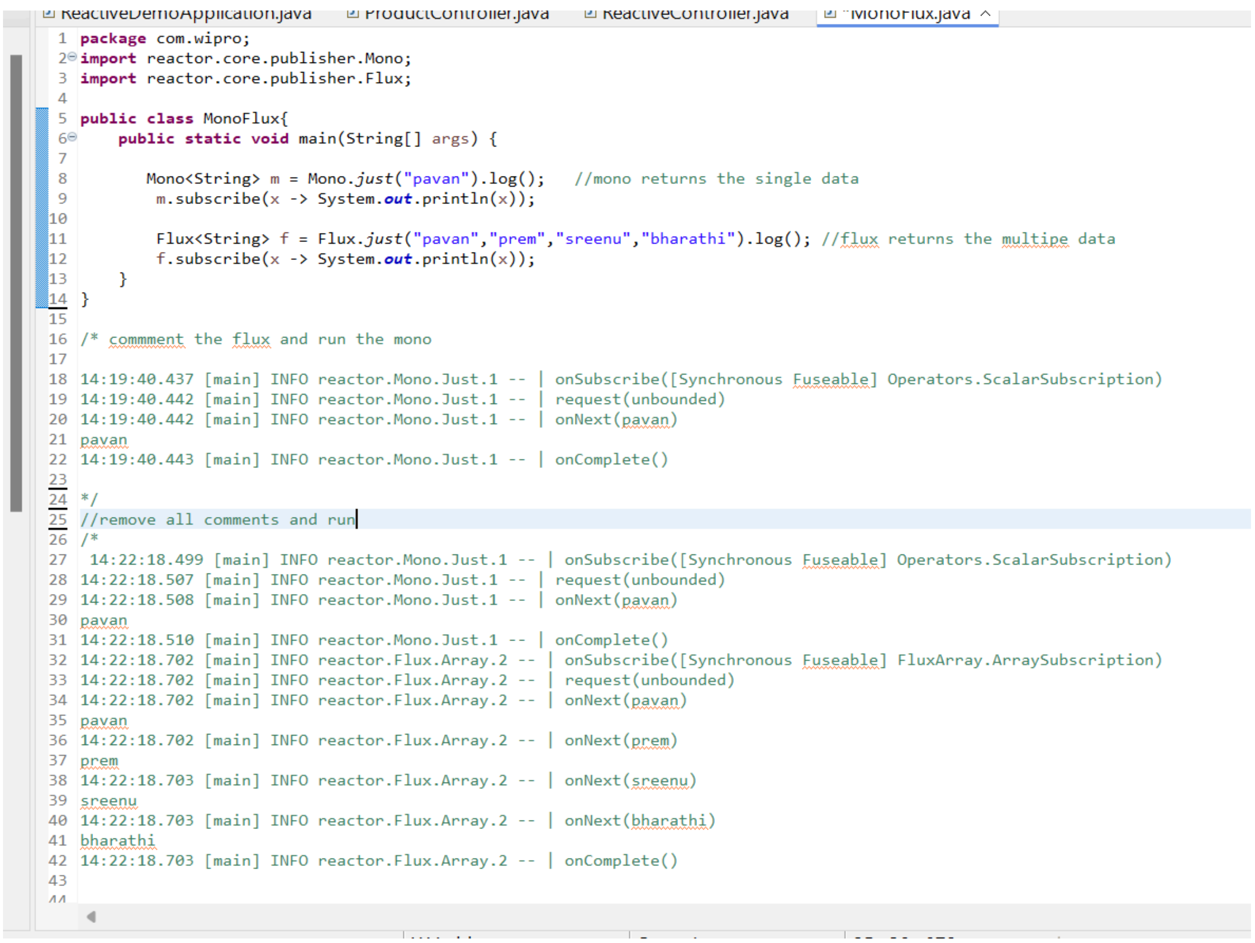
1. **Reactive Streams** – A specification for asynchronous stream processing with non-blocking backpressure.

2. **Publishers and Subscribers** – The core of reactive programming:

    - **Publisher** – Emits data.
    - **Subscriber** – Consumes data.
    - **Subscription** – Represents the connection between Publisher and Subscriber.
    - **Processor** – A hybrid of Publisher and Subscriber (used to transform data).
3. **Backpressure** – A mechanism to handle the situation where a subscriber cannot keep up with the publisher's data rate.

📚 Reactive Libraries in Spring Boot

- **Project Reactor** – Core library for reactive programming in Spring.
    - `Mono` – Represents 0 or 1 element.
    - `Flux` – Represents 0 to N elements (like a stream).



```java
package com.wipro;
import reactor.core.publisher.Mono;
import reactor.core.publisher.Flux;

public class MonoFlux{
    public static void main(String[] args) {

        Mono<String> m = Mono.just("pavan").log();   //mono returns the single data
         m.subscribe(x -> System.out.println(x));

         Flux<String> f = Flux.just("pavan","prem","sreenu","bharathi").log(); //flux returns the multipe data
         f.subscribe(x -> System.out.println(x));
    }
}

/* commment the flux and run the mono

14:19:40.437 [main] INFO reactor.Mono.Just.1 -- | onSubscribe([Synchronous Fuseable] Operators.ScalarSubscription)
14:19:40.442 [main] INFO reactor.Mono.Just.1 -- | request(unbounded)
14:19:40.442 [main] INFO reactor.Mono.Just.1 -- | onNext(pavan)
pavan
14:19:40.443 [main] INFO reactor.Mono.Just.1 -- | onComplete()

*/
//remove all comments and run
/*
 14:22:18.499 [main] INFO reactor.Mono.Just.1 -- | onSubscribe([Synchronous Fuseable] Operators.ScalarSubscription)
14:22:18.507 [main] INFO reactor.Mono.Just.1 -- | request(unbounded)
14:22:18.508 [main] INFO reactor.Mono.Just.1 -- | onNext(pavan)
pavan
14:22:18.510 [main] INFO reactor.Mono.Just.1 -- | onComplete()
14:22:18.702 [main] INFO reactor.Flux.Array.2 -- | onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
14:22:18.702 [main] INFO reactor.Flux.Array.2 -- | request(unbounded)
14:22:18.702 [main] INFO reactor.Flux.Array.2 -- | onNext(pavan)
pavan
14:22:18.702 [main] INFO reactor.Flux.Array.2 -- | onNext(prem)
prem
14:22:18.703 [main] INFO reactor.Flux.Array.2 -- | onNext(sreenu)
sreenu
14:22:18.703 [main] INFO reactor.Flux.Array.2 -- | onNext(bharathi)
bharathi
14:22:18.703 [main] INFO reactor.Flux.Array.2 -- | onComplete()
```

- **WebFlux** – Reactive alternative to Spring MVC, built on top of Project Reactor.
- **R2DBC** – Reactive SQL database client.
- **Spring Data MongoDB** – Reactive MongoDB client.
- **Spring Security** – Supports reactive programming for authentication and authorization

Spring Boot Reactive Programming - Explained Simply

What is Spring Boot Reactive Programming?

Imagine you run a restaurant. Traditional programming (blocking) is like having only one waiter who serves one customer at a time. If a customer takes time to decide their order, the waiter just stands there waiting, unable to serve others.

Now, **Reactive Programming** is like having a smart waiter who can handle multiple customers at once. If one customer is thinking, the waiter moves on to take another order and comes back later. This makes the restaurant run more efficiently.

In technical terms, **Spring Boot Reactive Programming** allows applications to handle multiple tasks efficiently by **not blocking** resources while waiting for slow operations (like database queries, API calls, etc.). It is built on **Project Reactor** and uses concepts like **Mono** and **Flux** to process data streams asynchronously.

Advantages of Spring Boot Reactive Programming

1. **Better Performance** 🏎️
   - It can handle more users with fewer system resources, making apps faster and more scalable.
2. **Efficient Resource Utilization** 🔄
   - No threads are wasted waiting for slow responses. The system continues working on other tasks.
3. **Handles High Traffic Easily** 🚦
   - Ideal for applications that deal with real-time data or heavy loads (e.g., chat applications, stock market updates, streaming services).
4. **Non-blocking Nature** ⏳
   - Unlike traditional approaches where a request waits for a response, in reactive programming, tasks run in the background while the system continues handling other requests.

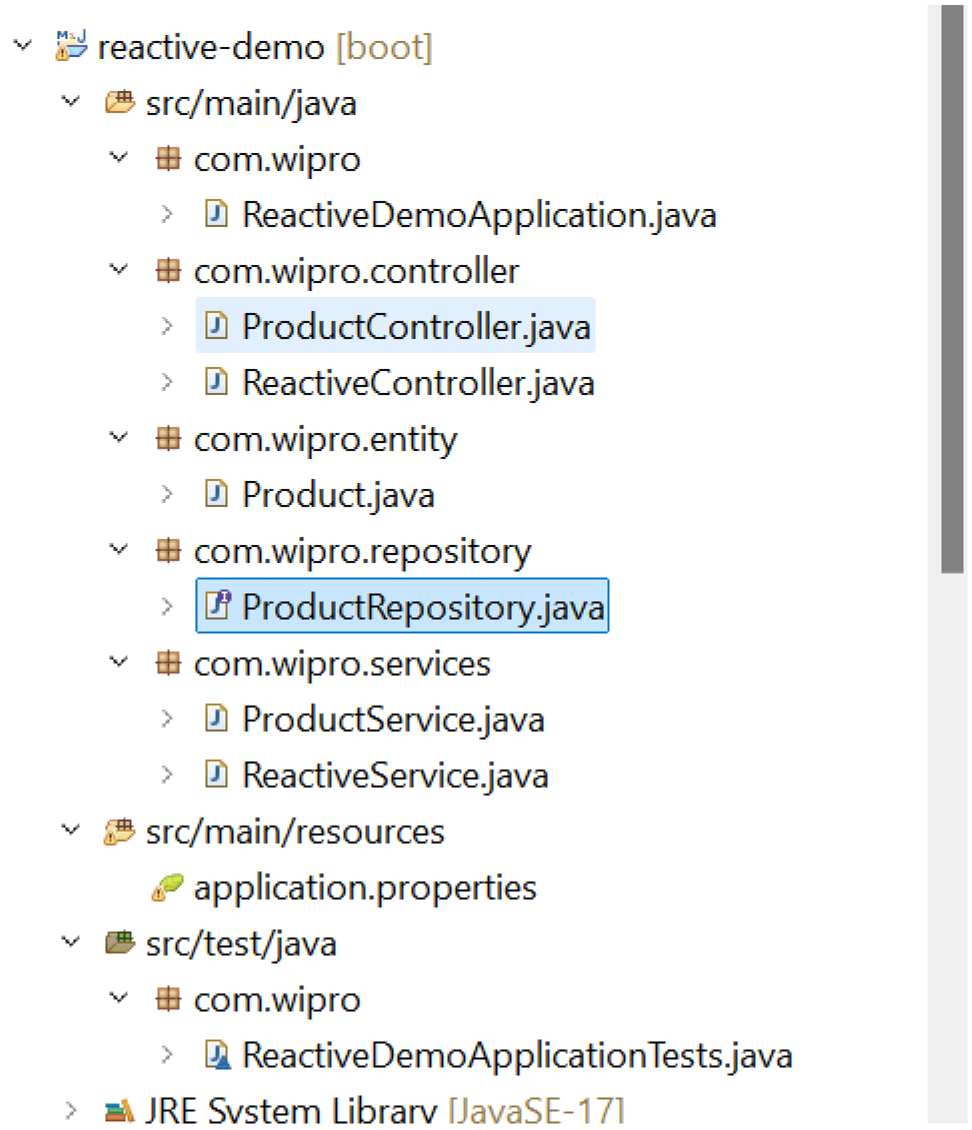
Why Use It?

- If your application deals with **large-scale, real-time data processing** (e.g., Netflix, Facebook feeds, IoT applications).
- If you want to improve **scalability and efficiency** without increasing hardware costs.
- If your app makes **a lot of database/API calls** and you want to avoid waiting time.

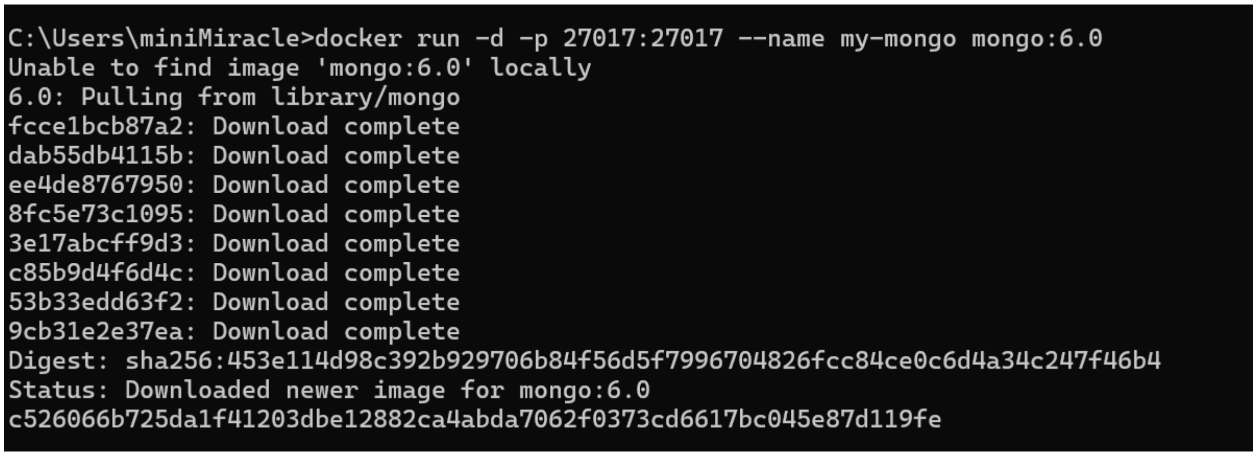
Conclusion

Spring Boot Reactive Programming helps build fast, scalable, and efficient applications by allowing tasks to execute **asynchronously and non-blockingly**. If your application needs to handle many concurrent users and real-time updates, it's a great choice!

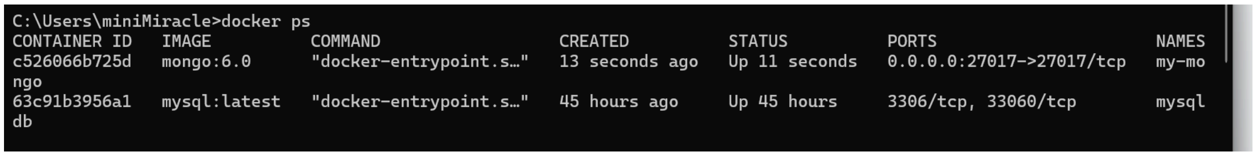
https://github.com/PavanKalyan96Dev/ReactiveProgramming-Demo



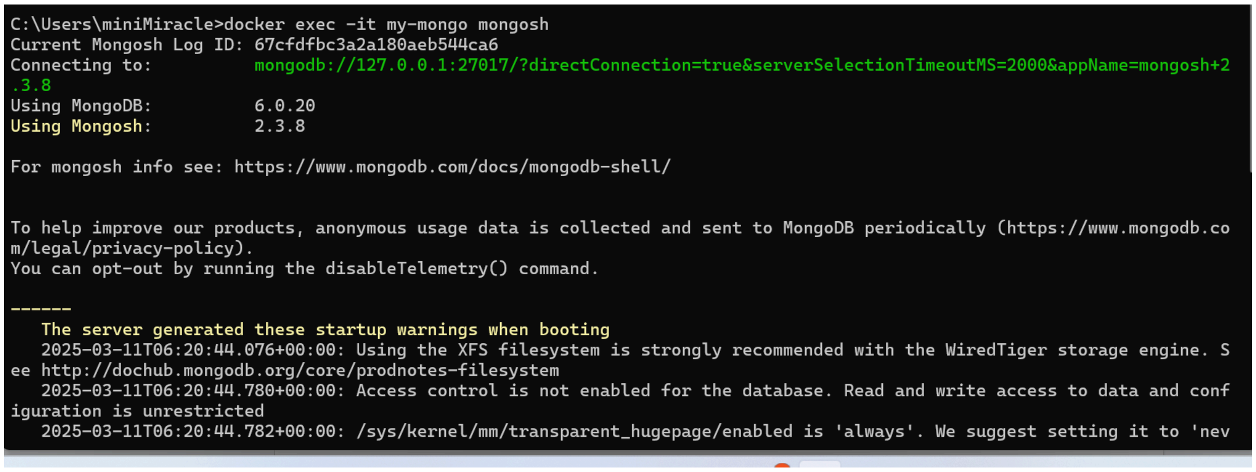
```
reactive-demo [boot]
  src/main/java
    com.wipro
      ReactiveDemoApplication.java
    com.wipro.controller
      ProductController.java
      ReactiveController.java
    com.wipro.entity
      Product.java
    com.wipro.repository
      ProductRepository.java
    com.wipro.services
      ProductService.java
      ReactiveService.java
  src/main/resources
    application.properties
  src/test/java
    com.wipro
      ReactiveDemoApplicationTests.java
  JRE System Library [JavaSE-17]
```



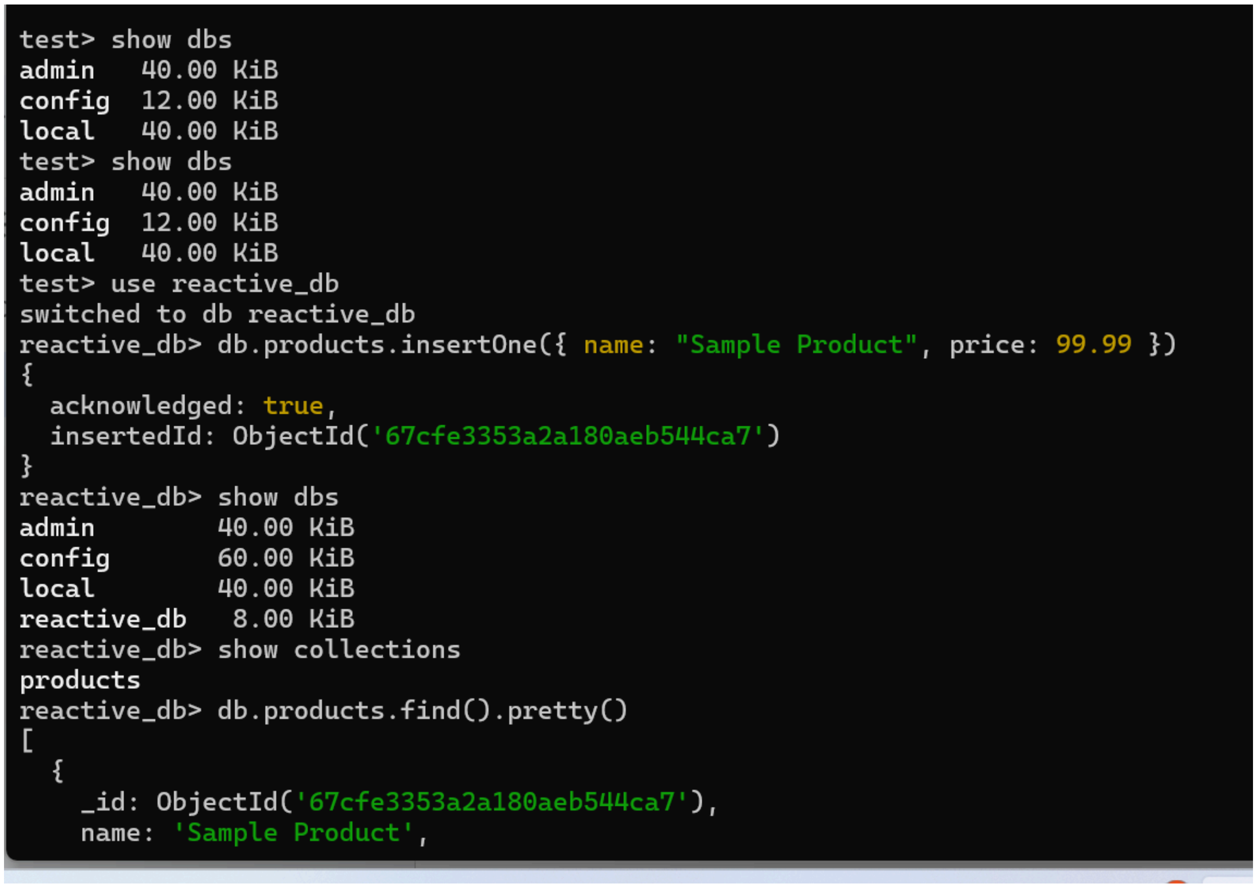
```
C:\Users\miniMiracle>docker run -d -p 27017:27017 --name my-mongo mongo:6.0
Unable to find image 'mongo:6.0' locally
6.0: Pulling from library/mongo
fcce1bcb87a2: Download complete
dab55db4115b: Download complete
ee4de8767950: Download complete
8fc5e73c1095: Download complete
3e17abcff9d3: Download complete
c85b9d4f6d4c: Download complete
53b33edd63f2: Download complete
9cb31e2e37ea: Download complete
Digest: sha256:453e114d98c392b929706b84f56d5f7996704826fcc84ce0c6d4a34c247f46b4
Status: Downloaded newer image for mongo:6.0
c526066b725da1f41203dbe12882ca4abda7062f0373cd6617bc045e87d119fe
```



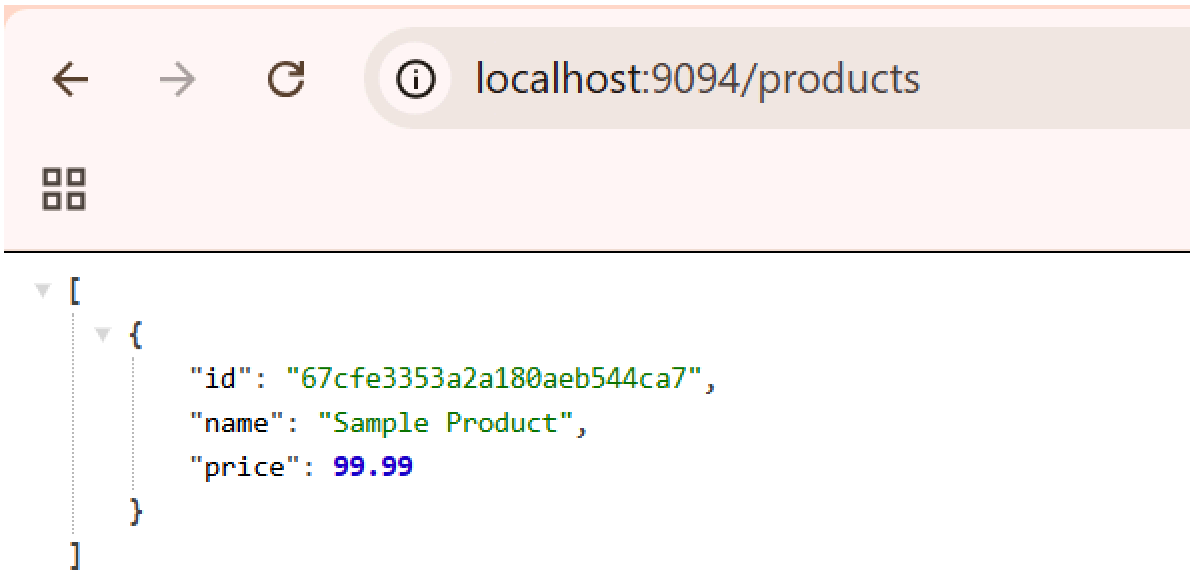
```
C:\Users\miniMiracle>docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS                      NAMES
c526066b725d   mongo:6.0      "docker-entrypoint.s…"   13 seconds ago   Up 11 seconds   0.0.0.0:27017->27017/tcp   my-mo
ngo
63c91b3956a1   mysql:latest   "docker-entrypoint.s…"   45 hours ago     Up 45 hours     3306/tcp, 33060/tcp        mysql
db
```



```
C:\Users\miniMiracle>docker exec -it my-mongo mongosh
Current Mongosh Log ID: 67cfdfbc3a2a180aeb544ca6
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2
.3.8
Using MongoDB:          6.0.20
Using Mongosh:          2.3.8

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/


To help improve our products, anonymous usage data is collected and sent to MongoDB periodically (https://www.mongodb.co
m/legal/privacy-policy).
You can opt-out by running the disableTelemetry() command.

------
   The server generated these startup warnings when booting
   2025-03-11T06:20:44.076+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. S
ee http://dochub.mongodb.org/core/prodnotes-filesystem
   2025-03-11T06:20:44.780+00:00: Access control is not enabled for the database. Read and write access to data and conf
iguration is unrestricted
   2025-03-11T06:20:44.782+00:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'nev
```

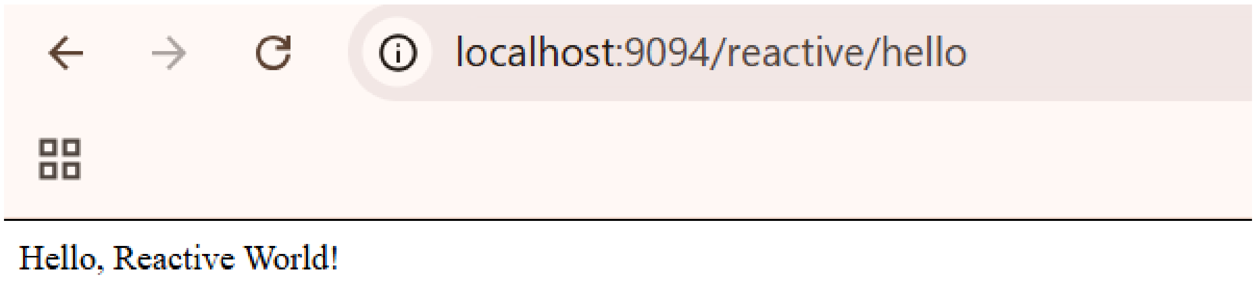


```
test> show dbs
admin   40.00 KiB
config  12.00 KiB
local   40.00 KiB
test> show dbs
admin   40.00 KiB
config  12.00 KiB
local   40.00 KiB
test> use reactive_db
switched to db reactive_db
reactive_db> db.products.insertOne({ name: "Sample Product", price: 99.99 })
{
  acknowledged: true,
  insertedId: ObjectId('67cfe3353a2a180aeb544ca7')
}
reactive_db> show dbs
admin        40.00 KiB
config       60.00 KiB
local        40.00 KiB
reactive_db   8.00 KiB
reactive_db> show collections
products
reactive_db> db.products.find().pretty()
[
  {
    _id: ObjectId('67cfe3353a2a180aeb544ca7'),
    name: 'Sample Product',
```

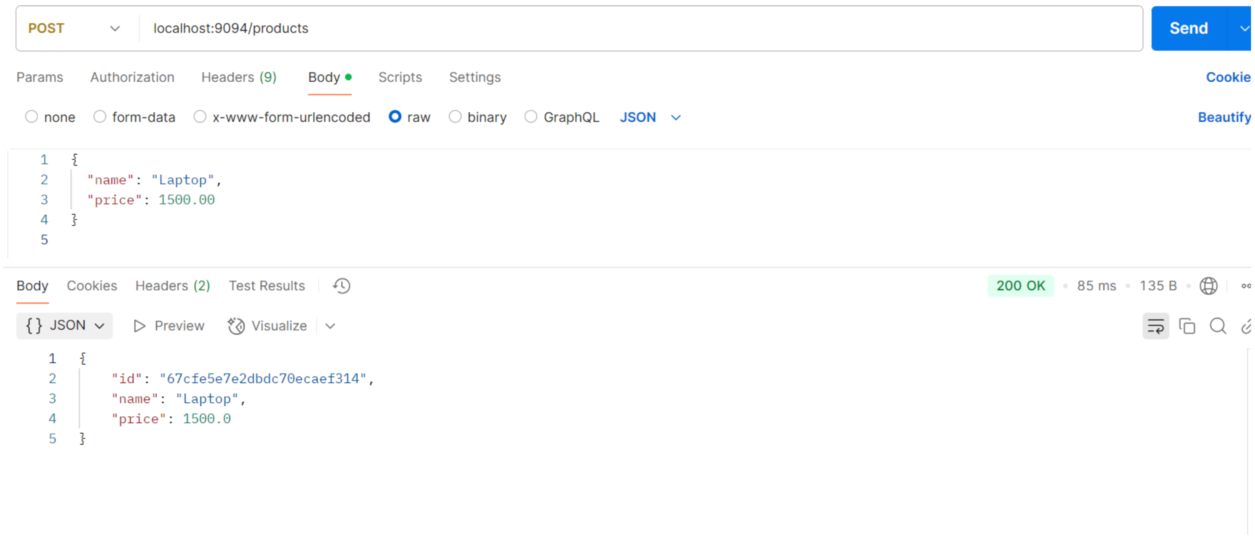


```
localhost:9094/products
```

```
[
  {
    "id": "67cfe3353a2a180aeb544ca7",
    "name": "Sample Product",
    "price": 99.99
  }
]
```



Hello, Reactive World!

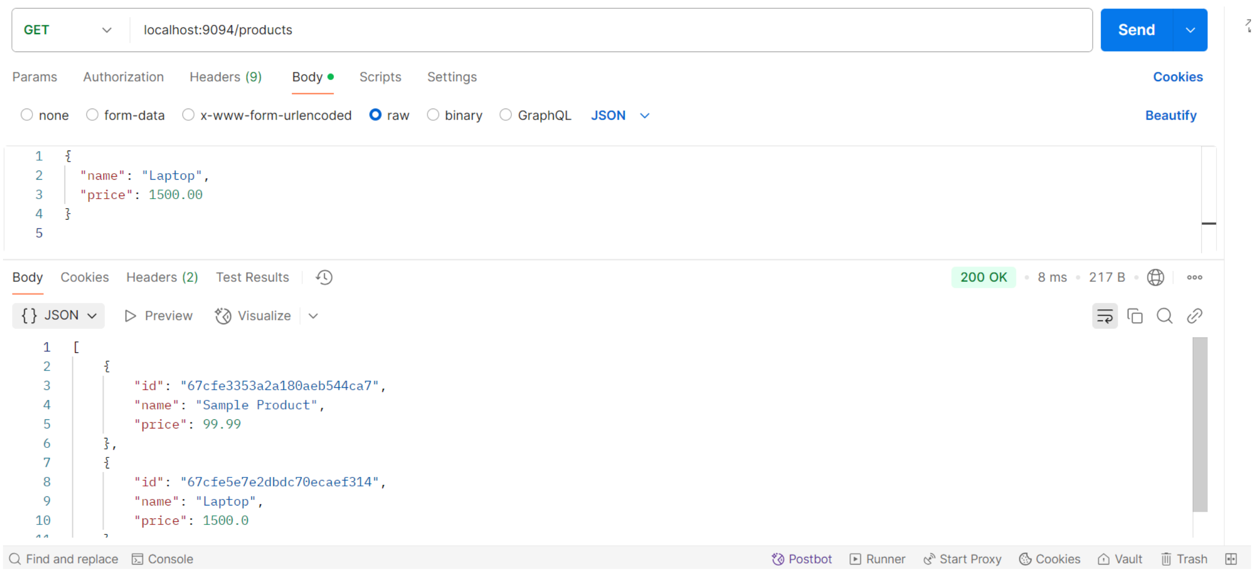


```
{
  "name": "Laptop",
  "price": 1500.00
}
```

```
{
    "id": "67cfe5e7e2dbdc70ecaef314",
    "name": "Laptop",
    "price": 1500.0
}
```

Adding the data in postman

Now we'll get them


GET localhost:9094/products

Params Authorization Headers (9) Body Scripts Settings Cookies

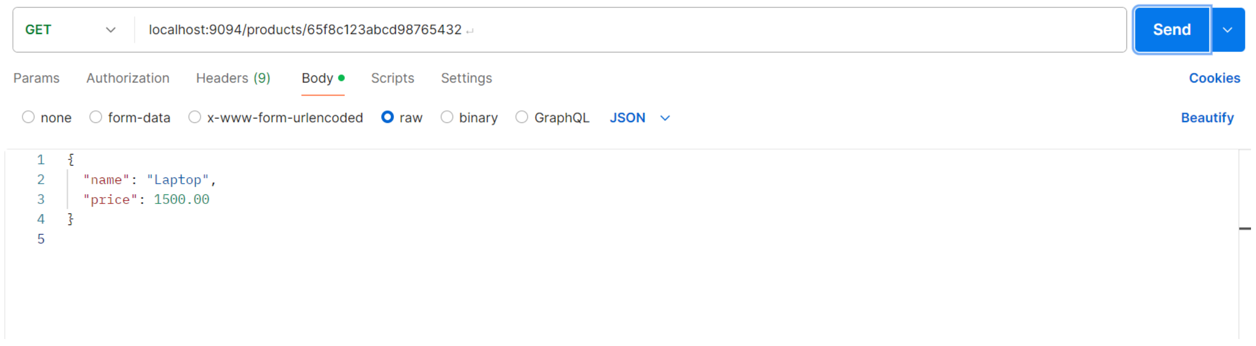
none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON Beautify

```
{
  "name": "Laptop",
  "price": 1500.00
}
```

Body Cookies Headers (2) Test Results 200 OK 8 ms 217 B

```
[
    {
        "id": "67cfe3353a2a180aeb544ca7",
        "name": "Sample Product",
        "price": 99.99
    },
    {
        "id": "67cfe5e7e2dbdc70ecaef314",
        "name": "Laptop",
        "price": 1500.0
```

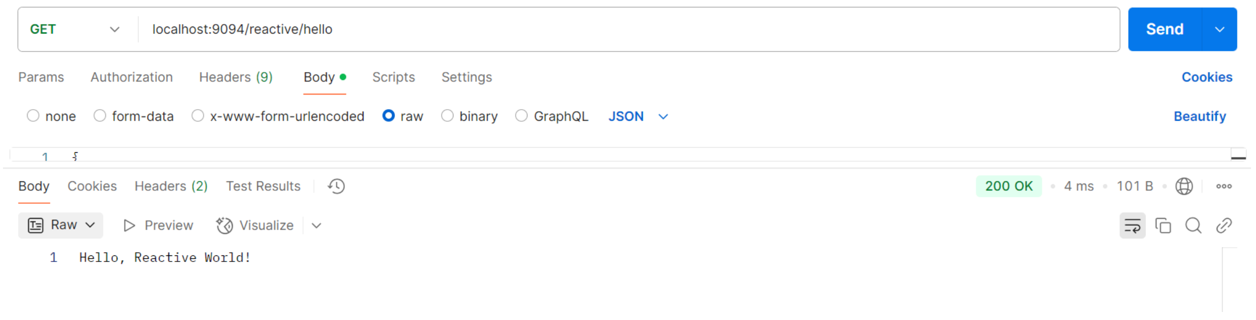
Get product by id


GET localhost:9094/products/65f8c123abcd98765432

Params Authorization Headers (9) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON Beautify

```
{
  "name": "Laptop",
  "price": 1500.00
}
```


GET localhost:9094/reactive/hello

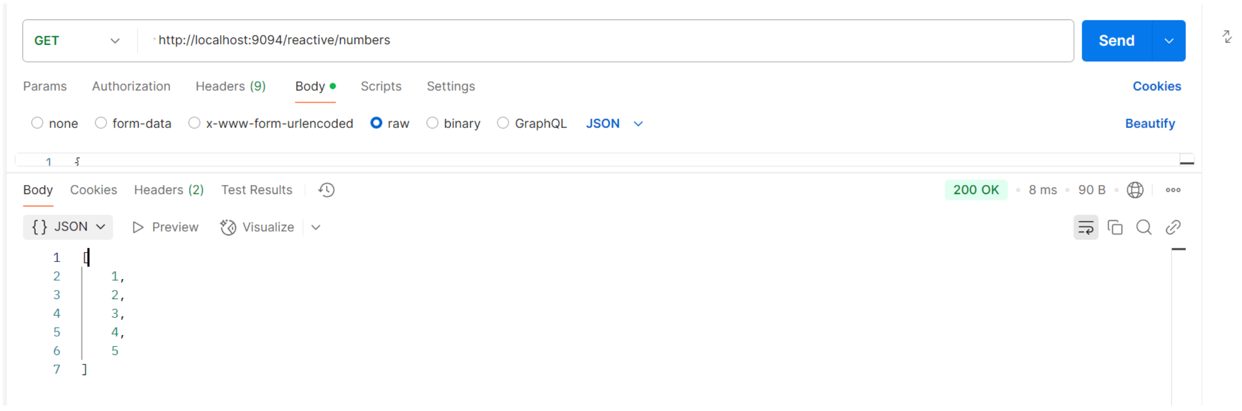
Params Authorization Headers (9) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON Beautify

Body Cookies Headers (2) Test Results 200 OK 4 ms 101 B

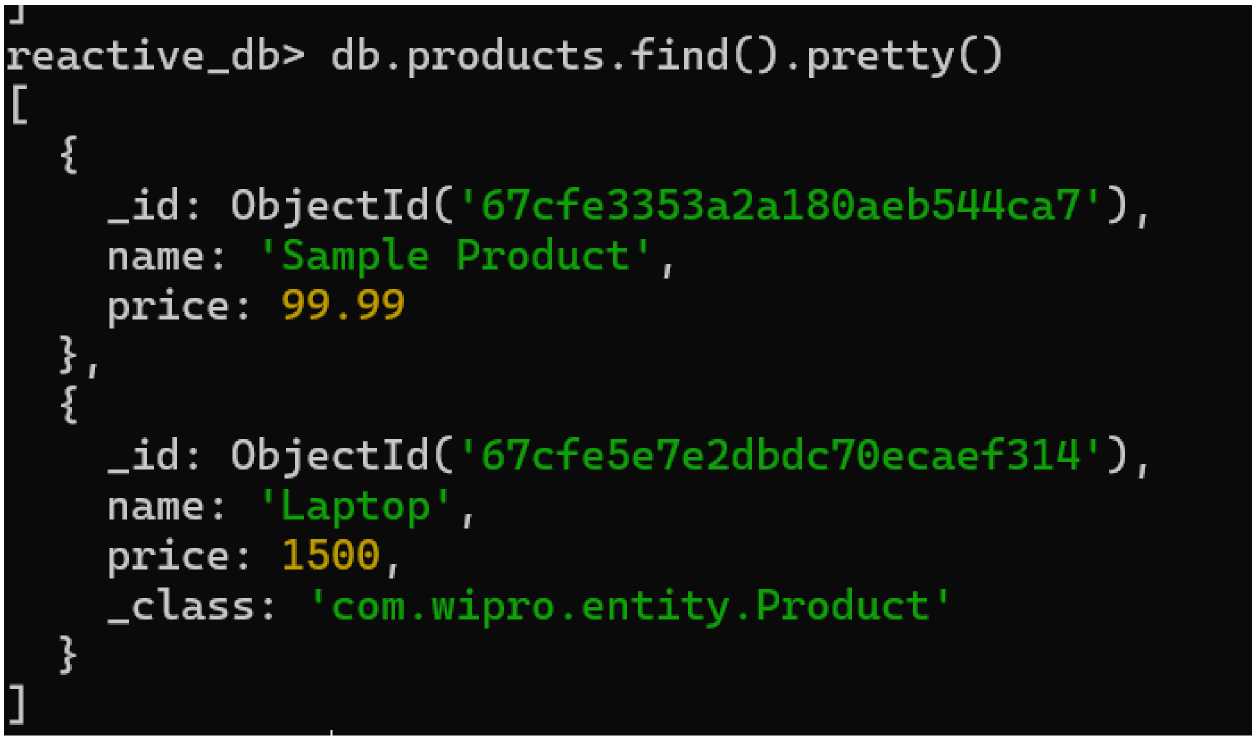
```
Hello, Reactive World!
```

Get the stream of numbers



```
http://localhost:9094/reactive/numbers
```

```
[
    1,
    2,
    3,
    4,
    5
]
```



```
reactive_db> db.products.find().pretty()
[
  {
    _id: ObjectId('67cfe3353a2a180aeb544ca7'),
    name: 'Sample Product',
    price: 99.99
  },
  {
    _id: ObjectId('67cfe5e7e2dbdc70ecaef314'),
    name: 'Laptop',
    price: 1500,
    _class: 'com.wipro.entity.Product'
  }
]
```

https://chatgpt.com/share/67d02277-e76c-8011-a33d-6af5a0f0c2aa

In **Project Reactor**, which is the reactive programming library used in **Spring WebFlux**, `Mono` and `Flux` are two core reactive types.

1. Mono<T>

- Represents **zero or one** element.
- Used for cases where you expect at most one result.
- Example use case: Fetching a single user from the database.

Example:

java

```java
Mono<String> monoExample = Mono.just("Hello, Mono!");
monoExample.subscribe(System.out::println);
```

Output:

```
Hello, Mono!
```

2. Flux<T>

- Represents **zero or more** elements (a stream of data).
- Used for cases where multiple results are expected.
- Example use case: Fetching all employees from a database.

Example:

```java
Flux<String> fluxExample = Flux.just("Java", "Spring", "WebFlux");
fluxExample.subscribe(System.out::println);
```

Output:

```nginx
Java
Spring
WebFlux
```

Key Differences

Feature	Mono	Flux
Number of elements	0 or 1	0 to N
Used for	Single item or empty result	Multiple items (stream)
Example	`Mono.just("Hello")`	`Flux.just("A", "B", "C")`

what is the beneift of using reactive programming when you compare with traditional programming?

Reactive programming offers several benefits over traditional (imperative) programming, especially in high-throughput and event-driven applications. Below are the key advantages:

---

1. Non-Blocking & Asynchronous Execution

Traditional: Uses blocking I/O operations, meaning each request waits until the previous one completes.
Reactive: Uses **non-blocking I/O**, allowing the system to handle multiple operations simultaneously.

🔹 **Example**: In a traditional Spring MVC app, a request to fetch user data would block the thread until the database responds. In WebFlux, the request is non-blocking, allowing the thread to handle other requests in the meantime.

2. Better Resource Utilization (Fewer Threads)

Traditional: Creates new threads for each request, leading to high memory usage and thread exhaustion.
Reactive: Uses **event loops** and a small thread pool (Netty event loop), making it efficient in handling thousands of concurrent users.

🔹 **Example**: A traditional Tomcat server uses one thread per request, whereas a reactive Netty-based server can handle multiple requests using fewer threads.

---

3. High Performance & Scalability

Traditional: Struggles with scaling under heavy load due to thread blocking.
Reactive: Handles high loads efficiently because non-blocking operations free up threads.

🔹 **Example**: Suitable for applications that deal with **real-time data streaming**, such as stock price updates, chat applications, or IoT.