# Spring WebFlux

michael.schaffler@ciit.at

# 1. Introduction to Spring WebFlux

## What is Reactive Programming?

- non-blocking applications that are asynchronous and event-driven and require a small number of threads to scale vertically (i.e. within the JVM) rather than horizontally (i.e. through clustering)
- A key aspect of reactive applications is the concept of backpressure which is a mechanism to ensure producers don’t overwhelm consumers
- Reactive programming also leads to a major shift from imperative to declarative async composition of logic. It is comparable to writing blocking code vs using the CompletableFuture from Java 8 to compose follow-up actions via lambda expressions

# Reactive API and Building Blocks

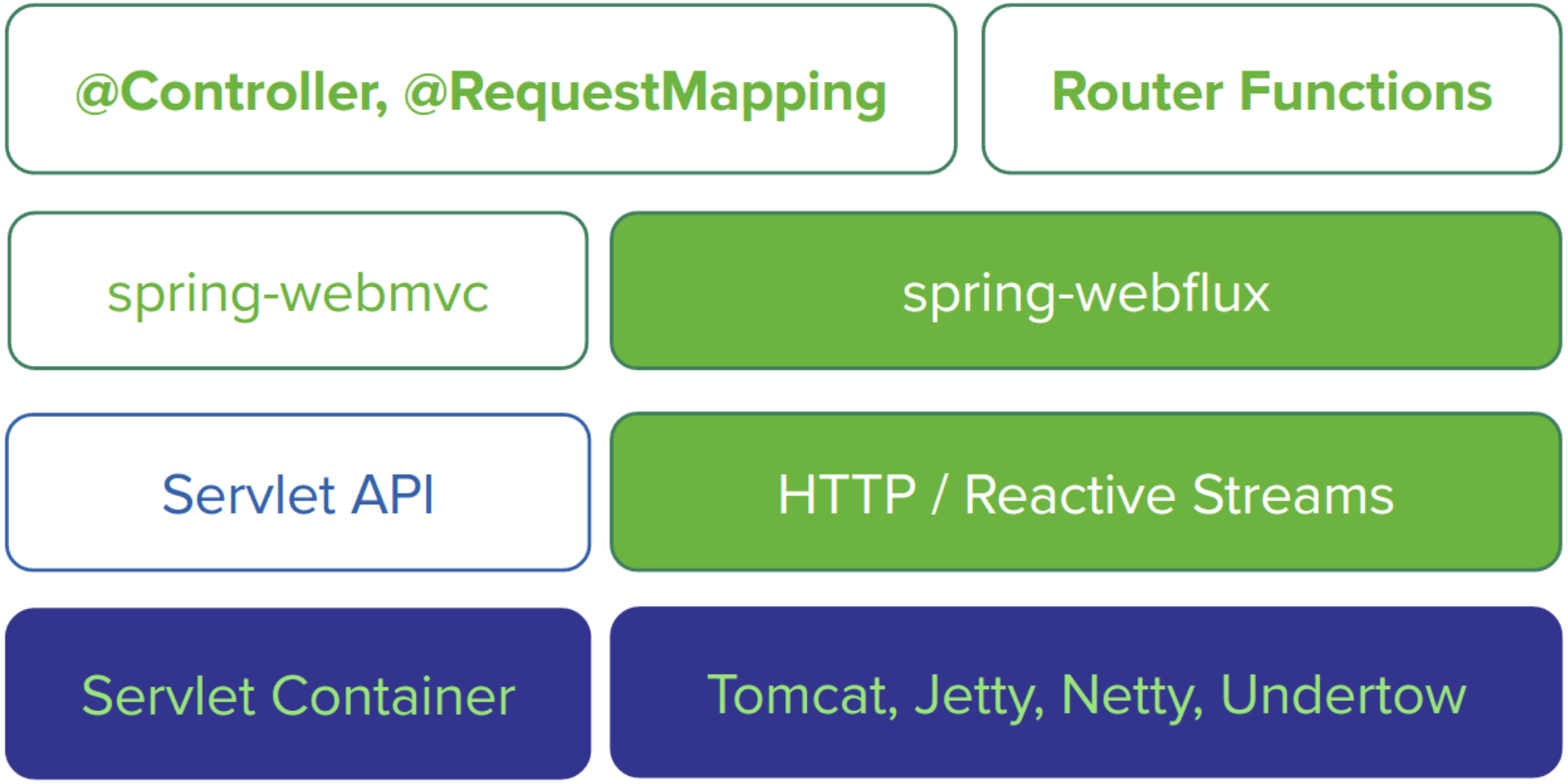
- Spring 5 uses Reactive Streams as the contract for communicating backpressure across async components and libraries
- Reactive Streams is a specification created through industry collaboration that has also been adopted in Java 9 as java.util.concurrent.Flow
- The Spring Framework uses Reactor internally for its own reactive support. Reactor is a Reactive Streams implementation that further extends the basic Reactive Streams Publisher contract with the Flux and Mono composable API types to provide declarative operations
- The Spring Framework exposes Flux and Mono in many of its own reactive APIs. At the application level Spring provides choice and fully supports the use of RxJava

# Spring WebFlux Module

- Spring Framework 5 includes a new spring-webflux module. The module contains support for reactive HTTP and WebSocket clients as well as for reactive server web applications including REST, HTML browser, and WebSocket style interactions.
- **Server Side** - 2 distinct programming models
  - Annotation-based with @Controller and the other annotations supported also with Spring MVC
  - Functional, Java 8 lambda style routing and handling

  Both programming models are executed on the same reactive foundation that adapts non-blocking HTTP runtimes to the Reactive Streams API

# **Server side** stack



# Annotation-based Programming Model

- The same @Controller programming model and the same annotations used in Spring MVC are also supported in WebFlux
- The main difference is that the underlying core, framework contracts — i.e. HandlerMapping, HandlerAdapter, are non-blocking and operate on the reactive ServerHttpRequest and ServerHttpResponse

# Annotation-based Programming Model (cont.)

```java
@RestController
public class PersonController {

	private final PersonRepository repository;

	public PersonController(PersonRepository repository) {
		this.repository = repository;
	}

	@PostMapping("/person")
	Mono<Void> create(@RequestBody Publisher<Person> personStream) {
		return this.repository.save(personStream).then();
	}

	@GetMapping("/person")
	Flux<Person> list() {
		return this.repository.findAll();
	}

	@GetMapping("/person/{id}")
	Mono<Person> findById(@PathVariable String id) {
		return this.repository.findOne(id);
	}
}
```

# Functional Programming Model: **HandlerFunctions**

- Incoming HTTP requests are handled by a HandlerFunction, which is essentially a function that takes a ServerRequest and returns a *Mono<ServerResponse>.* The annotation counterpart to a handler function would be a method with *@RequestMapping*.
- ServerRequest gives access to various HTTP request elements: the method, URI, query parameters, and — through the separate ServerRequest.Headers interface — the headers. Access to the body is provided through the body methods
- Similarly, ServerResponse provides access to the HTTP response. Since it is immutable, you create a ServerResponse with a builder. The builder allows you to set the response status, add response headers, and provide a body

# HandlerFunctions

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.BodyInserters.fromObject;

public class PersonHandler {

        private final PersonRepository repository;

        public PersonHandler(PersonRepository repository) {
                this.repository = repository;
        }

        public Mono<ServerResponse> listPeople(ServerRequest request) { ❶
                Flux<Person> people = repository.allPeople();
                return ServerResponse.ok().contentType(APPLICATION_JSON).body(people, Person.class);
        }

        public Mono<ServerResponse> createPerson(ServerRequest request) { ❷
                Mono<Person> person = request.bodyToMono(Person.class);
                return ServerResponse.ok().build(repository.savePerson(person));
        }

        public Mono<ServerResponse> getPerson(ServerRequest request) { ❸
                int personId = Integer.valueOf(request.pathVariable("id"));
                Mono<ServerResponse> notFound = ServerResponse.notFound().build();
                Mono<Person> personMono = this.repository.getPerson(personId);
                return personMono
                                .then(person -> ServerResponse.ok().contentType(APPLICATION_JSON).body(fromObject(person)))
                                .otherwiseIfEmpty(notFound);
        }
}
```

❶ `listPeople` is a handler function that returns all `Person` objects found in the repository as JSON.

❷ `createPerson` is a handler function that stores a new `Person` contained in the request body. Note that `PersonRepository.savePerson(Person)` returns `Mono<Void>`: an empty Mono that emits a completion signal when the person has been read from the request and stored. So we use the `build(Publisher<Void>)` method to send a response when that completion signal is received, i.e. when the `Person` has been saved.

❸ `getPerson` is a handler function that returns a single person, identified via the path variable `id`. We retrieve that `Person` via the repository, and create a JSON response if it is found. If it is not found, we use `otherwiseIfEmpty(Mono<T>)` to return a 404 Not Found response.

# Functional Programming Model: **RouterFunctions**

- Incoming requests are routed to handler functions with a RouterFunction, which is a function that takes a ServerRequest, and returns a Mono<HandlerFunction>
- If a request matches a particular route, a handler function is returned; otherwise it returns an empty Mono. The RouterFunction has a similar purpose as the @RequestMapping annotation in @Controller classes
- Typically, you do not write router functions yourself, but rather use RouterFunctions.route(RequestPredicate, HandlerFunction) to create one using a request predicate and handler function

# Functional Programming Model: **RouterFunctions**

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.RequestPredicates.*;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> personRoute =
        route(GET("/person/{id}").and(accept(APPLICATION_JSON)), handler::getPerson)
                .andRoute(GET("/person").and(accept(APPLICATION_JSON)), handler::listPeople)
                .andRoute(POST("/person").and(contentType(APPLICATION_JSON)), handler::createPerson);
```

Besides router functions, you can also compose request predicates, by calling `RequestPredicate.and(RequestPredicate)` or `RequestPredicate.or(RequestPredicate)`. These work as expected: for `and` the resulting predicate matches if **both** given predicates match; `or` matches if **either** predicate does. Most of the predicates found in `RequestPredicates` are compositions. For instance, `RequestPredicates.GET(String)` is a composition of `RequestPredicates.method(HttpMethod)` and `RequestPredicates.path(String)`.

# Running a Server

```
RouterFunction<ServerResponse> route = ...
HttpHandler httpHandler = RouterFunctions.toHttpHandler(route);
HttpServlet servlet = new ServletHttpHandlerAdapter(httpHandler);
Tomcat server = new Tomcat();
Context rootContext = server.addContext("", System.getProperty("java.io.tmpdir"));
Tomcat.addServlet(rootContext, "servlet", servlet);
rootContext.addServletMapping("/", "servlet");
tomcatServer.start();
```

- The HttpHandler allows you to run on a wide variety of reactive runtimes: Reactor Netty, RxNetty, Servlet 3.1+, and Undertow

# HandlerFilterFunction

```
import static org.springframework.http.HttpStatus.UNAUTHORIZED;

SecurityManager securityManager = ...
RouterFunction<ServerResponse> route = ...

RouterFunction<ServerResponse> filteredRoute =
        route.filter(request, next) -> {
                if (securityManager.allowAccessTo(request.path())) {
                        return next.handle(request);
                }
                else {
                        return ServerResponse.status(UNAUTHORIZED).build();
                }
  });
```

# Client Side

- WebFlux includes a functional, reactive WebClient that offers a fully non-blocking and reactive alternative to the RestTemplate

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

Mono<Account> account = client.get()
                .url("/accounts/{id}", 1L)
                .accept(APPLICATION_JSON)
                .exchange(request)
                .then(response -> response.bodyToMono(Account.class));
```

The `AsyncRestTemplate` also supports non-blocking interactions. The main difference is it can't support non-blocking streaming, like for example Twitter one, because fundamentally it's still based and relies on `InputStream` and `OutputStream`.

# Request and Response Body Conversion

- The spring-core module provides reactive Encoder and Decoder contracts that enable the serialization of a Flux of bytes to and from typed objects. The spring-web module adds JSON (Jackson) and XML (JAXB) implementations for use in web applications as well as others for SSE streaming and zero-copy file transfer.
- the request body can be one of the following way and it will be decoded automatically in both the annotation and the functional programming models:

- `Account account` — the account is deserialized without blocking before the controller is invoked.
- `Mono<Account> account` — the controller can use the `Mono` to declare logic to be executed after the account is deserialized
- `Single<Account> account` — same as with `Mono` but using RxJava
- `Flux<Account> accounts` — input streaming scenario.
- `Observable<Account> accounts` — input streaming with RxJava.

# Request and Response Body Conversion (cont.)

The response body can be one of the following:

- `Mono<Account>` — serialize without blocking the given Account when the `Mono` completes.
- `Single<Account>` — same but using RxJava.
- `Flux<Account>` — streaming scenario, possibly SSE depending on the requested content type.
- `Observable<Account>` — same but using RxJava `Observable` type.
- `Flowable<Account>` — same but using RxJava 2 `Flowable` type.
- `Flux<ServerSentEvent>` — SSE streaming.
- `Mono<Void>` — request handling completes when the `Mono` completes.
- `Account` — serialize without blocking the given Account; implies a synchronous, non-blocking controller method.
- `void` — specific to the annotation-based programming model, request handling completes when the method returns;

# Reactive WebSocket Support

```java
@Bean
public HandlerMapping webSocketMapping() {
        Map<String, WebSocketHandler> map = new HashMap<>();
        map.put("/foo", new FooWebSocketHandler());
        map.put("/bar", new BarWebSocketHandler());

        SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
        mapping.setUrlMap(map);
        return mapping;
}

@Bean
public WebSocketHandlerAdapter handlerAdapter() {
        return new WebSocketHandlerAdapter();
}
```

On the client side create a `WebSocketClient` for one of the supported libraries listed above:

```java
WebSocketClient client = new ReactorNettyWebSocketClient();
client.execute("ws://localhost:8080/echo"), session -> {... }).blockMillis(5000);
```

# Testing

- The spring-test module includes a WebTestClient that can be used to test WebFlux server endpoints with or without a running server.

- Tests without a running server are comparable to MockMvc from Spring MVC where mock request and response are used instead of connecting over the network using a socket. The WebTestClient however can also perform tests against a running server

# Manual Bootstrapping

- For dependencies start with spring-webflux and spring-context. Then add jackson-databind and io.netty:netty-buffer for JSON support. Lastly add the dependencies for one of the supported runtimes, e.g. Tomcat, Jetty, Undertow…

For the **annotation-based programming model** bootstrap with:

```
ApplicationContext context = new AnnotationConfigApplicationContext(DelegatingWebFluxConfiguration.class);  // (1)
HttpHandler handler = DispatcherHandler.toHttpHandler(context);  // (2)
```

The above loads default Spring Web framework configuration (1), then creates a `DispatcherHandler`, the main class driving request processing (2), and adapts it to `HttpHandler` — the lowest level Spring abstraction for reactive HTTP request handling.

For the **functional programming model** bootstrap as follows:

```
ApplicationContext context = new AnnotationConfigApplicationContext(); // (1)
context.registerBean(FooBean.class, () -> new FooBeanImpl()); // (2)
context.registerBean(BarBean.class); // (3)

HttpHandler handler = WebHttpHandlerBuilder
                .webHandler(RouterFunctions.toHttpHandler(...))
                .applicationContext(context)
                .build(); // (4)
```

The above creates an `AnnotationConfigApplicationContext` instance (1) that can take advantage of the new functional bean registration API (2) to register beans using a Java 8 `Supplier` or just by specifying its class (3). The `HttpHandler` is created using `WebHttpHandlerBuilder` (4).

# Manual Bootstrapping

- For dependencies start with spring-webflux and spring-context. Then add jackson-databind and io.netty:netty-buffer for JSON support. Lastly add the dependencies for one of the supported runtimes, e.g. Tomcat, Jetty, Undertow…

For the **annotation-based programming model** bootstrap with:

```
ApplicationContext context = new AnnotationConfigApplicationContext(DelegatingWebFluxConfiguration.class);  // (1)
HttpHandler handler = DispatcherHandler.toHttpHandler(context);  // (2)
```

The above loads default Spring Web framework configuration (1), then creates a `DispatcherHandler`, the main class driving request processing (2), and adapts it to `HttpHandler` — the lowest level Spring abstraction for reactive HTTP request handling.

For the **functional programming model** bootstrap as follows:

```
ApplicationContext context = new AnnotationConfigApplicationContext(); // (1)
context.registerBean(FooBean.class, () -> new FooBeanImpl()); // (2)
context.registerBean(BarBean.class); // (3)

HttpHandler handler = WebHttpHandlerBuilder
                .webHandler(RouterFunctions.toHttpHandler(...))
                .applicationContext(context)
                .build(); // (4)
```

The above creates an `AnnotationConfigApplicationContext` instance (1) that can take advantage of the new functional bean registration API (2) to register beans using a Java 8 `Supplier` or just by specifying its class (3). The `HttpHandler` is created using `WebHttpHandlerBuilder` (4).

# Manual Bootstrapping

- For dependencies start with spring-webflux and spring-context. Then add jackson-databind and io.netty:netty-buffer for JSON support. Lastly add the dependencies for one of the supported runtimes, e.g. Tomcat, Jetty, Undertow…

For the **annotation-based programming model** bootstrap with:

```
ApplicationContext context = new AnnotationConfigApplicationContext(DelegatingWebFluxConfiguration.class);  // (1)
HttpHandler handler = DispatcherHandler.toHttpHandler(context);  // (2)
```

The above loads default Spring Web framework configuration (1), then creates a `DispatcherHandler`, the main class driving request processing (2), and adapts it to `HttpHandler` — the lowest level Spring abstraction for reactive HTTP request handling.

For the **functional programming model** bootstrap as follows:

```
ApplicationContext context = new AnnotationConfigApplicationContext(); // (1)
context.registerBean(FooBean.class, () -> new FooBeanImpl()); // (2)
context.registerBean(BarBean.class); // (3)

HttpHandler handler = WebHttpHandlerBuilder
                .webHandler(RouterFunctions.toHttpHandler(...))
                .applicationContext(context)
                .build(); // (4)
```

The above creates an `AnnotationConfigApplicationContext` instance (1) that can take advantage of the new functional bean registration API (2) to register beans using a Java 8 `Supplier` or just by specifying its class (3). The `HttpHandler` is created using `WebHttpHandlerBuilder` (4).

# Spring WebFlux

michael.schaffler@ciit.at