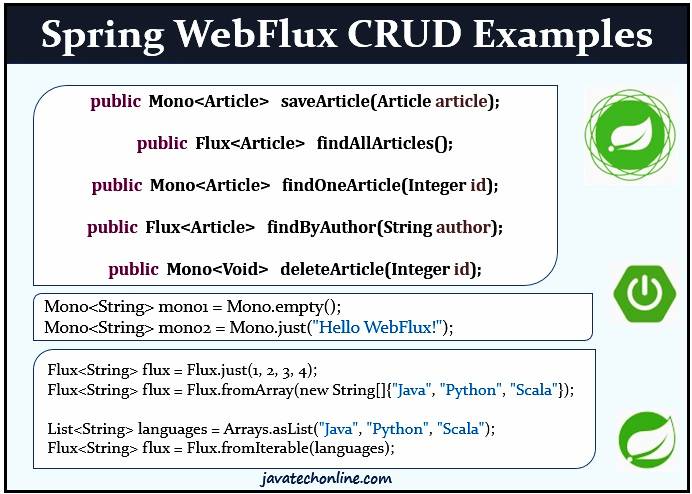
# Spring WebFlux CRUD Example:



All Java developers must have worked on traditional REST APIs at some point in their career. Other than the traditional approach of developing a REST API, now we have a new upgraded way of doing it. Reactive Programming as a new approach to working with REST APIs. Compared to traditional REST development, this approach is faster and offers for optimal resource utilization. In this article, we will work on the concepts of Spring WebFlux, which comes under the Reactive Programming model.

Spring WebFlux is a reactive web framework built on top of Spring Framework 5, designed to handle the demands of modern web applications. Spring WebFlux provides a non-blocking and event-driven programming model that allows developers to build scalable and high-performance web applications. In this article, we will walk you through a Spring WebFlux CRUD example that demonstrates the basic operations of Create, Read, Update, and Delete.

## What is Spring Webflux?

Spring WebFlux is a reactive web framework in the Spring ecosystem, introduced in Spring Framework 5.0. It provides a fully non-blocking and [reactive programming](https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape) model for building web applications that can handle a large number of concurrent connections with a small number of threads.

WebFlux is built on top of Reactor, a reactive programming library for building asynchronous and non-blocking applications. Spring WebFlux provides a reactive programming model that allows you to handle streams of data, such as events, and perform operations on them in a non-blocking manner.

Reactive programming is a programming paradigm that focuses on building asynchronous and non-blocking applications.

## What are Mono & Flux in Spring WebFlux?

In Spring WebFlux, Mono and Flux are two core classes used for handling asynchronous streams of data, such as the response from a REST API.

Mono represents a stream of zero or one element, while Flux represents a stream of zero or more elements. Here are some examples to illustrate their usage:

### Example 1: Creating a Mono

**Mono<String> mono = Mono.empty();**

**Mono<String> mono = Mono.just("Hello");**

In this example, we create a Mono that returns a zero or single element, the string “Hello, world!”.

### Example 2: Creating a Flux

**Flux<String> flux = Flux.just(1, 2, 3, 4);**

**Flux<String> flux = Flux.fromArray(new String[]{"Java", "Python", "Scala"});**

**List<String> languages = Arrays.asList("Java", "Python", "Scala");**

**Flux<String> flux = Flux.fromIterable(languages);**

### Example 3: Combining Monos and Fluxes

**Mono<Integer> mono1 = Mono.just(1);**

**Mono<Integer> mono2 = Mono.just(2);**

**Flux<Integer> flux = Flux.fromArray(new Integer[] { 3, 4, 5 });**

**Flux<Integer> combined = Flux.concat(mono1, mono2, flux);**

In this example, we created two Monos that returns a single integer each, and a Flux that returns three integers. We then used the concat operator to combine them into a single Flux that returns all the integers in the order in which they were defined.

These are just a few examples of how Mono and Flux can be used in Spring WebFlux. By using these classes and the many operators available in Reactor, we can build powerful and flexible asynchronous applications.

## What are the minimum Software required to support Spring WebFlux?

* Spring 5.x to support Spring Web Flux
* Servlets 3.1+
* Spring Boot uses Netty Server by default as it is well-established in the asynchronous, non-blocking space.

## Spring WebFlux CRUD Example

In this example, we will create a simple RESTful API that exposes CRUD operations for managing articles. We will use MongoDB as the database and ReactiveMongoRepository as the reactive implementation of the Repository to interact with the database.

### Prerequisites

Before we start with the example, make sure you have the following prerequisites installed on your machine.

* Java: JDK 8 or higher version
* Database: MongoDB
* IDE: STS, Eclipse, IntelliJ, or any other
* Build Tool: Maven or Gradle
* Testing Tool: Postman or any other REST client Testing Tool

### Use case Details

Let’s consider an entity ‘Article’ as a model to develop the Spring Webflux CRUD Example. The article entity will have 5 fields named as ‘id’, ‘title’, ‘content’, ‘author’, and ‘publishedAt’. As our target is to develop CRUD operations, we will work on various methods, such as createArticle(), findArticle(), findAllArticles(), updateArticle(), deleteArticle(), findArticleByAuthor() etc.

### Step#1: Create a Spring Boot Project

In this example, we have used STS (Spring Tool Suite) as an IDE to create Spring Boot Project. If you are new to create Spring Boot project using STS, visit the separate article on [how to create a sample project in spring boot using STS](https://javatechonline.com/saving-data-into-database-using-spring-boot-data-jpa-step-by-step-tutorial/#Step_1_Creating_Starter_Project_using_STS).

While creating starter project in STS, select starter dependencies as  ‘Spring Reactive Web’, ‘Spring Data Reactive Mongo DB’. You can also add ‘Spring Boot DevTools’ optionally.

Below are the required dependencies:

**<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-webflux</artifactId>**

**</dependency>**

Please note that we are using Mongo DB as database because reactive stack has direct support of only NoSQL databases. If you still have not installed Mongo DB in your system, follow the instructions given in the article ‘[How to install Mongo DB in your system?’](https://javatechonline.com/how-to-install-mongo-db-in-your-system/). To get an easy grasp on ‘How to work with Mongo DB’, please visit [Mongo DB With Spring Boot Tutorial](https://javatechonline.com/how-to-work-in-spring-boot-with-mongodb/).

### Step#2: Configure MongoDB

To connect with MongoDB in our Spring Boot project, we need to provide the connection details in the application.properties file.

**# application.properties**

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

**spring.data.mongodb.port=27017**

**spring.data.mongodb.database=reactivedb**

In this example, we are using a local MongoDB instance running on the default port 27017.

### Step#3: Create the Article Entity Class

Create a new entity class called Article.java as shown in the following code:

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

**Article.java**

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

**package com.dev.springboot.webflux.model;**

**import org.springframework.data.annotation.Id;**

**import org.springframework.data.mongodb.core.mapping.Document;**

**import java.util.Date;**

**@Document(collection = "articles")**

**public class Article {**

**@Id**

**private Integer id;**

**private String title;**

**private String content;**

**private String author;**

**private Date publishedAt;**

**public Integer getId() {**

**return id;**

**}**

**public void setId(Integer id) {**

**this.id = id;**

**}**

**public String getTitle() {**

**return title;**

**}**

**public void setTitle(String title) {**

**this.title = title;**

**}**

**public String getContent() {**

**return content;**

**}**

**public void setContent(String content) {**

**this.content = content;**

**}**

**public String getAuthor() {**

**return author;**

**}**

**public void setAuthor(String author) {**

**this.author = author;**

**}**

**public Date getPublishedAt() {**

**return publishedAt;**

**}**

**public void setPublishedAt(Date publishedAt) {**

**this.publishedAt = publishedAt;**

**}**

**}**

In this Article entity, we have defined the following fields:

id: The unique identifier for the Article.  
title: The title of the article.  
content: The content of the article.  
author: The author of the article.  
publishedAt: The article’s published date.

We have also annotated the class with @Document, which tells the Spring container to map this class to a MongoDB collection. As we have provided collection name as ‘**collection = “articles”‘,**it will map our data to ‘articles’ collection in Mongo DB. If we don’t provide collection attribute, it will search for an exact entity name to map the data.

### Step#4: Create Repository Interface as ArticleRepository.java

Create a new Java interface called ArticleRepository.java as shown below in the following code:

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

**ArticleRepository.java**

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

**package com.dev.springboot.webflux.repository;**

**import org.springframework.data.mongodb.repository.Query;**

**import org.springframework.data.mongodb.repository.ReactiveMongoRepository;**

**import org.springframework.stereotype.Repository;**

**import com.dev.springboot.webflux.model.Article;**

**import reactor.core.publisher.Flux;**

**@Repository**

**public interface ArticleRepository extends ReactiveMongoRepository<Article, Integer> {**

**@Query("{'author': ?0}")**

**Flux<Article> findByAuthor(String author);**

**}**

This repository interface extends the ReactiveMongoRepository interface provided by Spring Data MongoDB. It provides some basic methods by default in order to create CRUD operations for interacting with the MongoDB database. We can access those methods in our service classes directly with the help of the repository object without declaring them here. If we want to implement a custom method, we need to declare it here in the Repository interface. For example, findByAuthor() is our custom method.

In order to know more about usage of @Query annotation, kindly visit a separate article on [Spring Boot MongoDB @Query Examples](https://javatechonline.com/spring-boot-mongodb-query-examples/).

### Step#5: Create a Service Interface as ArticleService.java

Create a new service Interface called ArticleService.java and declare required methods as shown below:

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

**ArticleService.java**

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

**package com.dev.springboot.webflux.service;**

**import com.dev.springboot.webflux.model.Article;**

**import reactor.core.publisher.Flux;**

**import reactor.core.publisher.Mono;**

**public interface ArticleService {**

**public Mono<Article> saveArticle(Article article);**

**public Flux<Article> findAllArticles();**

**public Mono<Article> findOneArticle(Integer id);**

**public Flux<Article> findByAuthor(String author);**

**public Mono<Void> deleteArticle(Integer id);**

**}**

In this interface, we have declared the following methods:

saveArticle(): This method will create a new article and saves into the database.

findAllArticles(): This method will retrieve all the articles from the database and returns them as a Flux.

findOneArticle(): This method will retrieve a single article by its ID and returns it as a Mono.

findByAuthor(): This method will retrieve all the articles for a particular author from the database and returns them as a Flux.

deleteArticle(): This method will delete an existing article by its ID.

### Step#6: Create a Service Implementation as ArticleServiceImpl.java

Create a new service Implementation class called ArticleServiceImpl.java and implement all the methods of Service Interface as shown below:

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

**ArticleServiceImpl.java**

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

**package com.dev.springboot.webflux.service.impl;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.stereotype.Service;**

**import com.dev.springboot.webflux.model.Article;**

**import com.dev.springboot.webflux.repository.ArticleRepository;**

**import com.dev.springboot.webflux.service.ArticleService;**

**import reactor.core.publisher.Flux;**

**import reactor.core.publisher.Mono;**

**@Service**

**public class ArticleServiceImpl implements ArticleService {**

**@Autowired**

**private ArticleRepository articleRepository;**

**@Override**

**public Mono<Article> saveArticle(Article article) {**

**return articleRepository.save(article);**

**//for Mono<String> return type**

**//return Mono.just("saved successfully");**

**}**

**@Override**

**public Flux<Article> findAllArticles() {**

**return articleRepository.findAll().switchIfEmpty(Flux.empty());**

**}**

**@Override**

**public Mono<Article> findOneArticle(Integer id) {**

**return articleRepository.findById(id).switchIfEmpty(Mono.empty());**

**}**

**@Override**

**public Flux<Article> findByAuthor(String author) {**

**return articleRepository.findByAuthor(author);**

**}**

**@Override**

**public Mono<Void> deleteArticle(Integer id) {**

**return articleRepository.deleteById(id);**

**}**

**}**

In this class, we have implemented all the methods of ArticleService interface.

### Step#7: Create a Controller as ArticleController.java

Create a new controller class called ArticleController.java and implement all the methods with end points as shown below:

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

**ArticleController.java**

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

**package com.dev.springboot.webflux.controller;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.http.HttpStatus;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.DeleteMapping;**

**import org.springframework.web.bind.annotation.GetMapping;**

**import org.springframework.web.bind.annotation.PathVariable;**

**import org.springframework.web.bind.annotation.PostMapping;**

**import org.springframework.web.bind.annotation.PutMapping;**

**import org.springframework.web.bind.annotation.RequestBody;**

**import org.springframework.web.bind.annotation.RequestMapping;**

**import org.springframework.web.bind.annotation.RestController;**

**import com.dev.springboot.webflux.model.Article;**

**import com.dev.springboot.webflux.service.ArticleService;**

**import reactor.core.publisher.Flux;**

**import reactor.core.publisher.Mono;**

**@RestController**

**@RequestMapping("/api/articles")**

**public class ArticleController {**

**@Autowired**

**private ArticleService articleService;**

**@GetMapping("/findAll")**

**public Flux<Article> getAllArticles() {**

**return articleService.findAllArticles();**

**}**

**@PostMapping("/save")**

**public Mono<ResponseEntity<Article>> createArticle(@RequestBody Article article) {**

**return articleService.saveArticle(article)**

**.map(savedArticle -> new ResponseEntity<>(savedArticle, HttpStatus.CREATED));**

**}**

**@GetMapping("/{id}")**

**public Mono<ResponseEntity<Article>> getArticleById(@PathVariable Integer articleId) {**

**return articleService.findOneArticle(articleId)**

**.map(article -> ResponseEntity.ok(article))**

**.defaultIfEmpty(ResponseEntity.notFound().build());**

**}**

**@GetMapping("/{author}")**

**public Flux<ResponseEntity<Article>> getArticleByAuthor(@PathVariable String author) {**

**return articleService.findByAuthor(author)**

**.map(article -> ResponseEntity.ok(article))**

**.defaultIfEmpty(ResponseEntity.notFound().build());**

**}**

**@PutMapping("/update/{id}")**

**public Mono<ResponseEntity<Article>> updateArticle(@PathVariable Integer articleId,**

**@RequestBody Article article) {**

**return articleService.findOneArticle(articleId)**

**.flatMap(existingArticle -> {**

**existingArticle.setTitle(article.getTitle());**

**existingArticle.setContent(article.getContent());**

**existingArticle.setAuthor(article.getAuthor());**

**existingArticle.setPublishedAt(article.getPublishedAt());**

**return articleService.saveArticle(existingArticle);**

**})**

**.map(updatedArticle -> new ResponseEntity<>(updatedArticle, HttpStatus.OK))**

**.defaultIfEmpty(new ResponseEntity<>(HttpStatus.NOT\_FOUND));**

**}**

**@DeleteMapping("/delete/{id}")**

**public Mono<ResponseEntity<Void>> deleteArticle(@PathVariable Integer articleId) {**

**return articleService.deleteArticle(articleId)**

**.then(Mono.just(new ResponseEntity<Void>(HttpStatus.OK)))**

**.onErrorResume(error -> Mono.just(new ResponseEntity<Void>(HttpStatus.NOT\_FOUND)));**

**}**

**}**

In this class, we have implemented the following methods:

**createArticle():** This method creates a new article and returns the URI of the created resource in the Location header of the response.

**getAllAtricles():** This method retrieves all the articles from the database and returns them as a Flux.

**getArticleById():** This method retrieves a single article by its ID and returns it as a Mono. If the article is not found, it returns a 404 Not Found response.

**getArticleByAuthor():** This method retrieves all articles by its author and returns it as a Flux. If the article is not found, it returns a 404 Not Found response.

**updateArticle():** This method updates an existing article and returns the updated article as a Mono. If the article is not found, it returns a 404 Not Found response.

**deleteArticle():** This method deletes an existing article by its ID and returns a 200 OK response. If the article is not found, it returns a 404 Not Found response.

### Step#8: Test the Application

We have implemented the CRUD operations, let’s test our application.

We can use the Postman tool to test the application.

* To create a new article, send a POST request to ‘http://localhost:8080/api/articles/save’ with the following JSON payload:

**{**

**"title": "Title1",**

**"content": "Content for Title1",**

**"author": "Author for Title1",**

**"publishedAt": "2023-04-24T15:19:06.539Z"**

**}**

The response should include the Location header with the URI of the created resource. Take extra care while passing the date value. You may need to convert it into a specific format at service class.

* To retrieve all articles, send a GET request to ‘http://localhost:8080/api/articles/findAll’.

The response should include a JSON array with all the articles.

* To retrieve a single article by its ID, send a GET request to ‘http://localhost:8080/api/articles/{id}’, where {id} is the ID of the article.

The response should include a JSON object with the article details.

* To update an article, send a PUT request to ‘http://localhost:8080/api/articles/update/{id}’, where {id} is the ID of the article, with the following JSON payload:

**{**

**"title": "Title2",**

**"content": "Content for Title2",**

**"author": "Author for Title2",**

**"publishedAt": "2023-04-24T124:19:06.539Z"**

**}**

The response should include a JSON object with the updated article details.

* To delete an article, send a DELETE request to ‘http://localhost:8080/api/articles/delete/{id}’, where {id} is the ID of the article.

The response should include a 200 OK response.

## Conclusion

In this article, we have learned how to implement a Spring WebFlux CRUD example. We started with the basics of Spring WebFlux, followed by setting up the Spring Boot project and configuring the database. Then, we created the entity ‘Article’ and corresponding repository interface, followed by Service Interface & Service Implementation class. Subsequently, we implemented the Controller methods for each CRUD operation. Finally, we tested the application using the Postman tool with all endpoints.

Spring WebFlux provides a reactive programming model for building web applications that can handle a large number of requests with minimal resources. It offers a non-blocking and event-driven architecture that can handle thousands of concurrent connections. The reactive programming model is particularly useful for applications that need to handle real-time data streams or IoT devices.

In conclusion, Spring WebFlux is a powerful framework that can help developers build scalable and high-performance web applications. If we follow the steps described in this article, we should easily implement a CRUD application using Spring WebFlux.

## FAQs

### ****Q#1. What is the difference between**** Mono****and**** Flux****and traditional Java**** List?

Mono and Flux represent asynchronous, non-blocking operations that can handle streams of data. In contrast, List is not suitable for asynchronous and reactive programming.

### ****Q#2. Is Spring Boot Reactive suitable for all types of applications?****

No, Spring Boot Reactive is best convenient for applications that need to deal with a large number of concurrent users and require high concurrency, such as web applications with real-time features or microservices dealing with a high volume of requests.

### ****Q#3. What are some common use cases for**** Mono****and**** Flux****in Spring Boot Reactive?****

Some common use cases for Mono & Flux can include handling web requests, interacting with databases or external APIs, real-time messaging, and building reactive microservices.

### ****Q#4. Are**** Mono****and**** Flux****only used in Spring WebFlux?****

They are generally used in Spring WebFlux, but we can also use them in other parts of our Spring Boot based application to benefit from reactive programming, even if we are not using WebFlux for the entire application.

### ****Q#5. What is Spring WebFlux?****

Spring WebFlux is a reactive web framework introduced as part of Spring 5. It provides support for reactive programming. It is designed to create fully asynchronous and non-blocking applications. Unlike Spring MVC, WebFlux is built to handle a large number of concurrent requests efficiently.

### ****Q#6****. ****How does Spring WebFlux differ from Spring MVC?****

Spring WebFlux is designed for reactive programming, meaning it’s non-blocking and can handle more simultaneous requests than Spring MVC. In contrast, Spring MVC is synchronous and blocking, which can be less efficient under heavy load.

### ****Q#7****. ****What are the key use cases for Spring WebFlux?****

Spring WebFlux is ideal for applications requiring high scalability and responsiveness, such as real-time data streaming, chat applications, and IoT systems. It is particularly useful when dealing with large numbers of concurrent connections.

### ****Q#8****. ****Can Spring WebFlux be integrated with existing Spring MVC applications?****

Yes, Spring WebFlux can coexist with Spring MVC in the same application, which allows developers to utilize both synchronous and asynchronous processing as per our need. This enables a gradual adoption of reactive programming in existing applications.

### Related