Servlet and Reactive Stacks in Spring Framework 5

This item in japanese

Key Takeaways

- There is a major shift towards asynchronous, non-blocking concurrency in Java, the JVM, and beyond.
- Spring Framework 5 introduces a fully non-blocking, reactive stack for web applications.
- The reactive stack handles higher concurrency with less hardware resources, and excels at streaming scenarios, both client and server side.
- Some reactive features are also available in Spring MVC for existing applications on the servlet stack.
- Spring Boot 2 includes a reactive web starter with Netty as the default server, and Tomcat, Jetty, and Undertow as alternative options.

Spring Framework 5 provides a choice of two web stacks, Servlet and Reactive, available side by side. This reflects a big, general shift towards asynchronous, non-blocking concurrency in applications. The goal of this article is to provide some context, explain the range of available choices, and provide guidance for choosing between these stacks.

For the remainder of this article I will use the "servlet stack" and the "reactive stack" as shorthand for the two web stacks supported in Spring Framework 5, that applications can use through the Spring MVC (`spring-webmvc` module) and the Spring WebFlux (`spring-webflux` module) web frameworks.

Motivation for Change

By now you've heard "reactive" from many different angles, and it has become quite a buzzword. Nevertheless there is a real trend behind it, a story of growing asynchronicity, starting from the early days of servlet containers when applications were relatively simple and self-contained, to the present day where asynchronicity hides in just about every corner. How well equipped are we to deal with all of this asynchronicity?

Traditionally, Java used thread pools for the concurrent execution of blocking, I/O bound operations (e.g. making remote calls). This seems simple on the surface, but it can be deceptively complex for two reasons: one, it's hard to make applications work correctly when you're wrestling with synchronization and shared structures. Two, it's hard to scale efficiently when every blocking operation requires an extra thread to just sit around and wait, and you're at the mercy of latency outside your control (e.g. slow remote clients and services).

Not surprisingly, various solutions have evolved, (such as coroutines, actors, etc.), to relegate the concurrency to the framework or language. There is now a proposal in Java for a lightweight thread model called Project Loom, but it will be years before we see this in production. What can we do today to better deal with asynchronicity?

A common misconception, especially among Java developers, is that scale requires a lot of threads. While this may be true with paradigms like imperative programming and blocking I/O, it is not true in general. If an application is fully non-blocking it can scale with a small, fixed number of threads. Node.js is proof you can scale with a single thread only. In Java we don't have to be limited to one thread so we can fire enough threads to keep all cores busy. Yet the principle remains-- we don't rely on extra threads for higher concurrency.

How do we make our applications non-blocking? First and foremost, we must forfeit the traditional sequential logic associated with imperative programming, in favor of asynchronous APIs, and we must learn to react to events they generate. Of course, working with asynchronous callbacks can become unwieldy very quickly. For a better model we can look to the CompletableFuture introduced in Java 8, which gave us a continuation-style fluent API, where logic is declared in sequential steps rather than in nested callbacks:

```
CompletableFuture.supplyAsync(() -> "stage0")
        .thenApply(s -> s + "-stage1")
        .thenApplyAsync(s -> {
            // insert async call here...
            return s + "-stage2";
        })
        .thenApply(s -> s + "-stage3")
        .thenApplyAsync(s -> {
            // insert async call here...
            return s + "-stage4";
        })
        .thenAccept(System.out::println);
```

This is a better model but supports only a single value. What about handling asynchronous sequences? The Java 8 Stream API provides functional-style operations on streams of elements, but that's built for collections, where a consumer pulls available data, and not for "live" streams where the producer pushes elements, with potential latency in between.

This is where reactive libraries such as RxJava, Reactor, Akka Streams, and others come in. They look like Java 8 Streams, but are designed for asynchronous sequences and add Reactive Streams back pressure, to give consumers control over the publisher's rate.

The switch from imperative to a functional, declarative style does not feel "natural" at first and takes time to adjust. The same is true for many other things in life, for example learning to ride a bicycle or a new language. Don't let that stop you. It does get easier and brings great benefits.

The imperative to declarative transition can be compared to rewriting explicit loops with the Java 8 Stream API, with similar benefits. The Java 8 Stream API lets you declare "what" should be done, not "how", rendering your code more readable. Similarly in reactive libraries, you declare what should be done, not how to deal with concurrency, threads, and synchronization, and so code scales more efficiently using fewer hardware resources.

Last but not least, an additional motivation for change is the lambda syntax in Java 8, which is crucial for the adoption of functional, declarative APIs and reactive libraries, while allowing us to imagine new programming models. Much like annotations let us build annotated REST endpoints, the lambda syntax in Java 8 let us build functional-style routings and request handlers.

Stack Choices

The Spring Framework is not the first in the async, non-blocking web space. But it brings the perspective of Java enterprise applications and choice at all levels. Choice matters because not all existing applications can change and because not every application needs to change. Choice, and also consistency and continuity, come in handy with microservice architecture where independent decisions can be made by application.

Let’s review what choices are available.

The Server

For a long time the Servlet API was the de facto standard for server independence. Over time however, alternatives have appeared, and projects seeking the efficient scale of event loop concurrency and non-blocking I/O have already looked beyond the Servlet API and servlet containers.

To be sure, Tomcat and Jetty have evolved greatly over the years to become more efficient under the hood. What hasn't changed as much in 20 years is the way they're used through the Servlet API with blocking I/O. Non-blocking I/O was introduced in Servlet API 3.1 but has not been adopted because it requires deep change, replacing core, web framework and application contracts built around blocking I/O. In practice this has meant choosing between the Servlet API with blocking I/O, or alternative asynchronous runtimes such as Netty that don’t depend on the Servlet API.

The reactive stack in Spring Framework 5 defers that decision, allowing you to have your blocking and your reactive too. Spring WebFlux applications can run on servlet containers, or adapt to other native server APIs. In Spring Boot 2 the WebFlux starter uses Netty by default, but you can easily change to Tomcat or Jetty in a few lines of configuration. The reactive stack restores a degree of choice that was once possible only through the Servlet API.

Annotated Controllers

The Spring MVC annotation based programming model, familiar to many, is supported on both a servlet stack (Spring MVC) and on a reactive stack (Spring WebFlux). That means you can choose between blocking and non-blocking, event loop concurrency, but keep the web framework programming model.

Reactive Clients

Use of reactive clients enables applications to compose remote service calls efficiently, and concurrently, and yet without explicitly dealing with threads. The benefits are greatly amplified on the server side, with high concurrency.

The use of reactive clients is not limited to just the reactive stack or Spring WebFlux. The code below is also supported on the servlet stack and it shows that a Spring MVC controller can handle requests and render the response through reactive types:

```
@RestController
public class CarController {

    private final WebClient carsClient =
                    WebClient.create("http://localhost:8081");
    private final WebClient bookClient =
                    WebClient.create("http://localhost:8082");

    @PostMapping("/booking")
    public Mono<ResponseEntity<Void>> book() {
        return carsClient.get().uri("/cars")
                .retrieve()
                .bodyToFlux(Car.class)
                .take(5)
                .flatMap(this::requestCar)
                .next();
```

```
    }

    private Mono<ResponseEntity<Void>> requestCar(Car car) {
        return bookClient.post()
                .uri("/cars/{id}/booking", car.getId())
                .exchange()
                .flatMap(response -> response.toEntity(Void.class));
    }
}
```

In the above sample, the controller uses the reactive, non-blocking `WebClient` to fetch cars from one remote service, then attempts to book one of them through a second remote service, and finally returns a response to the client. Note the ease and expressiveness with which we can declare asynchronous remote calls and have them executed concurrently (event loop style) without having to deal with threads and synchronization.

The Reactive Library

One advantage of the annotation programming model is flexible controller method signatures. Applications can choose from a wide range of supported method arguments and return values, and the framework adapts to the preferred usage style. This makes it easy to support multiple reactive libraries.

On both the servlet and reactive stacks, the use of Reactor or RxJava types is supported in controller method signatures. This is configurable, so other reactive libraries can be plugged in as well.

Functional Web Endpoints

In addition to expressing controller logic through annotated controllers, which is a common choice among Java web frameworks, Spring WebFlux also supports a lambda-based, lightweight, functional programming model for routing and handling requests.

Functional endpoints are very different from annotated controllers. When you use annotations, you describe to the framework what should be done and let it do as much work as it can on your behalf -- remember the Hollywood principle "don't call us, we'll call you"? By contrast, the functional programming model consists of a small set of helper utilities available to the application to drive request processing from start to finish. A short snippet of routing and handling a request would look something like the following:

```
RouterFunction<?> route = RouterFunctions.route(POST("/cars/{id}/boo
   request -> {
      Long carId = Long.valueOf(request.pathVariable("id"));
      Long id = ... ;  // Create booking

      URI location = URI.create("/car/" + carId + "/booking/" + id);
      return ServerResponse.created(location).build();
});
```

And here is one way to run it, for instance, on Netty with Reactor Netty:

```
HttpHandler handler = RouterFunctions.toHttpHandler(route);
ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(ha
HttpServer server = HttpServer.create("localhost", 8080);
server.startAndAwait(adapter);
```

Functional endpoints and reactive libraries go well together since they’re both built around functional, declarative style APIs.

Web Stack Architecture

Let’s take a closer look inside the servlet and reactive stacks.

The servlet stack is a classic servlet container and Servlet API, with Spring MVC as the web framework. In the beginning the Servlet API was built on a "thread-per-request" model, i.e. making a full pass through the filter-servlet chain on a single thread, blocking along the way as needed. Over time extra options were added to adapt to the changing needs and expectations of web applications:

1997 1.0 Initial version

...

2009 3.0 Async Servlet

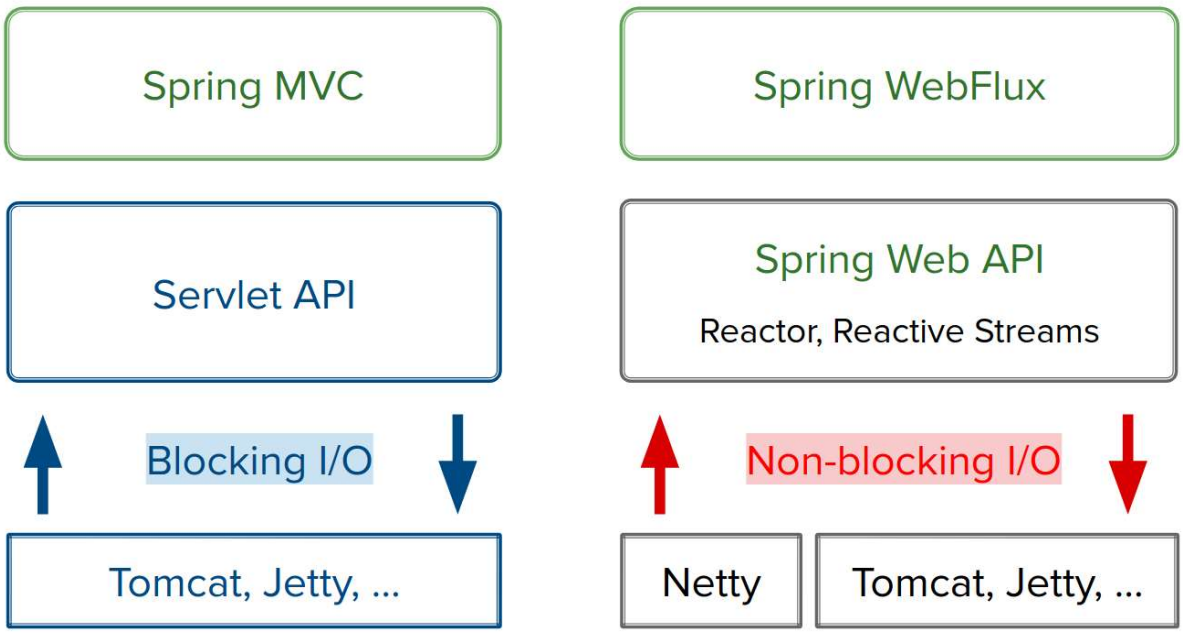
2013 3.1 Servlet Non-Blocking I/O

The Async Servlet feature in 3.0 made it possible to leave the response open for further processing after a full pass through the filter-servlet chain. Spring MVC builds extensively on this feature and it's what makes reactive return value types in annotated controllers possible.

Unfortunately, the non-blocking I/O features in 3.1 cannot be integrated into existing web framework contracts built around imperative, blocking semantics. That is why it is not supported in Spring MVC, and instead Spring WebFlux was created on a new foundation of non-blocking web framework contracts that support both the Servlet API and other servers.

The reactive stack can run with Tomcat, Jetty, Servlet 3.1 containers, Netty, and Undertow. Each server is adopted to a common Reactive Streams API for HTTP request handling. On top of that foundation is a slightly higher level, but still general purpose WebHandler API, comparable to the Servlet API but with asynchronous, non-blocking contracts.

Below is a diagram of the two stacks side by side:



Note that although Tomcat and Jetty are supported on both sides, they're used through different APIs in each. On the servlet stack they're used through the Servlet API with blocking I/O. On the reactive stack they're used through Servlet 3.1 non-blocking I/O without ever exposing the Servlet API -- many parts of which remain blocking and synchronous (e.g. request parameters, multipart requests, etc), for application use.

Reactive, Non-Blocking Back Pressure

Both the servlet and the reactive stacks support annotated controllers, however there is a crucial difference in the concurrency model and assumptions.

On the servlet stack, applications are allowed to block. That is why servlet containers use a large thread pool to absorb potential blocking in the application. This assumption is reflected in the `Filter` and `Servlet` contracts, both of which are imperative and return `void`, as well as in the blocking InputStream and OutputStream.

On the reactive stack applications must never block; since they are invoked in one of a small, fixed number of threads on the event loop, you would soon block the entire server. This assumption is reflected in the `WebFilter` and `WebHandler` contracts that return `Mono<Void>`, the Reactor type for 0..1 asynchronous values (in this no values, just success or error), as well as in the reactive types used for the request and response body.

The request body is accessed through `Flux<DataBuffer>`, the Reactor type for an asynchronous sequence, and that means we must be prepared to process entire chunks of data at a time, as they become available. This may sound scary but there are built-in codecs to transform the stream of bytes into a stream of Objects.

For example, we can upload a JSON stream from the client side:

```
// Pump out a new car every second

Flux<Car> body = Flux.interval(Duration.ofSeconds(1)).map(i -> new C

// Post the live streaming data to the server

WebClient.create().post()
        .uri("http://localhost:8081/cars")
        .contentType(MediaType.APPLICATION_STREAM_JSON)
```

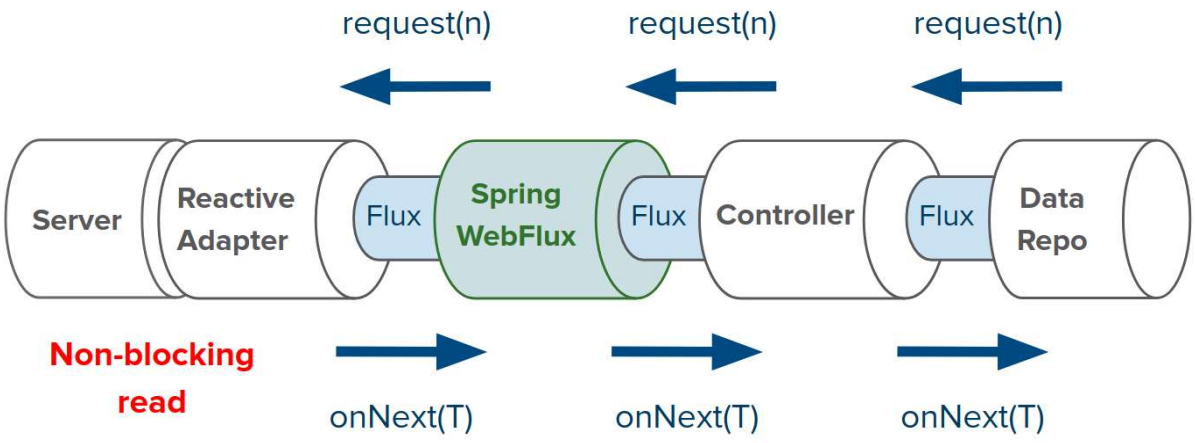
```
        .body(body, Car.class)
        .retrieve()
        .bodyToMono(Void.class)
        .block();
```

On the server side, a Spring WebFlux controllers can ingest the stream, and use a Spring Data reactive repository to insert it into a data store:

```
// Server posts the streamed data as it comes (see discussion below)

@PostMapping(path="/cars", consumes = "application/stream+json")
public Mono<Void> loadCars(@RequestBody Flux<Car> cars) {
    return repository.insert(cars).then();
}
```

The stream can continue for a long time, for days if needed. It's handled efficiently, without holding on to extra threads or memory. In this scenario both Spring WebFlux and Spring Data support reactive streams, so the above code expands into a processing pipeline with reactive streams back pressure signals flowing from the data store through to the HTTP runtime. The data store effectively controls the rate at which data chunks are read from the HTTP request and turned into Objects:



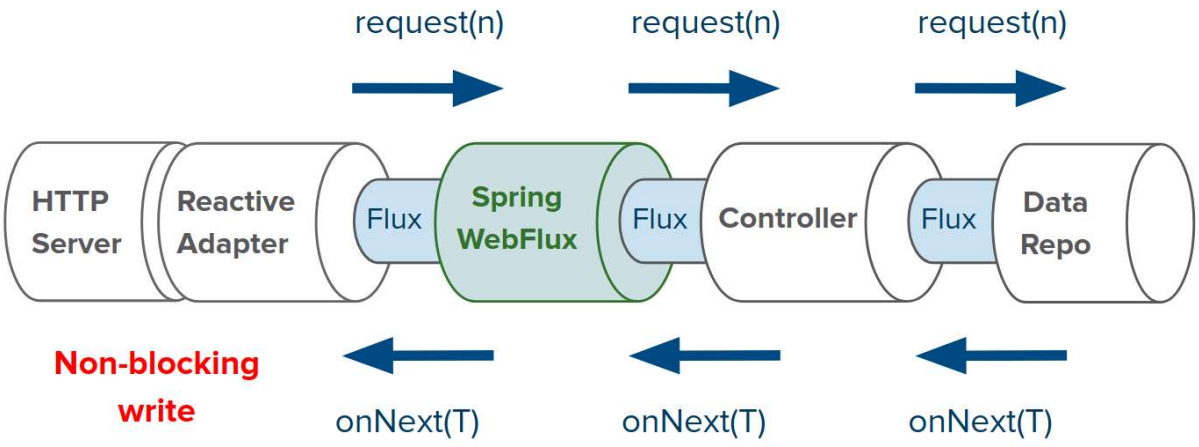
Let's assume the controller ingesting car data inserts it into a capped, tailable, MongoDB collection. Now other clients can request a JSON stream to observe the collection:

```
WebClient.create().get()
          .uri("http://localhost:8081/cars")
          .accept(MediaType.APPLICATION_STREAM_JSON)
          .retrieve()
          .bodyToFlux(Car.class)
          .doOnNext(car -> {
              logger.debug("Received " + car));
              //...
           })
          .block();
```

On the server side, the controller can serve the JSON stream as follows:

```
@GetMapping(path = "/cars", produces = "application/stream+json")
public Flux<Car> getCarStream() {
    return this.repository.findCars();
}
```

This time reactive streams back pressure signals flow in the opposite direction, from the HTTP runtime to the data store. The HTTP runtime effectively controls the rate at which Objects are fetched from the data store, serialized to JSON, and written as data chunks to the HTTP response:



The above scenario illustrates how natural it is to ingest or stream data on the reactive stack, by accepting or returning `Flux<Car>`. What about working with a regular (finite) collection? `Flux` supports any data sequence, whether finite or infinite, so not much needs to change.

We still return `Flux` and the web framework adapts accordingly by checking the media type.

The below renders a JSON array with content type "application/json":

```
@GetMapping(path = "/cars", produces = "application/json")
public Flux<Car> getCars() {
    return this.repository.findAll();
}
```

For "application/json", a non-streaming media type, the web framework assumes the `Flux` represents a finite collection and uses `Flux.collectoToList()` that requests all items, gathers them in a List, and then writes the collection to the response.

So far in this section we've talked about the reactive stack. The servlet stack relies on blocking I/O and therefore a non-blocking, or a streaming `@RequestBody` is not supported. It is possible however for the controller method to do asynchronous work, thanks to the Servlet 3.0 async request feature, and so a controller in Spring MVC can call reactive clients and return reactive types for response handling.

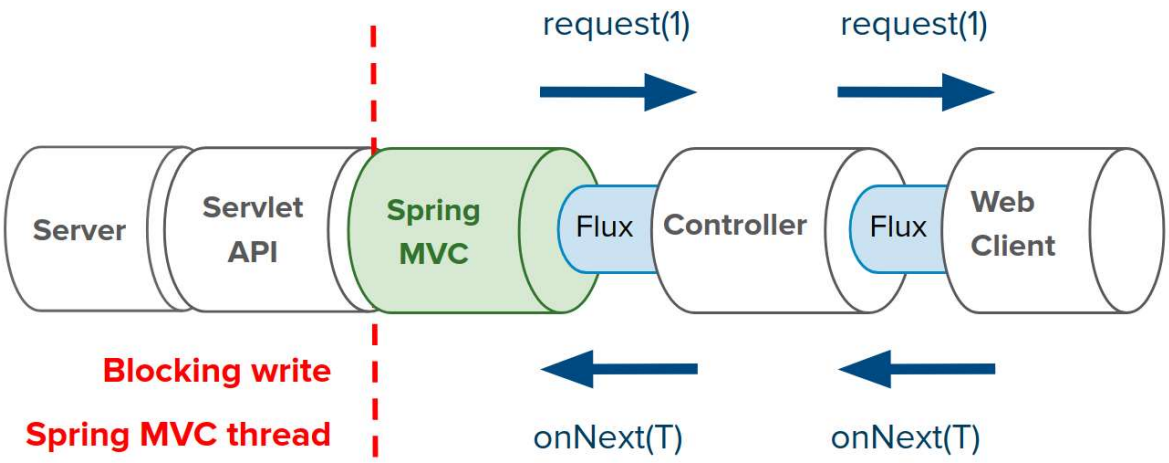
For "application/json", on the servlet stack, Spring MVC also collects Flux items to a List, and writes the result to the response as a JSON array:

```
@GetMapping("/cars")
public Flux<Car> getCars() {
    return this.repository.findAll();
}
```

For "application/stream+json" and other streaming media types, Spring MVC also applies reactive streams back pressure to the upstream source:

```
@GetMapping(path = "/cars", produces = "application/stream+json")
public Flux<Car> getCarStream() {
    return this.repository.findCars();
}
```

However, unlike the reactive stack, on the servlet stack writes are blocking, and performed in a separate thread, with the completion of the write used to signal demand to the upstream source:

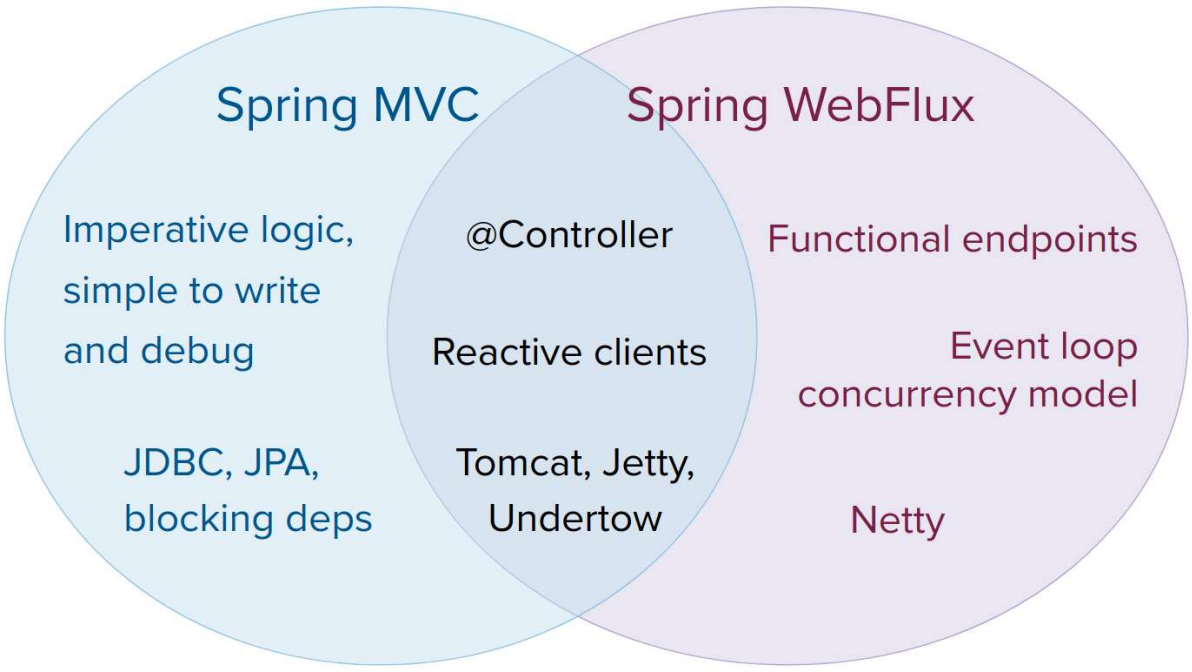


Effectively on the servlet stack, Spring MVC goes as far as possible to extend the benefits of reactive and asynchronous request handling to existing applications.

Making Choices

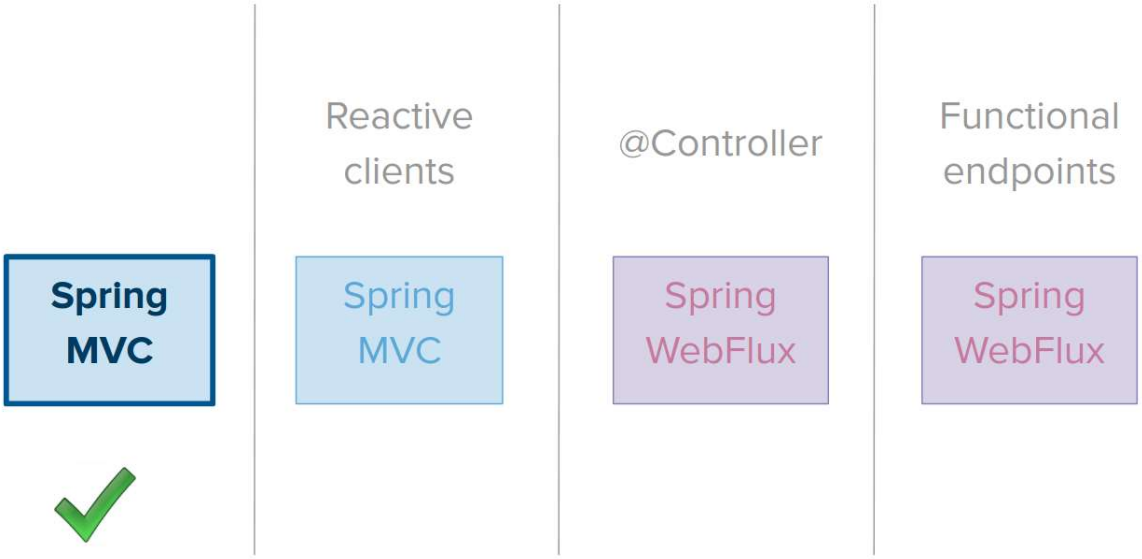
Spring MVC or Spring WebFlux, which should you use?

A perfectly valid question, but one that sets up an unsound dichotomy. Both are maintained, co-evolved, and available side by side. They are designed with consistency and continuity in mind, where ideas and feedback from one benefit the other. It is more accurate to think of it as being both Spring MVC and Spring WebFlux, together extending the range of possibilities:

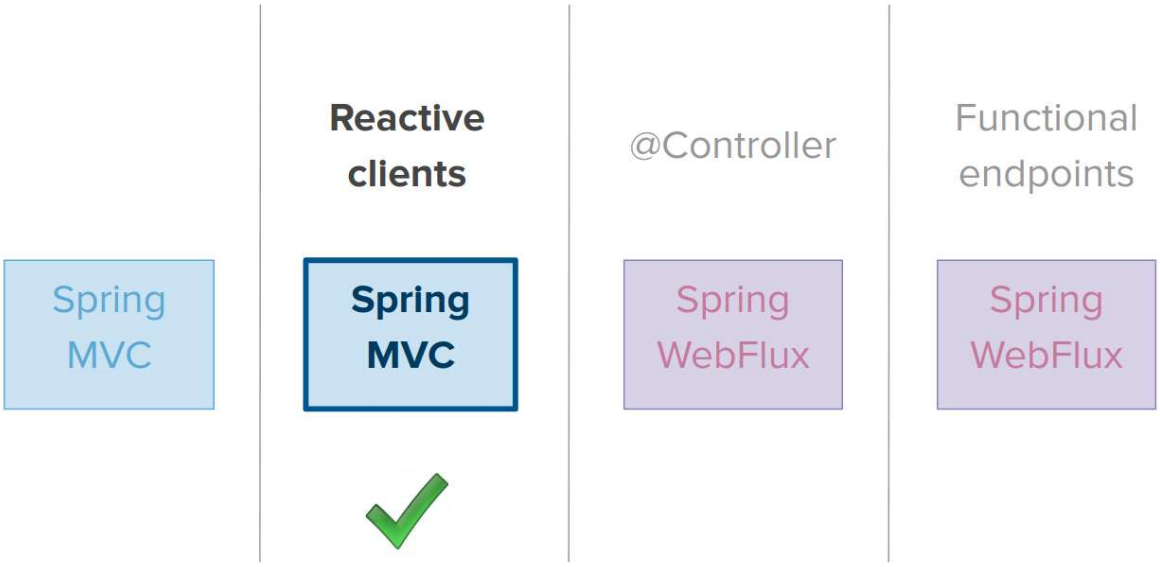


Spring MVC offers a simple, imperative model on a classic servlet stack foundation, for use with blocking or non-blocking dependencies. Spring WebFlux offers the benefits of event loop concurrency and an alternative, functional programming model. The common ground in the middle is where we aim for consistency and continuity.

If your application isn't suffering issues of scale, and the cost of horizontal scaling is within budget, then there is no need to change. Imperative is the easiest way to write code and I expect a majority of applications to remain closer to this end of the spectrum for the foreseeable future:



Of course, imperative is easy until it's not. It is common in modern applications to make calls to remote services. In some cases it may be okay to do that in imperative style, synchronously, but if you want to scale more with less hardware, you'll need to embrace concurrency and asynchronicity. Once you're in that world, reactive programming offers a comprehensive and effective solution. I expect many applications will use the WebClient and other reactive libraries (e.g. Spring Data reactive repositories) in Spring MVC because it requires relatively little change and provides substantial benefits:



The reactive stack and Spring WebFlux may appeal in a variety of ways. Perhaps you're looking for the kind of scale that traditional blocking I/O cannot provide for scenarios with high concurrency and variable latency, or perhaps the price of horizontal vs vertical scale is too high. For Spring Cloud Gateway for example, the choice to build on Spring WebFlux is obvious, as it needs to optimize for high concurrency and efficiency.

The reactive stack, with its reactive processing pipeline, is exceptionally well suited for streaming scenarios, a requirement in many applications. It is supported for both request and response content, with reactive streams back pressure, and with non-blocking writes.

Some have chosen the reactive stack simply to provide a functional web programming model that is only available in Spring WebFlux. Functional endpoints have been very popular with Kotlin applications, for which WebFlux provides a Kotlin DSL for request routing. Due to its simplicity and transparency (a dozen classes or so), the functional web programming model may be a good fit for relatively small, focused microservices.

The repository demo-reactive-spring contains the source code used in this article.

About the Author

As a committer on the Spring Framework team, Rossen Stoyanchev's contributions span both the evolution of Spring MVC over three generations, and the creation of Spring WebFlux from inception to release.

6

Please see https://www.infoq.com for the latest version of this information.