Spring WebFlux Tutorial

By Lokesh Gupta | Filed Under: Spring WebFlux

The reactive-stack web framework, **Spring WebFlux**, has been added Spring 5.0. It is fully non-blocking, supports reactive streams back pressure, and runs on such servers as Netty, Undertow, and Servlet 3.1+ containers. In this **spring webflux tutorial**, we will learn the basic concepts behind reactive programming, webflux apis and a fully functional hello world example.

1. Reactive Programming

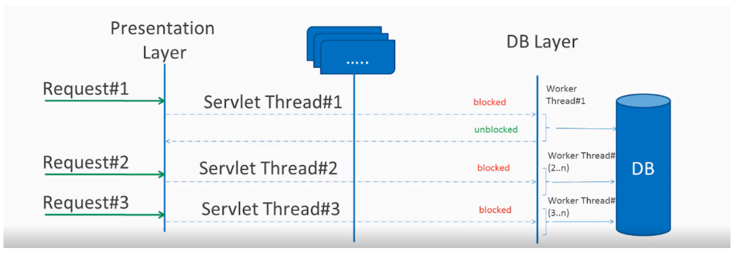
Reactive programming is a programming paradigm that promotes an asynchronous, non-blocking, event-driven approach to data processing. Reactive programming involves modeling data and events as observable data streams and implementing data processing routines to react to the changes in those streams.

Before digging deeper into reactive world, first understand the difference between blocking vs non-blocking request processing.

1.1. Blocking vs non-blocking (async) request processing

1.1.1. Blocking request processing

In traditional MVC applications, when a request come to server, a servlet thread is created. It delegates the request to worker threads for I/O operations such as database access etc. During the time worker threads are busy, servlet thread (request thread) remain in waiting status and thus it is blocked. It is also called **synchronous request processing**.



Blocking request processing

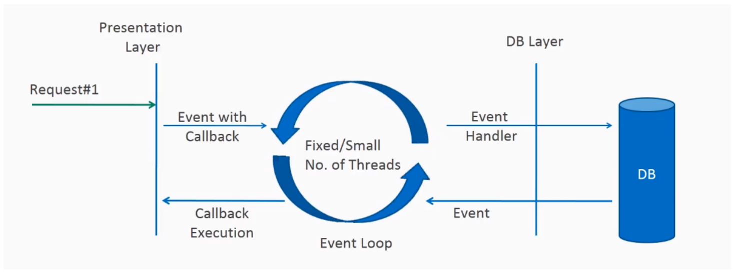
As server can have some finite number of request threads, it limits the server capability to process that number of requests at maximum server load. It may hamper the performance and limit the full utilization of server capability.

1.1.2. Non-blocking request processing

In non-blocking or asynchronous request processing, no thread is in waiting state. There is generally only one request thread receiving the request.

All incoming requests come with a event handler and call back information. Request thread delegates the incoming requests to a thread pool (generally small number of threads) which delegate the request to it’s handler function and immediately start processing other incoming requests from request thread.

When the handler function is complete, one of thread from pool collect the response and pass it to the call back function.



Non-blocking request processing

Non-blocking nature of threads helps in scaling the performance of the application. Small number of threads means less memory utilization and also less context switching as well.

1.2. What is reactive programming?

The term, “reactive,” refers to programming models that are built around reacting to changes. It is build around publisher-subscriber pattern (observer pattern). In reactive style of programming, we make a request for resource and start performing other things. When the data is available, we get the notification along with data inform of call back function. In callback function, we handle the response as per application/user needs.

One important thing to remember is back pressure. In non-blocking code, it becomes important to **control the rate of events** so that a fast producer does not overwhelm its destination.

Reactive web programming is great for applications that have streaming data, and clients that consume it and stream it to their users. It is not great for developing traditional CRUD applications. If you’re developing the next *Facebook* or *Twitter* with lots of data, a reactive API might be just what you’re looking for.

2. Reactive Streams API

The new Reactive Streams API was created by engineers from Netflix, Pivotal, Lightbend, RedHat, Twitter, and Oracle, among others and is now part of Java 9. It defines four interfaces:

- Publisher: Emits a sequence of events to subscribers according to the demand received from its subscribers. A publisher can serve multiple subscribers.

  It has a single method:

  ```
  Publisher.java

  public interface Publisher<T>
  {
      public void subscribe(Subscriber<? super
  }
  ```

- Subscriber: Receives and processes events emitted by a Publisher. Please note that no notifications will be received until `Subscription#request(long)` is called to signal the demand.

It has four methods to handle various kind of responses received.

Subscriber.java

```
public interface Subscriber<T>
{
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

- Subscription: Defines a one-to-one relationship between a `Publisher` and a `Subscriber`. It can only be used once by a single `Subscriber`. It is used to both signal desire for data and cancel demand (and allow resource cleanup).

Subscription.java

```
public interface Subscription<T>
{
    public void request(long n);
    public void cancel();
}
```

- Processor: Represents a processing stage consisting of both a `Subscriber` and a `Publisher` and obeys the contracts of both.

Processor.java

```
public interface Processor<T, R> extends Subs
{
}
```

Two popular implementations of reactive streams are RxJava (https://github.com/ReactiveX/RxJava) and **Project Reactor** (https://projectreactor.io/).

3. What is Spring WebFlux ?

Spring WebFlux is parallel version of Spring MVC and supports fully non-blocking reactive streams. It support the back pressure concept and uses **Netty** as inbuilt server to run reactive applications. If you are familiar with Spring MVC programming style, you can easily work on webflux also.

Spring webflux uses project reactor as reactive library. Reactor is a Reactive Streams library and, therefore, all of its operators support non-blocking back pressure. It is developed in close collaboration with Spring.

Spring WebFlux heavily uses two publishers :

- **Mono**: Returns 0 or 1 element.

```
Mono<String> mono = Mono.just("Alex");
Mono<String> mono = Mono.empty();
```

- **Flux**: Returns 0…N elements. A Flux can be endless, meaning that it can keep emitting elements forever. Also it can return a sequence of elements and then send a completion notification when it has returned all of its elements.

```
Flux<String> flux = Flux.just("A", "B", "C");
Flux<String> flux = Flux.fromArray(new String
Flux<String> flux = Flux.fromIterable(Arrays.

//To subscribe call method

flux.subscribe();
```

In Spring WebFlux, we call reactive APIs/functions that return monos and fluxes and your controllers will return monos and fluxes. When you invoke an API that returns a mono or a flux, it will return immediately. The results of the function call will be delivered to you through the mono or flux when they become available.

> To build a truly non-blocking application, we must aim to create/use all of its components as non-blocking i.e. client, controller, middle services and even the database. If one of them is blocking the requests, our aim will be defeated.

4. Spring Boot WebFlux Example

In this Spring boot 2 application, I am creating employee management system. I chosen it because, while learning, you can compare it with traditional MVC style application. To make it fully non-blocking, I am using **mongodb** as back-end database.

4.1. Maven dependencies

Include `spring-boot-starter-webflux`, `spring-boot-starter-data-mongodb-reactive`, `spring-boot-starter-test` and `reactor-test` dependencies.

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
    xsi:schemaLocation="http://maven.apache.org/POM
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId
        <artifactId>spring-boot-starter-parent</art
        <version>2.1.1.RELEASE</version>
        <relativePath /> <!-- lookup parent from re
    </parent>

    <groupId>com.howtodoinjava</groupId>
    <artifactId>spring-webflux-demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-webflux-demo</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</proje
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</grou
            <artifactId>spring-boot-starter-webflux
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</grou
            <artifactId>spring-boot-starter-data-mo
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</grou
            <artifactId>spring-boot-starter-test</a
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-test</artifactId>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
            <version>2.3.0</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifact
            <version>3.1.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

</project>
```

4.2. Configurations

Webflux Configuration

WebFluxConfig.java

```
import org.springframework.context.annotation.Confi

@Configuration
@EnableWebFlux
public class WebFluxConfig implements WebFluxConfig
{
}
```

MongoDb Configuration

MongoConfig.java

```
import org.springframework.beans.factory.annotatior
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Confi
import org.springframework.data.mongodb.config.Abst
import org.springframework.data.mongodb.core.Reacti
import org.springframework.data.mongodb.repository.

import com.mongodb.reactivestreams.client.MongoClie
import com.mongodb.reactivestreams.client.MongoClie

@Configuration
```

```
@EnableReactiveMongoRepositories(basePackages = "co
public class MongoConfig extends AbstractReactiveMo
{
    @Value("${port}")
    private String port;

    @Value("${dbname}")
    private String dbName;

    @Override
    public MongoClient reactiveMongoClient() {
        return MongoClients.create();
    }

    @Override
    protected String getDatabaseName() {
        return dbName;
    }

    @Bean
    public ReactiveMongoTemplate reactiveMongoTempl
        return new ReactiveMongoTemplate(reactiveMo
    }
}
```

Application Configuration

```
AppConfig.java
```

```
import org.springframework.beans.factory.config.Pro
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Confi
import org.springframework.core.io.ClassPathResourc

@Configuration
public class AppConfig
{
    @Bean
    public static PropertyPlaceholderConfigurer get
    {
        PropertyPlaceholderConfigurer ppc = new Pro
        ppc.setLocation(new ClassPathResource("appl
        ppc.setIgnoreUnresolvablePlaceholders(true)
        return ppc;
    }
}
```

Properties file

```
application.properties
```

```
port=27017
dbname=testdb
```

Logging configuration

logback.xml

```
<configuration>

    <appender name="STDOUT"
        class="ch.qos.logback.core.ConsoleAppender'
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-!
            </pattern>
        </encoder>
    </appender>

    <logger name="org.springframework" level="DEBU(
        additivity="false">
        <appender-ref ref="STDOUT" />
    </logger>

    <root level="ERROR">
        <appender-ref ref="STDOUT" />
    </root>

</configuration>
```

Spring boot application

WebfluxFunctionalApp.java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.Sprir

@SpringBootApplication
public class WebfluxFunctionalApp {

    public static void main(String[] args) {
        SpringApplication.run(WebfluxFunctionalApp.
    }
}
```

4.3. REST Controller

EmployeeController.java

```
import org.springframework.beans.factory.annotatior
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.Patl
import org.springframework.web.bind.annotation.Requ
import org.springframework.web.bind.annotation.Requ
import org.springframework.web.bind.annotation.Requ
import org.springframework.web.bind.annotation.Resp
import org.springframework.web.bind.annotation.Rest

import com.howtodoinjava.demo.model.Employee;
import com.howtodoinjava.demo.service.EmployeeServi

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@RestController
public class EmployeeController {
    @Autowired
    private EmployeeService employeeService;

    @RequestMapping(value = { "/create", "/" }, met
    @ResponseStatus(HttpStatus.CREATED)
    public void create(@RequestBody Employee e) {
        employeeService.create(e);
    }

    @RequestMapping(value = "/{id}", method = Reque
    public ResponseEntity<Mono<Employee>> findById(
        Mono<Employee> e = employeeService.findById
        HttpStatus status = e != null ? HttpStatus.
        return new ResponseEntity<Mono<Employee>>(e
    }

    @RequestMapping(value = "/name/{name}", method
    public Flux<Employee> findByName(@PathVariable(
        return employeeService.findByName(name);
    }

    @RequestMapping(method = RequestMethod.GET, pro
    public Flux<Employee> findAll() {
        Flux<Employee> emps = employeeService.findA
        return emps;
    }

    @RequestMapping(value = "/update", method = Req
    @ResponseStatus(HttpStatus.OK)
    public Mono<Employee> update(@RequestBody Emplo
        return employeeService.update(e);
    }

    @RequestMapping(value = "/delete/{id}", method
    @ResponseStatus(HttpStatus.OK)
    public void delete(@PathVariable("id") Integer
        employeeService.delete(id).subscribe();
    }

}
```

4.4. Service classes

```
IEmployeeService.java
```

```
import com.howtodoinjava.demo.model.Employee;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

public interface IEmployeeService
{
    void create(Employee e);

    Mono<Employee> findById(Integer id);

    Flux<Employee> findByName(String name);

    Flux<Employee> findAll();

    Mono<Employee> update(Employee e);

    Mono<Void> delete(Integer id);
}
```

EmployeeService.java

```
import org.springframework.beans.factory.annotatio
import org.springframework.stereotype.Service;

import com.howtodoinjava.demo.dao.EmployeeReposito
import com.howtodoinjava.demo.model.Employee;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@Service
public class EmployeeService implements IEmployeeS

    @Autowired
    EmployeeRepository employeeRepo;

    public void create(Employee e) {
        employeeRepo.save(e).subscribe();
    }

    public Mono<Employee> findById(Integer id) {
        return employeeRepo.findById(id);
    }

    public Flux<Employee> findByName(String name)
        return employeeRepo.findByName(name);
    }

    public Flux<Employee> findAll() {
        return employeeRepo.findAll();
    }

    public Mono<Employee> update(Employee e) {
        return employeeRepo.save(e);
    }
    public Mono<Void> delete(Integer id) {
        return employeeRepo.deleteById(id);
    }

}
```

4.5. DAO repository

```
EmployeeRepository.java
```

```
import org.springframework.data.mongodb.repository.
import org.springframework.data.mongodb.repository.

import com.howtodoinjava.demo.model.Employee;

import reactor.core.publisher.Flux;

public interface EmployeeRepository extends Reacti
    @Query("{ 'name': ?0 }")
    Flux<Employee> findByName(final String name);
}
```

4.6. Model

```
Employee.java
```

```
import org.springframework.context.annotation.Scope
import org.springframework.context.annotation.Scope
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mappir

@Scope(scopeName = "request", proxyMode = ScopedPro
@Document
public class Employee {

    @Id
    int id;
    String name;
    long salary;

    //Getters and setters

    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + r
    }
}
```

5. Demo

Start the application and check requests and responses.

- HTTP POST http://localhost:8080/create

```
API Request 1
```

```
{
	"id":1,
	"name":"user_1",
	"salary":101
}
```

```
API Request 2
```

```
{
	"id":2,
	"name":"user_2",
	"salary":102
}
```

- HTTP PUT http://localhost:8080/update

```
API Request
```

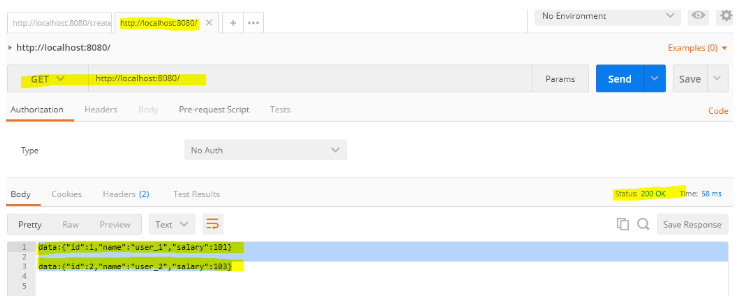
```
{
	"id":2,
	"name":"user_2",
	"salary":103
}
```

- HTTP GET http://localhost:8080/

```
API Response
```

```
data:{"id":1,"name":"user_1","salary":101}

data:{"id":2,"name":"user_2","salary":102}
```



http://localhost:8080/

GET http://localhost:8080/

Type: No Auth

Status: 200 OK Time: 58 ms

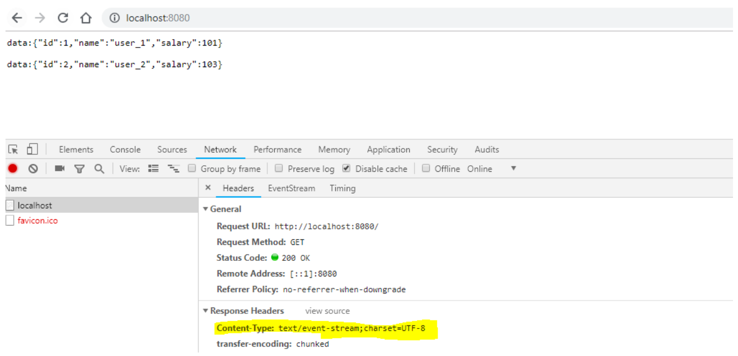
```
data:{"id":1,"name":"user_1","salary":101}

data:{"id":2,"name":"user_2","salary":103}
```

Spring WebFlux Demo

Notice that I am testing the API with **Postman chrome browser extension** which is a blocking client. It will display the result only when It has collected the response for both employees.

To verify the non-blocking response feature, hit the URL in the chrome browser directly. The results will appear one by one, as and when they are available in form of events (**text/event-stream**). To better view the result, consider adding a delay to controller API.



```
data:{"id":1,"name":"user_1","salary":101}

data:{"id":2,"name":"user_2","salary":103}
```

```
▾ General
    Request URL: http://localhost:8080/
    Request Method: GET
    Status Code: 200 OK
    Remote Address: [::1]:8080
    Referrer Policy: no-referrer-when-downgrade
▾ Response Headers
    Content-Type: text/event-stream;charset=UTF-8
    transfer-encoding: chunked
```

Spring WebFlux Demo – Event Stream

6. Spring WebFlux Tutorial – Conclusion

Both Spring MVC and Spring WebFlux support client-server architecture but there is a key difference in the concurrency model and the default behavior for blocking nature and threads. In Spring MVC, it is assumed that applications can block the current thread while in webflux, threads are non-blocking by default. It is the main difference between **spring webflux vs mvc**.

Reactive and non-blocking generally do not make applications run faster. The expected benefit of reactive and non-blocking is the ability to scale the application with a small, fixed number of threads and lesser memory requirements. It makes applications more resilient under load because they scale in a more predictable manner.

Drop me your questions related to this **spring boot webflux tutorial**.

Happy Learning !!