**Spring WebFlux Tutorial**

By Lokesh Gupta | Filed Under: [Spring WebFlux](https://howtodoinjava.com/spring-webflux/)

The reactive-stack web framework, **Spring WebFlux**, has been added Spring 5.0. It is fully non-blocking, supports [reactive streams](http://www.reactive-streams.org/) 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**

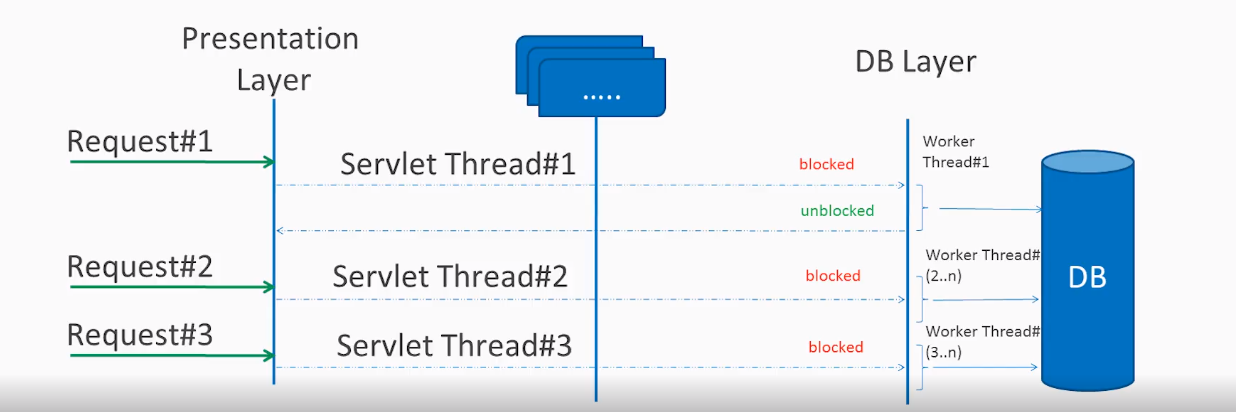
Reactive programming is a programming paradigm that promotes an asynchronous, non-blocking, event-driven approach to data processing. Reactive programming involves modeling data and events as observable data streams and implementing data processing routines to react to the changes in those streams.

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

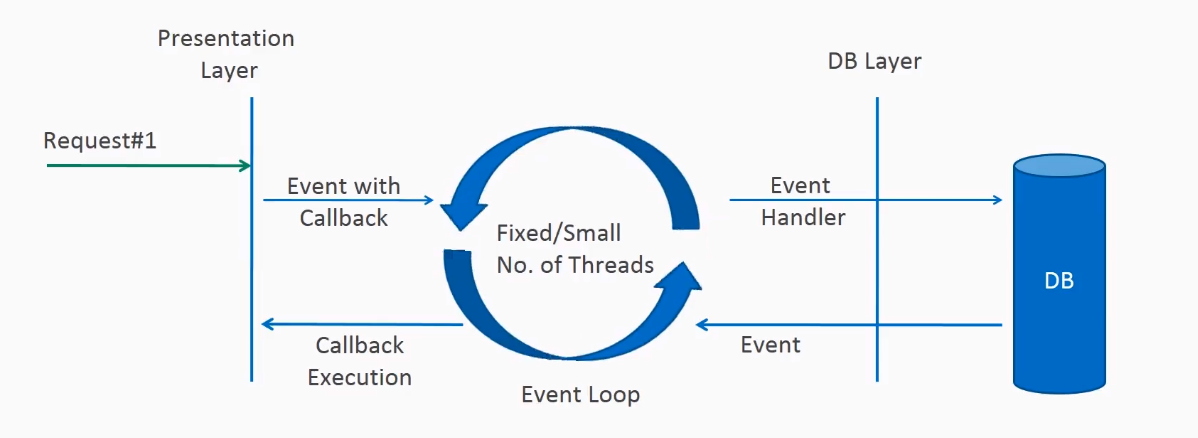
As server can have some finite number of request threads, it limits the server capability to process that number of requests at maximum server load. It may hamper the performance and limit the full utilization of server capability.

**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](https://howtodoinjava.com/design-patterns/behavioral/observer-design-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.

**Spring Boot WebFlux Example**

In this [**Spring boot 2**](https://howtodoinjava.com/spring-boot-tutorials/) 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-instance>"      xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0> <http://maven.apache.org/xsd/maven-4.0.0.xsd>">      <modelVersion>4.0.0</modelVersion>        <parent>          <groupId>org.springframework.boot</groupId>          <artifactId>spring-boot-starter-parent</artifactId>          <version>2.1.1.RELEASE</version>          <relativePath /> <!-- lookup parent from repository -->      </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](http://maven.apache.org/)</url>        <properties>          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>          <java.version>1.8</java.version>      </properties>        <dependencies>          <dependency>              <groupId>org.springframework.boot</groupId>              <artifactId>spring-boot-starter-webflux</artifactId>          </dependency>          <dependency>              <groupId>org.springframework.boot</groupId>              <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>          </dependency>          <dependency>              <groupId>org.springframework.boot</groupId>              <artifactId>spring-boot-starter-test</artifactId>              <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</artifactId>              <version>3.1.0</version>              <scope>provided</scope>          </dependency>      </dependencies>    </project> |

**4.2. Configurations**

**Webflux Configuration**

|  |
| --- |
| WebFluxConfig.java |
| import org.springframework.context.annotation.Configuration;    @Configuration  @EnableWebFlux  public class WebFluxConfig implements WebFluxConfigurer  {  } |

**MongoDb Configuration**

|  |
| --- |
| MongoConfig.java |
| import org.springframework.beans.factory.annotation.Value;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.data.mongodb.config.AbstractReactiveMongoConfiguration;  import org.springframework.data.mongodb.core.ReactiveMongoTemplate;  import org.springframework.data.mongodb.repository.config.EnableReactiveMongoRepositories;    import com.mongodb.reactivestreams.client.MongoClient;  import com.mongodb.reactivestreams.client.MongoClients;    @Configuration  @EnableReactiveMongoRepositories(basePackages = "com.howtodoinjava.demo.dao")  public class MongoConfig extends AbstractReactiveMongoConfiguration  {      @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 reactiveMongoTemplate() {          return new ReactiveMongoTemplate(reactiveMongoClient(), getDatabaseName());      }  } |

**Application Configuration**

|  |
| --- |
| AppConfig.java |
| import org.springframework.beans.factory.config.PropertyPlaceholderConfigurer;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.core.io.ClassPathResource;    @Configuration  public class AppConfig  {      @Bean      public static PropertyPlaceholderConfigurer getPropertyPlaceholderConfigurer()      {          PropertyPlaceholderConfigurer ppc = new PropertyPlaceholderConfigurer();          ppc.setLocation(new ClassPathResource("application.properties"));          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] %-5level %logger{5} - %msg%n              </pattern>          </encoder>      </appender>        <logger name="org.springframework" level="DEBUG"          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.SpringBootApplication;    @SpringBootApplication  public class WebfluxFunctionalApp {        public static void main(String[] args) {          SpringApplication.run(WebfluxFunctionalApp.class, args);      }  } |

**4.3. REST Controller**

|  |
| --- |
| EmployeeController.java |
| import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.http.HttpStatus;  import org.springframework.http.MediaType;  import org.springframework.http.ResponseEntity;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.RequestBody;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.ResponseStatus;  import org.springframework.web.bind.annotation.RestController;    import com.howtodoinjava.demo.model.Employee;  import com.howtodoinjava.demo.service.EmployeeService;    import reactor.core.publisher.Flux;  import reactor.core.publisher.Mono;    @RestController  public class EmployeeController {      @Autowired      private EmployeeService employeeService;        @RequestMapping(value = { "/create", "/" }, method = RequestMethod.POST)      @ResponseStatus(HttpStatus.CREATED)      public void create(@RequestBody Employee e) {          employeeService.create(e);      }        @RequestMapping(value = "/{id}", method = RequestMethod.GET)      public ResponseEntity<Mono<Employee>> findById(@PathVariable("id") Integer id) {          Mono<Employee> e = employeeService.findById(id);          HttpStatus status = e != null ? HttpStatus.OK : HttpStatus.NOT\_FOUND;          return new ResponseEntity<Mono<Employee>>(e, status);      }        @RequestMapping(value = "/name/{name}", method = RequestMethod.GET)      public Flux<Employee> findByName(@PathVariable("name") String name) {          return employeeService.findByName(name);      }        @RequestMapping(method = RequestMethod.GET, produces = MediaType.TEXT\_EVENT\_STREAM\_VALUE)      public Flux<Employee> findAll() {          Flux<Employee> emps = employeeService.findAll();          return emps;      }        @RequestMapping(value = "/update", method = RequestMethod.PUT)      @ResponseStatus(HttpStatus.OK)      public Mono<Employee> update(@RequestBody Employee e) {          return employeeService.update(e);      }        @RequestMapping(value = "/delete/{id}", method = RequestMethod.DELETE)      @ResponseStatus(HttpStatus.OK)      public void delete(@PathVariable("id") Integer id) {          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.annotation.Autowired;  import org.springframework.stereotype.Service;    import com.howtodoinjava.demo.dao.EmployeeRepository;  import com.howtodoinjava.demo.model.Employee;    import reactor.core.publisher.Flux;  import reactor.core.publisher.Mono;    @Service  public class EmployeeService implements IEmployeeService {        @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.Query;  import org.springframework.data.mongodb.repository.ReactiveMongoRepository;    import com.howtodoinjava.demo.model.Employee;    import reactor.core.publisher.Flux;    public interface EmployeeRepository extends ReactiveMongoRepository<Employee, Integer> {      @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.ScopedProxyMode;  import org.springframework.data.annotation.Id;  import org.springframework.data.mongodb.core.mapping.Document;    @Scope(scopeName = "request", proxyMode = ScopedProxyMode.TARGET\_CLASS)  @Document  public class Employee {        @Id      int id;      String name;      long salary;        //Getters and setters        @Override      public String toString() {          return "Employee [id=" + id + ", name=" + name + ", salary=" + salary + "]";      }  } |

**5. Demo**

Start the application and check requests and responses.

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

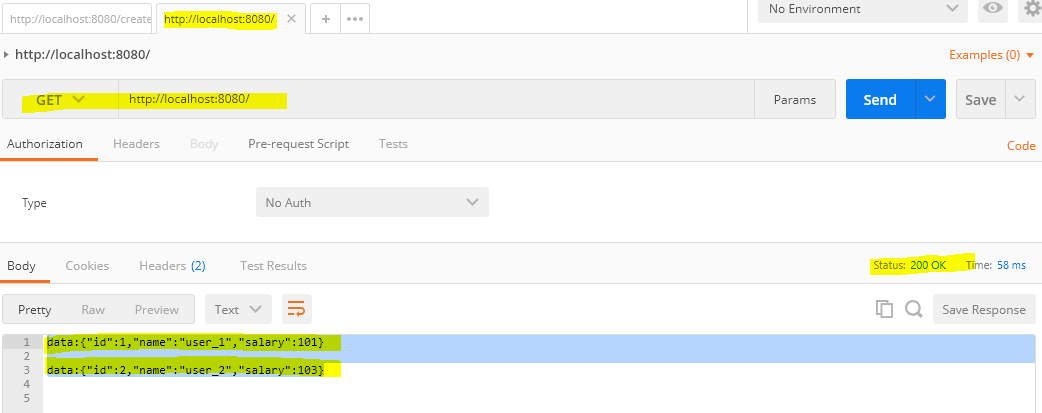
|  |
| --- |
| API Request 1 |
| {      "id":1,      "name":"user\_1",      "salary":101  } |
| API Request 2 |
| {      "id":2,      "name":"user\_2",      "salary":102  } |

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

|  |
| --- |
| API Request |
| {      "id":2,      "name":"user\_2",      "salary":103  } |

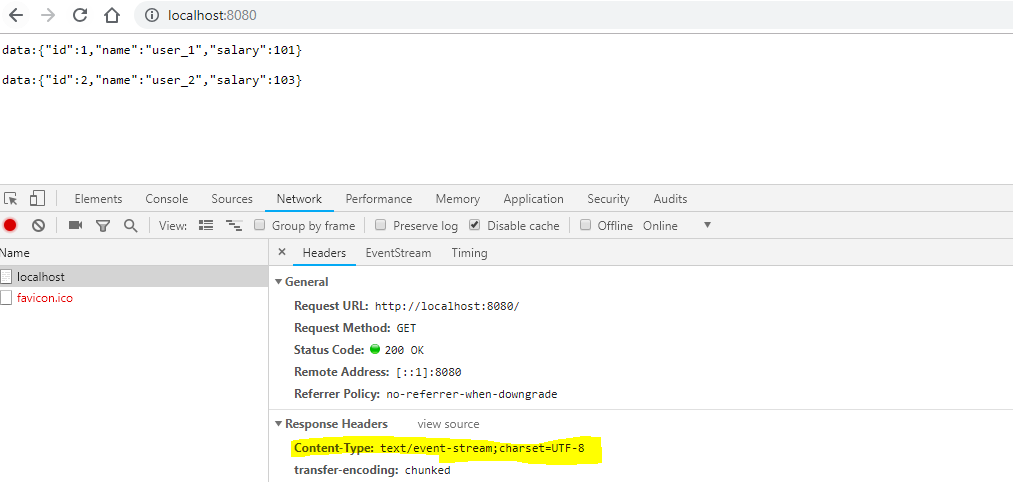
 HTTP GET http://localhost:8080/

|  |
| --- |
| API Response |
| data:{"id":1,"name":"user\_1","salary":101}    data:{"id":2,"name":"user\_2","salary":102} |

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.

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](https://howtodoinjava.com/java-concurrency-tutorial/) 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.