Using Spring WebFlux

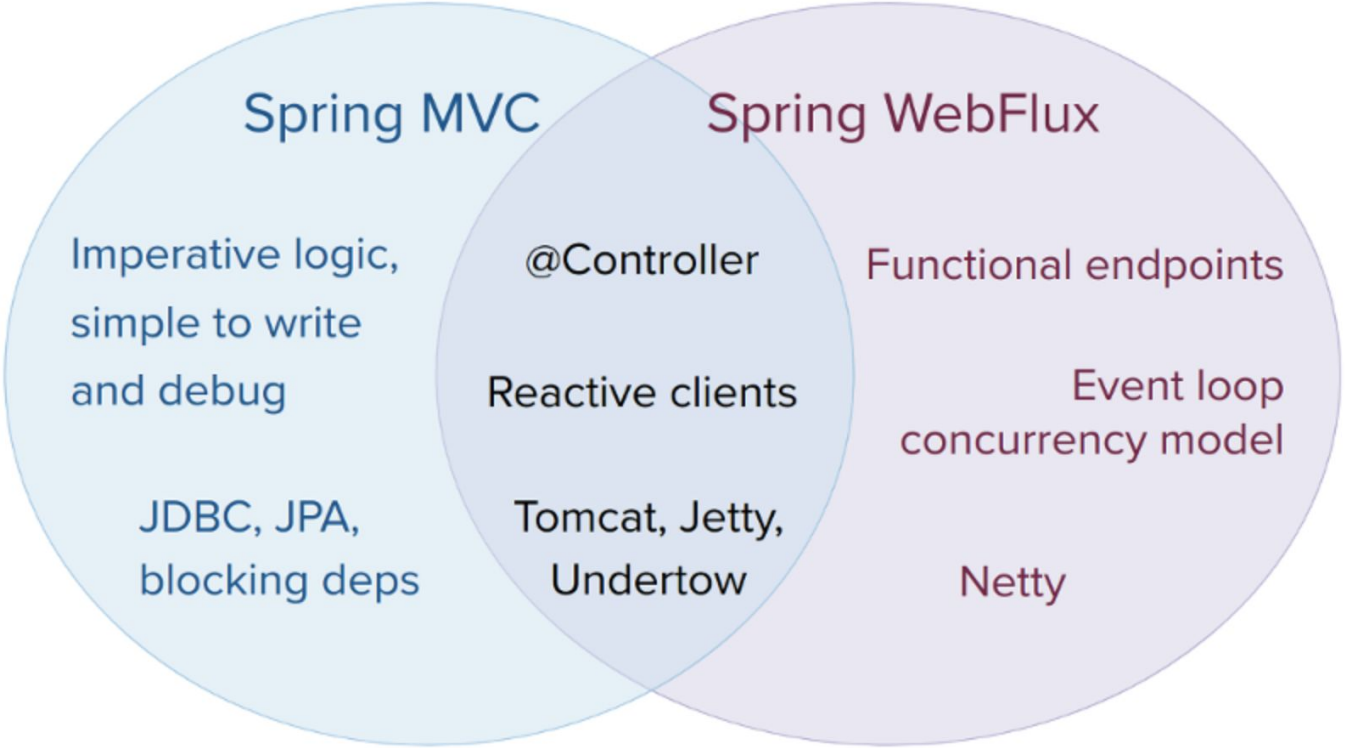
Zoltan Altfatter

Spring WebFlux

Spring WebFlux supports 2 distinct programming models

- Annotation based like **@Controller**
  - Spring MVC and WebFlux controllers support reactive return types
  - WebFlux also supports reactive **@RequestBody** arguments
- Functional with Java 8 lambda style
  - Lightweight and functional
  - Big difference to annotation based on is that the application is in charge of the request handling

Spring WebFlux



Spring WebFlux

Servers

- Tomcat, Jetty
- Servlet 3.1+ container (building a bridge between non-blocking IO to reactive streams with backpressure)
- Non Servlet runtimes: Netty, Undertow
- Spring WebFlux does not have built-in support to start or stop a server
- Spring Boot has WebFlux starter that automates this. By default uses Netty, but easy to switch to other.

HttpHandler

- Lowest level contract for reactive HTTP request handling that serves as a common denominator across different runtimes.

Reactive Http Server Abstraction

```
public interface HttpHandler {

  Mono<Void> handle(ServerHttpRequest request, ServerHttpResponse response);

}

public interface Servlet {

   public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException;

}

ServerHttpRequest -> Flux<DataBuffer> getBody();

ServerHttpResponse -> Mono<Void> writeWith(Publisher<? extends DataBuffer> body);

HttpMessageReader, HttpMessageWriter -> Flux<DataBuffer> to/from Flux<T>, Mono<T>

Encoder, Decoder -> are contracts for encoding and decoding content, independent of HTTP
```

Concurreny Model

Spring MVC

- applications block the current thread (remote call, database access)
- Servlet containers use large thread pool

Spring WebFlux

- applications do not block
- use small, fixed-size thread pool (event loop workers) to handle request
- What if you need to use blocking library? There is **publishOn** operator to continue processing on different thread.

WebFlux Config vs Spring MVC config

@EnableWebFlux - registers number of Spring WebFlux infrastructure beans (see **WebFluxConfigurationSupport**)

```
@EnableWebFlux
@Configuration
public class WebConfig {
}
```

```
@Configuration
@EnableWebMvc
public class WebConfig {
}
```

- Spring Boot provides auto-configuration for Spring WebFlux (see **WebFluxAutoConfiguration**)
- Keep additional Spring Boot WebFlux config and want to add additional WebFlux configuration you just need to add **@Configuration** of class type **WebFluxConfigurer** without **@EnableWebFlux**

```
@Configuration
public class WebConfig implements WebFluxConfigurer {
   // implement configuration methods
}
```

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
   //implement configuration methods
}
```

- To take complete control of Spring WebFlux use:

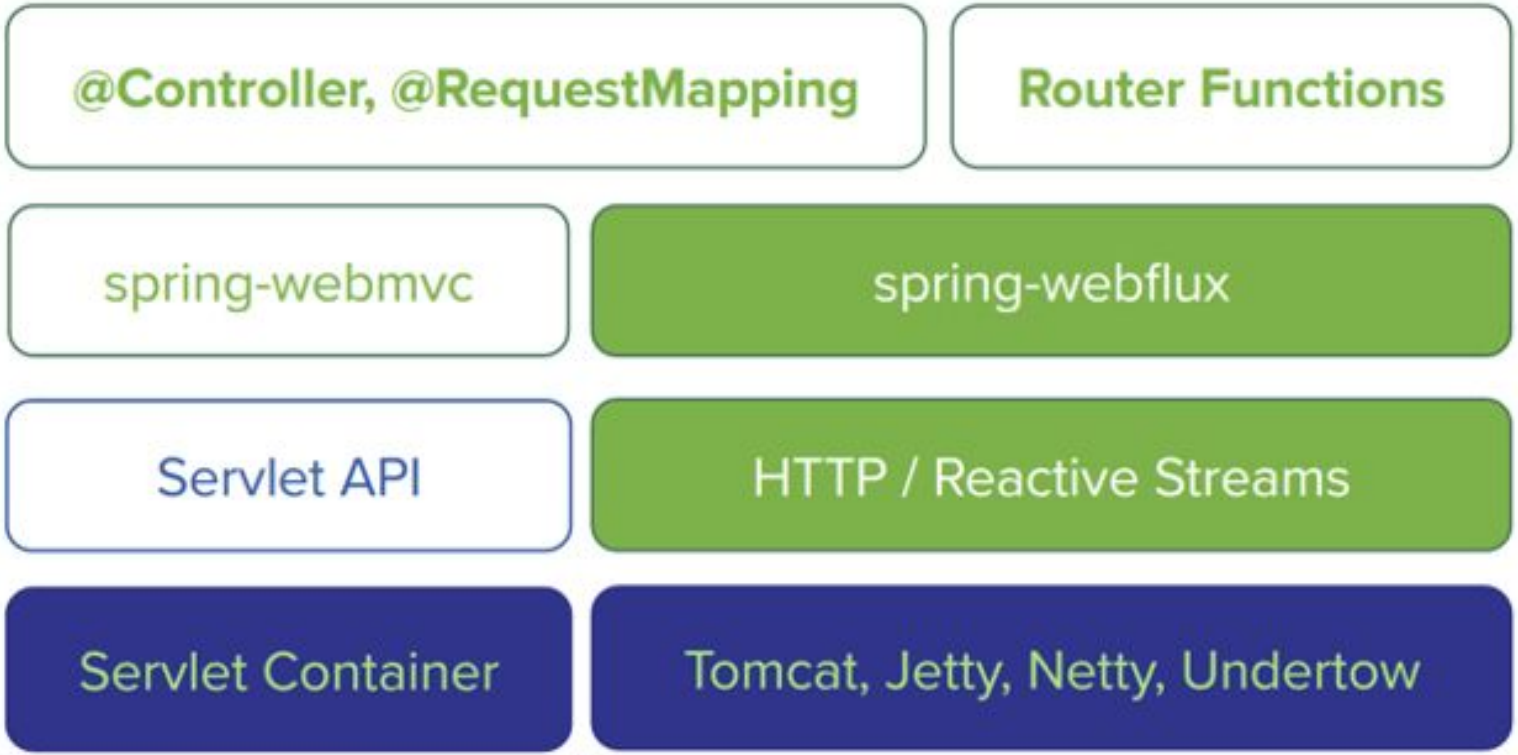
```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {
   // implement configuration methods
}
```

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
   //implement configuration methods
}
```

Reactive Web Controller

```
@RestController
public class CustomerController {

   @GetMapping("/customers/{id}")
   public Mono<Customer> getCustomer(@PathVariable Long id) {
       return customerRepository.findById(id);
   }

   @GetMapping("/customers")
   public Flux<Customer> getCustomers() {
       return customerRepository.findAll();
   }

   @PostMapping("/customers")
   public Mono<Void> addCustomer(@RequestBody Customer customer) {
       return customerRepository.save(customer);
   }

}
```

Spring WebFlux



WebFlux.fn

Design Goals:

- Functional Style(java.util.function, java.util.stream)
- Fully reactive (based on Reactor)
- More library, less framework
- No reflection

WebFlux.fn

```
@FunctionalInterface
public interface HandlerFunction<T extends ServerResponse> {

   Mono<T> handle(ServerRequest request);

}

@FunctionalInterface
public interface RouterFunction<T extends ServerResponse> {

  Mono<HandlerFunction<T>> route(ServerRequest request);

}

@FunctionalInterface
public interface RequestPredicate {

  boolean test(ServerRequest request);
}


@FunctionalInterface
public interface HandlerFilterFunction<T extends ServerResponse, R extends ServerResponse> {

  Mono<R> filter(ServerRequest request, HandlerFunction<T> next);
}
```

WebFlux.fn

```
@Bean
public RouterFunction<ServerResponse> customerRouterFunction(CustomerHandler customerHandler) {
   return route(GET("/customers/{id}"), customerHandler::getCustomer)
           .andRoute(GET("/customers"), customerHandler::getCustomers);
}
```

New in Spring 5.1 builder style:

```
@Bean
public RouterFunction<ServerResponse> customerRouterFunction2(CustomerHandler customerHandler) {
   return route()
           .GET("/customers/{id}", accept(APPLICATION_JSON), customerHandler::getCustomer)
           .GET("/customers", accept(APPLICATION_JSON), customerHandler::getCustomers)
           .build();
}
```

How can we add value to existing applications?

RestTemplate

- Created 10 years ago, for Java 1.5
- Template methods, 24 to start, 40+ today
- Synchronous API
- No streaming

WebClient

- Functional, fluent API for Java 8+
- Async, non-blocking by design
- Reactive, declarative
- Streaming

WebClient

```
WebClient client = WebClient.create("http://example.com");

Mono<Customer> customer = client.get().
       uri("/customers/{id}", id).accept(MediaType.APPLICATION_JSON)
       .retrieve().bodyToMono(Customer.class);

Flux<Customer> customers = client.get().
       uri("/customers", id).accept(MediaType.TEXT_EVENT_STREAM)
       .retrieve().bodyToFlux(Customer.class);
```

WebClient
- reactive, non-blocking client for HTTP requests
- Internally delegates to an HTTP client library, by default uses Reactor Netty
- Compared to RestTemplate it has a functional API

```
WebClient.create()
WebClient.create(String baseUrl)
```

`WebClient.builder()` - to customize headers, filters, client connector, etc..

WebClient with Spring Boot

```java
@Service
public class CustomerService {

   private final WebClient webClient;

   public CustomerService(WebClient.Builder builder) {
       this.webClient = builder
               .filter(basicAuthentication("user", "password"))
               .baseUrl("http://example.org").build();
   }

   public Mono<Customer> getCustomer(String name) {
       return webClient.get().uri("/customers/{name}").retrieve().bodyToMono(Customer.class);
   }
}
```

Spring Boot creates and pre-configures a **WebClient.Builder** which can be injected into your components (see **WebClientAutoConfiguration**)

Spring Boot also auto-detects which **ClientHttpConnector** to use to drive **WebClient** depeding on the libraries available on the classpath.

Spring MVC Reactive Support

- Controller can return **Flux**, **Mono**, RxJava types, etc
- Decoupled from container thread (same mechanism as DeferredResult)
- Built on Servlet 3.0 async support
- **Mono** -> **DeferredResult**
- **Flux / non-streaming** -> **DeferredResult**<List<T>>
  - if MediaType is **application/json** we aggregate them in a List
- **Flux / streaming** -> **ResponseBodyEmitter** with **back pressure**
  - if MediaType is **application/stream+json**

WebClient in Spring MVC controller

1. Concise declaration of remote calls
2. No dealing with threads
3. Efficient scale and execution

Best Practices

- Don’t mix blocking and non-blocking APIs
- Identify vertical non-blocking slices
- Don’t put non-blocking code behind synchronous APIs
- Keep the flight going instead of landing :)
- In the end you want to compose a single, **deferred** request handling chain
- Don’t use **block()**, **subscribe()**
- Let Spring MVC handle it

WebTestClient

- end-to-end tests for HTTP
- thin layer over WebClient for testing
- once you get the data you have verification methods in a functional style

Further Resources

Good article:
https://www.infoq.com/articles/Servlet-and-Reactive-Stacks-Spring-Framework-5

Spring Framework Web documentation updated:

https://docs.spring.io/spring/docs/5.1.1.RELEASE/spring-framework-reference/