Reactive Programming in Spring Boot:

Thread Per Request Model:

- In server Thread Pool will be available to handle incoming requests.
- For every request one thread will be used to handle that.
- In this model, each incoming HTTP request is handled by a dedicated thread from a thread pool.
- The thread is responsible for processing the request, including invoking the appropriate controller logic, and generating the response. Once the response is sent back to the client, the thread becomes available for handling another request.
- Until The request processing got completed our thread will be blocked.
- In this approach waiting period will be increased.

Here's how the Thread Per Request model works:

1. **Request Arrival:** When a client sends an HTTP request to the server, the server's container (e.g., Tomcat, Jetty) receives the request.
2. **Thread Assignment:** The container assigns a thread from its thread pool to handle the incoming request.
3. **Request Processing:** The assigned thread processes the request, invoking the necessary controller logic, service methods, or database access.
4. **Response Generation:** After processing the request, the thread generates an HTTP response.
5. **Thread Release:** Once the response is sent back to the client, the thread is returned to the thread pool, where it can be reused for handling another incoming request.

Note: To overcome this problem, we are using Reactive Programming.

Reactive Programming:

- Reactive programming in Spring Boot refers to the use of the Spring Framework's reactive programming model to build asynchronous and non-blocking applications.
- In this approach our threads will not be blocked.
- Reactive Programming will work based on Event driven approach.
- Reactive Programming Uses Back Pressure mechanism to ensure producers don't overburden consumers.
- In a reactive system, Back Pressure mechanism ensures that a slower consumer can signal a faster producer to slow down the rate of data emission.
- Using reactive programming we can process multiple requests asynchronously with non-blocking technique.
- Reactive programming often leverages functional programming concepts and constructs, such as higher-order functions, immutability, and declarative programming styles.
- Functional programming enables concise and expressive code for handling data transformations and stream processing.

Reactive Programming Features:

1. **New Programming paradigm**
   - ✓ Reactive programming introduces a paradigm shift from the traditional imperative programming model to a more event-driven, declarative, asynchronous approach to handling data streams.
2. **Asynchronous & non-blocking**
   - ✓ Reactive programming emphasizes asynchronous and non-blocking operations, allowing applications to perform tasks execution concurrently without blocking threads.
   - ✓ This ensures that applications remain responsive even under heavy loads.
3. **Functional Style of code**
   - ✓ Reactive programming often adopts a functional programming style, leveraging concepts such as higher-order functions, immutability, and pure functions.
   - ✓ Functional programming facilitates concise, composable, and declarative code, making it easier to work with asynchronous data streams and transformations.
4. **Data Flow as event stream**
   - ✓ In reactive programming, data flows as a stream of events or messages, rather than being processed in a batch-oriented manner.
   - ✓ Applications react to incoming events, such as user actions, sensor readings, or messages from other components, and perform actions based on these events.
   - ✓ This event-driven approach enables real-time processing and responsiveness.
5. **Back Pressure**
   - ✓ Back pressure is a fundamental concept in reactive programming that addresses the issue of overwhelming downstream components with more data than they can handle.
   - ✓ Reactive systems incorporate back pressure mechanisms to allow downstream consumers to control the rate of data consumption from upstream producers, preventing overload and ensuring smooth processing.

Spring Boot Web Flux:

- Spring web flux is a reactive programming model introduced in spring 5.x version.
- Web Flux supports asynchronous and non-blocking event driven architecture for building web application.
- It enables developers to build scalable application which can handle loads of traffic without compromising on performance.
- Spring Web Flux will provide netty as default embedded container.

Reactive Programming Components:

- Reactive Programming works based on publisher and subscriber model.
- In Reactive programming mainly we have below 2 publishers,

1. **Mono:** It can produce only one value at a time (0 or 1)
2. **Flux:** It can produce zero or more values (0 to N)

Add Spring Web Flux Maven Dependency,

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

```
@Test
public void testMonoReactive() {
        Mono<String> mono = Mono.just("Reactive Programming").log();
        mono.subscribe(System.out::println);
}

@Test
public void testFluxReactive() {
        Flux<String> flux = Flux.just("Hello, ", "Reactive Programming").log();
        flux.subscribe(System.out::println);
}
```

Reactive Programming working based on Events:

a. **onSubscribe( ):**
   - ✓ This method is called when a subscriber subscribes to a publisher.
   - ✓ It provides the subscription object, which the subscriber can use to request data from the publisher and manage back pressure.

b. **request( ):**
   - ✓ After subscribing, the subscriber can call the request() method on the subscription object to request a certain number of data items from the publisher.
   - ✓ This method signals demand from the subscriber to the publisher, allowing the publisher to emit data accordingly.

c. **onNext( ):**
   - ✓ This method is called by the publisher when it emits a new data item to the subscriber.
   - ✓ It delivers the next element in the stream to the subscriber, allowing the subscriber to process or react to the incoming data.

d. **onComplete( ):**
   - ✓ The onComplete() method is called by the publisher when it has successfully completed emitting all the data items in the stream.
   - ✓ It signals to the subscriber that the stream has ended successfully, allowing the subscriber to perform any necessary cleanup or finalization tasks.

e. **onError( ):**
   - ✓ If an error occurs during the processing of the stream, the publisher calls the onError() method and provides the error object to the subscriber.
   - ✓ This method signals to the subscriber that an error has occurred, allowing the subscriber to handle the error gracefully, perform error recovery, or propagate the error further downstream.