Web on Reactive Stack

Version 5.0.0.RELEASE

Table of Contents

This part of the documentation covers support for reactive stack, web applications built on a Reactive Streams API to run on top of non-blocking servers such as Netty, Undertow, and Servlet 3.1+ containers. Individual chapters cover Spring WebFlux and its functional programming model. For Servlet stack, web applications, to go Web on Servlet Stack.

Chapter 1. Spring WebFlux

1.1. Introduction

The original web framework included in the Spring Framework, Spring Web MVC, was purpose built for the Servlet API and Servlet containers. The reactive stack, web framework, Spring WebFlux, was added later in version 5.0. It is fully non-blocking, supports Reactive Streams back pressure, and runs on servers such as Netty, Undertow, and Servlet 3.1+ containers.

Both web frameworks mirror the names of their source modules spring-webmvc and spring-webflux and co-exist side by side in the Spring Framework. Each module is optional. Applications may use one or the other module, or in some cases both — e.g. Spring MVC controllers with the reactive `WebClient`.

In addition to the web framework, Spring WebFlux also provides a WebClient for performing HTTP requests, a `WebTestClient` for testing web endpoints, and also client and server reactive, WebSocket support.

1.1.1. Why a new web framework?

Part of the answer is the need for a non-blocking web stack to handle concurrency with a small number of threads and scale with less hardware resources. Servlet 3.1 did provide an API for non-blocking I/O. However, using it leads away from the rest of the Servlet API where contracts are synchronous (`Filter`, `Servlet`) or blocking (`getParameter`, `getPart`). This was the motivation for a new common API to serve as a foundation across any non-blocking runtime. That is important because of servers such as Netty that are well established in the async, non-blocking space.

The other part of the answer is functional programming. Much like the addition of annotations in Java 5 created opportunities — e.g. annotated REST controllers or unit tests, the addition of lambda expressions in Java 8 created opportunities for functional APIs in Java. This is a boon for non-blocking applications and continuation style APIs — as popularized by `CompletableFuture` and ReactiveX, that allow declarative composition of asynchronous logic. At the programming model level Java 8 enabled Spring WebFlux to offer functional web endpoints alongside with annotated controllers.

1.1.2. Reactive: what and why?

We touched on non-blocking and functional but why reactive and what do we mean?

The term "reactive" refers to programming models that are built around reacting to change — network component reacting to I/O events, UI controller reacting to mouse events, etc. In that sense non-blocking is reactive because instead of being blocked we are now in the mode of reacting to notifications as operations complete or data becomes available.

There is also another important mechanism that we on the Spring team associate with "reactive" and that is non-blocking back pressure. In synchronous, imperative code, blocking calls serve as a natural form of back pressure that forces the caller to wait. In non-blocking code it becomes important to control the rate of events so that a fast producer does not overwhelm its destination.

Reactive Streams is a small spec, also adopted in Java 9, that defines the interaction between asynchronous components with back pressure. For example a data repository — acting as Publisher, can produce data that an HTTP server — acting as Subscriber, can then write to the response. The main purpose of Reactive Streams is to allow the subscriber to control how fast or how slow the publisher will produce data.

Common question: what if a publisher can't slow down?
The purpose of Reactive Streams is only to establish the mechanism and a boundary. If a publisher can't slow down then it has to decide whether to buffer, drop, or fail.

1.1.3. Reactive API

Reactive Streams plays an important role for interoperability. It is of interest to libraries and infrastructure components but less useful as an application API because it is too low level. What applications need is a higher level and richer, functional API to compose async logic — similar to the Java 8 `Stream` API but not only for collections. This is the role that reactive libraries play.

Reactor is the reactive library of choice for Spring WebFlux. It provides the Mono and Flux API types to work on data sequences of 0..1 and 0..N through a rich set of operators aligned with the ReactiveX vocabulary of operators. Reactor is a Reactive Streams library and therefore all of its operators support non-blocking back pressure. Reactor has a strong focus on server-side Java. It is developed in close collaboration with Spring.

WebFlux requires Reactor as a core dependency but it is interoperable with other reactive libraries via Reactive Streams. As a general rule WebFlux APIs accept a plain `Publisher` as input, adapt it to Reactor types internally, use those, and then return either `Flux` or `Mono` as output. So you can pass any `Publisher` as input and you can apply operations on the output, but you'll need to adapt the output for use with another reactive library. Whenever feasible — e.g. annotated controllers, WebFlux adapts transparently to the use of RxJava or other reactive library. See Reactive Libraries for more details.

1.1.4. Programming models

The `spring-web` module contains the reactive foundation that underlies Spring WebFlux — HTTP abstractions, Reactive Streams server adapters, reactive codecs, and a core Web API whose role is comparable to the Servlet API but with non-blocking semantics.

On that foundation Spring WebFlux provides a choice of two programming models:

- Annotated Controllers — consistent with Spring MVC, and based on the same annotations from the `spring-web` module. Both Spring MVC and WebFlux controllers support reactive (Reactor, RxJava) return types and as a result it is not easy to tell them apart. One notable difference is that WebFlux also supports reactive `@RequestBody` arguments.
- Functional Endpoints — lambda-based, lightweight, functional programming model. Think of this as a small library or a set of utilities that an application can use to route and handle requests. The big difference with annotated controllers is that the application is in charge of request handling from start to finish vs declaring intent through annotations and being called

back.

1.1.5. Choosing a web framework

Should you use Spring MVC or WebFlux? Let's cover a few different perspectives.

If you have a Spring MVC application that works fine, there is no need to change. Imperative programming is the easiest way to write, understand, and debug code. You have maximum choice of libraries since historically most are blocking.

If you are already shopping for a non-blocking web stack, Spring WebFlux offers the same execution model benefits as others in this space and also provides a choice of servers — Netty, Tomcat, Jetty, Undertow, Servlet 3.1+ containers, a choice of programming models — annotated controllers and functional web endpoints, and a choice of reactive libraries — Reactor, RxJava, or other.

If you are interested in a lightweight, functional web framework for use with Java 8 lambdas or Kotlin then use the Spring WebFlux functional web endpoints. That can also be a good choice for smaller applications or microservices with less complex requirements that can benefit from greater transparency and control.

In a microservice architecture you can have a mix of applications with either Spring MVC or Spring WebFlux controllers, or with Spring WebFlux functional endpoints. Having support for the same annotation-based programming model in both frameworks makes it easier to re-use knowledge while also selecting the right tool for the right job.

A simple way to evaluate an application is to check its dependencies. If you have blocking persistence APIs (JPA, JDBC), or networking APIs to use, then Spring MVC is the best choice for common architectures at least. It is technically feasible with both Reactor and RxJava to perform blocking calls on a separate thread but you wouldn't be making the most of a non-blocking web stack.

If you have a Spring MVC application with calls to remote services, try the reactive `WebClient`. You can return reactive types (Reactor, RxJava, or other) directly from Spring MVC controller methods. The greater the latency per call, or the interdependency among calls, the more dramatic the benefits. Spring MVC controllers can call other reactive components too.

If you have a large team, keep in mind the steep learning curve in the shift to non-blocking, functional, and declarative programming. A practical way to start without a full switch is to use the reactive `WebClient`. Beyond that start small and measure the benefits. We expect that for a wide range of applications the shift is unnecessary.

If you are unsure what benefits to look for, start by learning about how non-blocking I/O works (e.g. concurrency on single-threaded Node.js is not an oxymoron) and its effects. The tag line is "scale with less hardware" but that effect is not guaranteed, not without some network I/O that can be slow or unpredictable. This Netflix blog post is a good resource.

1.1.6. Choosing a server

Spring WebFlux is supported on Netty, Undertow, Tomcat, Jetty, and Servlet 3.1+ containers. Each

server is adapted to a common Reactive Streams API. The Spring WebFlux programming models are built on that common API.

Common question: how can Tomcat and Jetty be used in both stacks?
Tomcat and Jetty are non-blocking at their core. It's the Servlet API that adds a blocking facade. Starting in version 3.1 the Servlet API adds a choice for non-blocking I/O. However its use requires care to avoid other synchronous and blocking parts. For this reason Spring's reactive web stack has a low-level Servlet adapter to bridge to Reactive Streams but the Servlet API is otherwise not exposed for direct use.

Spring Boot 2 uses Netty by default with WebFlux because Netty is more widely used in the async, non-blocking space and also provides both client and server that can share resources. By comparison Servlet 3.1 non-blocking I/O hasn't seen much use because the bar to use it is so high. Spring WebFlux opens one practical path to adoption.

The default server choice in Spring Boot is mainly about the out-of-the-box experience. Applications can still choose any of the other supported servers which are also highly optimized for performance, fully non-blocking, and adapted to Reactive Streams back pressure. In Spring Boot it is trivial to make the switch.

1.1.7. Performance vs scale

Performance has many characteristics and meanings. Reactive and non-blocking generally do not make applications run faster. They can, in some cases, for example if using the `WebClient` to execute remote calls in parallel. On the whole it requires more work to do things the non-blocking way and that can increase slightly the required processing time.

The key expected benefit of reactive and non-blocking is the ability to scale with a small, fixed number of threads and less memory. That makes applications more resilient under load because they scale in a more predictable way. In order to observe those benefits however you need to have some latency including a mix of slow and unpredictable network I/O. That's where the reactive stack begins to show its strengths and the differences can be dramatic.

1.2. Reactive Spring Web

The `spring-web` module provides low level infrastructure and HTTP abstractions — client and server, to build reactive web applications. All public APIs are build around Reactive Streams with Reactor as a backing implementation.

Server support is organized in two layers:

- HttpHandler and server adapters — the most basic, common API for HTTP request handling with Reactive Streams back pressure.
- WebHandler API — slightly higher level but still general purpose server web API with filter chain style processing.

1.2.1. HttpHandler

Every HTTP server has some API for HTTP request handling. HttpHandler is a simple contract with one method to handle a request and response. It is intentionally minimal. Its main purpose is to provide a common, Reactive Streams based API for HTTP request handling over different servers.

The `spring-web` module contains adapters for every supported server. The table below shows the server APIs are used and where Reactive Streams support comes from:

Server name	Server API used	Reactive Streams support
Netty	Netty API	Reactor Netty
Undertow	Undertow API	spring-web: Undertow to Reactive Streams bridge
Tomcat	Servlet 3.1 non-blocking I/O; Tomcat API to read and write ByteBuffers vs byte[]	spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge
Jetty	Servlet 3.1 non-blocking I/O; Jetty API to write ByteBuffers vs byte[]	spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge
Servlet 3.1 container	Servlet 3.1 non-blocking I/O	spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge

Here are required dependencies, supported versions, and code snippets for each server:

Server name	Group id	Artifact name
Reactor Netty	io.projectreactor.ipc	reactor-netty
Undertow	io.undertow	undertow-core
Tomcat	org.apache.tomcat.embed	tomcat-embed-core
Jetty	org.eclipse.jetty	jetty-server, jetty-servlet

Reactor Netty:

```
HttpHandler handler = ...
ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(handler);
HttpServer.create(host, port).newHandler(adapter).block();
```

Undertow:

```
HttpHandler handler = ...
UndertowHttpHandlerAdapter adapter = new UndertowHttpHandlerAdapter(handler);
Undertow server = Undertow.builder().addHttpListener(port, host).setHandler(adapter)
.build();
server.start();
```

Tomcat:

```
HttpHandler handler = ...
Servlet servlet = new TomcatHttpHandlerAdapter(handler);

Tomcat server = new Tomcat();
File base = new File(System.getProperty("java.io.tmpdir"));
Context rootContext = server.addContext("", base.getAbsolutePath());
Tomcat.addServlet(rootContext, "main", servlet);
rootContext.addServletMappingDecoded("/", "main");
server.setHost(host);
server.setPort(port);
server.start();
```

Jetty:

```
HttpHandler handler = ...
Servlet servlet = new JettyHttpHandlerAdapter(handler);

Server server = new Server();
ServletContextHandler contextHandler = new ServletContextHandler(server, "");
contextHandler.addServlet(new ServletHolder(servlet), "/");
contextHandler.start();

ServerConnector connector = new ServerConnector(server);
connector.setHost(host);
connector.setPort(port);
server.addConnector(connector);
server.start();
```

You can also deploy as a WAR to any Servlet 3.1 container by wrapping the handler with `ServletHttpHandlerAdapter` as a `Servlet`.

1.2.2. WebHandler API

`HttpHandler` is the basis for running on different servers. On that base the WebHandler API provides a slightly higher level processing chain of exception handlers (WebExceptionHandler), filters (WebFilter), and a target handler (WebHandler).

All components work on `ServerWebExchange` — a container for the HTTP request and response that also adds request attributes, session attributes, access to form data, multipart data, and more.

The processing chain can be put together with `WebHttpHandlerBuilder` which builds an `HttpHandler` that in turn can be run with a server adapter. To use the builder either add components individually or point to an `ApplicationContext` to have the following detected:

Bean name	Bean type	Count	Description
"webHandler"	WebHandler	1	Target handler after filters
<any>	WebFilter	0..N	Filters

Bean name	Bean type	Count	Description
<any>	WebExceptionHandler	0..N	Exception handlers after filter chain
"webSessionManager"	WebSessionManager	0..1	Custom session manager; `DefaultWebSessionManager` by default
"serverCodecConfigurer"	ServerCodecConfigurer	0..1	Custom form and multipart data decoders; `ServerCodecConfigurer.create()` by default
"localeContextResolver"	LocaleContextResolver	0..1	Custom resolver for `LocaleContext`; `AcceptHeaderLocaleContextResolver` by default

1.2.3. Codecs

The `spring-web` module provides HttpMessageReader and HttpMessageWriter for encoding and decoding the HTTP request and response body with Reactive Streams. It builds on lower level contracts from `spring-core`:

- DataBuffer — abstraction for byte buffers — e.g. Netty `ByteBuf`, `java.nio.ByteBuffer`
- Encoder — serialize a stream of Objects to a stream of data buffers
- Decoder — deserialize a stream of data buffers into a stream of Objects

Basic `Encoder` and `Decoder` implementations exist in `spring-core` but `spring-web` adds more for JSON, XML, and other formats. You can wrap any `Encoder` and `Decoder` as a reader or writer with `EncoderHttpMessageWriter` and `DecoderHttpMessageReader`. There are some additional, web-only reader and writer implementations for server-sent events, form data, and more.

Finally, `ClientCodecConfigurer` and `ServerCodecConfigurer` can be used to initialize a list of readers and writers. They include support for classpath detection and a of defaults along with the ability to override or replace those defaults.

1.3. The DispatcherHandler

Same in Spring MVC

Spring WebFlux, like Spring MVC, is designed around the front controller pattern where a central `WebHandler`, the `DispatcherHandler`, provides a shared algorithm for request processing while actual work is performed by configurable, delegate components. This model is flexible and supports diverse workflows.

`DispatcherHandler` discovers the delegate components it needs from Spring configuration. It is also designed to be a Spring bean itself and implements `ApplicationContextAware` for access to the context it runs in. If `DispatcherHandler` is declared with the bean name "webHandler" it is in turn discovered by WebHttpHandlerBuilder which puts together a request processing chain as described in WebHandler API.

Spring configuration in a WebFlux application typically contains:

- `DispatcherHandler` with the bean name "webHandler"
- `WebFilter` and `WebExceptionHandler` beans
- DispatcherHandler special beans
- Others

The configuration is given to `WebHttpHandlerBuilder` to build the processing chain:

```
ApplicationContext context = ...
HttpHandler handler = WebHttpHandlerBuilder.applicationContext(context);
```

The resulting `HttpHandler` is ready for use with a server adapter.

1.3.1. Special bean types

Same in Spring MVC

The `DispatcherHandler` delegates to special beans to process requests and render the appropriate responses. By "special beans" we mean Spring-managed Object instances that implement one of the framework contracts listed in the table below. Spring WebFlux provides built-in implementations of these contracts but you can also customize, extend, or replace them.

Table 1. Special bean types in the ApplicationContext

Bean type	Explanation
HandlerMapping	Map a request to a handler. The mapping is based on some criteria the details of which vary by `HandlerMapping` implementation — annotated controllers, simple URL pattern mappings, etc.
HandlerAdapter	Helps the `DispatcherHandler` to invoke a handler mapped to a request regardless of how the handler is actually invoked. For example invoking an annotated controller requires resolving various annotations. The main purpose of a `HandlerAdapter` is to shield the `DispatcherHandler` from such details.
HandlerResultHandler	Process the `HandlerResult` returned from a `HandlerAdapter`.

1.3.2. Processing sequence

Same in Spring MVC

The `DispatcherHandler` processes requests as follows:

- Each `HandlerMapping` is asked to find a matching handler and the first match is used.
- If a handler is found, it is executed through an appropriate `HandlerAdapter` which exposes the return value from the execution as `HandlerResult`.

- The `HandlerResult` is given to an appropriate `HandlerResultHandler` to complete processing by writing to the response directly or using a view to render.

1.4. Annotated Controllers

Same in Spring MVC

Spring WebFlux provides an annotation-based programming model where `@Controller` and `@RestController` components use annotations to express request mappings, request input, exception handling, and more. Annotated controllers have flexible method signatures and do not have to extend base classes nor implement specific interfaces.

Here is a basic example:

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String handle() {
        return "Hello WebFlux";
    }
}
```

In this example the methods returns a String to be written to the response body.

1.4.1. @Controller declaration

Same in Spring MVC

You can define controller beans using a standard Spring bean definition. The `@Controller` stereotype allows for auto-detection, aligned with Spring general support for detecting `@Component` classes in the classpath and auto-registering bean definitions for them. It also acts as a stereotype for the annotated class, indicating its role as a web component.

To enable auto-detection of such `@Controller` beans, you can add component scanning to your Java configuration:

```
@Configuration
@ComponentScan("org.example.web")
public class WebConfig {

    // ...
}
```

`@RestController` is a composed annotation that is itself annotated with `@Controller` and `@ResponseBody` indicating a controller whose every method inherits the type-level `@ResponseBody` annotation and therefore writes to the response body (vs model-and-vew rendering).

1.4.2. Mapping Requests

Same in Spring MVC

The `@RequestMapping` annotation is used to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types. It can be used at the class-level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.

There are also HTTP method specific shortcut variants of `@RequestMapping`:

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

The shortcut variants are composed annotations — themselves annotated with `@RequestMapping`. They are commonly used at the method level. At the class level an `@RequestMapping` is more useful for expressing shared mappings.

```
@RestController
@RequestMapping("/persons")
class PersonController {

    @GetMapping("/{id}")
    public Person getPerson(@PathVariable Long id) {
        // ...
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void add(@RequestBody Person person) {
        // ...
    }
}
```

URI Patterns

Same in Spring MVC

You can map requests using glob patterns and wildcards:

- ? matches one character
- * matches zero or more characters within a path segment
- ** match zero or more path segments

You can also declare URI variables and access their values with `@PathVariable`:

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
    // ...
}
```

URI variables can be declared at the class and method level:

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class OwnerController {

    @GetMapping("/pets/{petId}")
    public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
        // ...
    }
}
```

URI variables are automatically converted to the appropriate type or `TypeMismatchException` is raised. Simple types — `int`, `long`, `Date`, are supported by default and you can register support for any other data type.

URI variables can be named explicitly — e.g. `@PathVariable("customId")`, but you can leave that detail out if the names are the same and your code is compiled with debugging information or with the `-parameters` compiler flag on Java 8.

The syntax `{*varName}` declares a URI variable that matches zero or more remaining path segments. For example `/resources/{*path}` matches all files `/resources/` and the `"path"` variable captures the complete relative path.

The syntax `{varName:regex}` declares a URI variable with a regular expressions with the syntax `{varName:regex}` — e.g. given URL `"/spring-web-3.0.5 .jar"`, the below method extracts the name, version, and file extension:

```
@GetMapping("/{name:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{ext:\\.[a-z]+}")
public void handle(@PathVariable String version, @PathVariable String ext) {
    // ...
}
```

URI path patterns can also have embedded `${…}` placeholders that are resolved on startup via `PropertyPlaceHolderConfigurer` against local, system, environment, and other property sources. This

can be used for example to parameterize a base URL based on some external configuration.

Spring WebFlux uses `PathPattern` and the `PathPatternParser` for URI path matching support both of which are located in `spring-web` and expressly designed for use with HTTP URL paths in web applications where a large number of URI path patterns are matched at runtime.

Spring WebFlux does not support suffix pattern matching — unlike Spring MVC, where a mapping such as `/person` also matches to `/person.*`. For URL based content negotiation, if needed, we recommend using a query parameter, which is more simpler, more explicit, and less vulnerable to URL path based exploits.

Pattern Comparison

Same in Spring MVC

When multiple patterns match a URL, they must be compared to find the best match. This is done with `PathPattern.SPECIFICITY_COMPARATOR` which looks for patterns that more specific.

For every pattern, a score is computed based the number of URI variables and wildcards where a URI variable scores lower than a wildcard. A pattern with a lower total score wins. If two patterns have the same score, then the longer is chosen.

Catch-all patterns, e.g. `**`, `{*varName}`, are excluded from the scoring and are always sorted last instead. If two patterns are both catch-all, the longer is chosen.

Consumable Media Types

Same in Spring MVC

You can narrow the request mapping based on the `Content-Type` of the request:

```
@PostMapping(path = "/pets", <strong>consumes = "application/json"</strong>)
public void addPet(@RequestBody Pet pet) {
    // ...
}
```

The consumes attribute also supports negation expressions — e.g. `!text/plain` means any content type other than "text/plain".

You can declare a shared consumes attribute at the class level. Unlike most other request mapping attributes however when used at the class level, a method-level consumes attribute will overrides rather than extend the class level declaration.

`MediaType` provides constants for commonly used media types — e.g. `APPLICATION_JSON_VALUE`, `APPLICATION_JSON_UTF8_VALUE`.

Producible Media Types

Same in Spring MVC

You can narrow the request mapping based on the `Accept` request header and the list of content types that a controller method produces:

```
@GetMapping(path = "/pets/{petId}", <strong>produces = "application/json;charset=UTF-
8"</strong>)
@ResponseBody
public Pet getPet(@PathVariable String petId) {
    // ...
}
```

The media type can specify a character set. Negated expressions are supported — e.g. `!text/plain` means any content type other than "text/plain".

You can declare a shared produces attribute at the class level. Unlike most other request mapping attributes however when used at the class level, a method-level produces attribute will overrides rather than extend the class level declaration.

`MediaType` provides constants for commonly used media types — e.g. `APPLICATION_JSON_VALUE`, `APPLICATION_JSON_UTF8_VALUE`.

Parameters and Headers

Same in Spring MVC

You can narrow request mappings based on query parameter conditions. You can test for the presence of a query parameter (`"myParam"`), for the absence (`"!myParam"`), or for a specific value (`"myParam=myValue"`):

```
@GetMapping(path = "/pets/{petId}", <strong>params = "myParam=myValue"</strong>)
public void findPet(@PathVariable String petId) {
    // ...
}
```

You can also use the same with request header conditions:

```
@GetMapping(path = "/pets", <strong>headers = "myHeader=myValue"</strong>)
public void findPet(@PathVariable String petId) {
    // ...
}
```

HTTP HEAD, OPTIONS

Same in Spring MVC

`@GetMapping` — and also `@RequestMapping(method=HttpMethod.GET)`, support HTTP HEAD transparently for request mapping purposes. Controller methods don't need to change. A response wrapper, applied in the `HttpHandler` server adapter, ensures a `"Content-Length"` header is set to the number of bytes written and without actually writing to the response.

By default HTTP OPTIONS is handled by setting the "Allow" response header to the list of HTTP methods listed in all `@RequestMapping` methods with matching URL patterns.

For a `@RequestMapping` without HTTP method declarations, the "Allow" header is set to `"GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS"`. Controller methods should always declare the supported HTTP methods for example by using the HTTP method specific variants — `@GetMapping`, `@PostMapping`, etc.

`@RequestMapping` method can be explicitly mapped to HTTP HEAD and HTTP OPTIONS, but that is not necessary in the common case.

1.4.3. Handler methods

Same in Spring MVC

`@RequestMapping` handler methods have a flexible signature and can choose from a range of supported controller method arguments and return values.

Method arguments

Same in Spring MVC

The table below shows supported controller method arguments.

Reactive types (Reactor, RxJava, or other) are supported on arguments that require blocking I/O, e.g. reading the request body, to be resolved. This is marked in the description column. Reactive types are not expected on arguments that don't require blocking.

JDK 1.8's `java.util.Optional` is supported as a method argument in combination with annotations that have a `required` attribute — e.g. `@RequestParam`, `@RequestHeader`, etc, and is equivalent to `required=false`.

Controller method argument	Description
`ServerWebExchange`	Access to the full `ServerWebExchange` — container for the HTTP request and response, request and session attributes, `checkNotModified` methods, and others.
`ServerHttpRequest`, `ServerHttpResponse`	Access to the HTTP request or response.
`WebSession`	Access to the session; this does not forcing the start of a new session unless attributes are added. Supports reactive types.
`java.security.Principal`	Currently authenticated user; possibly a specific `Principal` implementation class if known. Supports reactive types.
`org.springframework.http.HttpMethod`	The HTTP method of the request.

Controller method argument	Description
`java.util.Locale`	The current request locale, determined by the most specific `LocaleResolver` available, in effect, the configured `LocaleResolver` /`LocaleContextResolver`.
Java 6+: `java.util.TimeZone` Java 8+: `java.time.ZoneId`	The time zone associated with the current request, as determined by a `LocaleContextResolver`.
`@PathVariable`	For access to URI template variables.
`@RequestParam`	For access to Servlet request parameters. Parameter values are converted to the declared method argument type.
`@RequestHeader`	For access to request headers. Header values are converted to the declared method argument type.
`@RequestBody`	For access to the HTTP request body. Body content is converted to the declared method argument type using `HttpMessageReader`'s. Supports reactive types.
`HttpEntity<B>`	For access to request headers and body. The body is converted with `HttpMessageReader`'s. Supports reactive types.
`@RequestPart`	For access to a part in a "multipart/form-data" request. Supports reactive types.
`java.util.Map`, `org.springframework.ui.Model`, `org.springframework.ui.ModelMap`	For access and updates of the implicit model that is exposed to the web view.
Command or form object (with optional `@ModelAttribute`)	Command object whose properties to bind to request parameters — via setters or directly to fields, with customizable type conversion, depending on `@InitBinder` methods and/or the HandlerAdapter configuration (see the `webBindingInitializer` property on `RequestMappingHandlerAdapter`). Command objects along with their validation results are exposed as model attributes, by default using the command class name - e.g. model attribute "orderAddress" for a command object of type "some.package.OrderAddress". `@ModelAttribute` can be used to customize the model attribute name. Supports reactive types.
`Errors`, `BindingResult`	Validation results for the command/form object data binding; this argument must be declared immediately after the command/form object in the controller method signature.
`SessionStatus`	For marking form processing complete which triggers cleanup of session attributes declared through a class-level `@SessionAttributes` annotation.
`UriComponentsBuilder`	For preparing a URL relative to the current request's host, port, scheme, context path, and the literal part of the servlet mapping also taking into account `Forwarded` and `X-Forwarded-*` headers.
`@SessionAttribute`	For access to any session attribute; in contrast to model attributes stored in the session as a result of a class-level `@SessionAttributes` declaration.

Controller method argument	Description
`@RequestAttribute`	For access to request attributes.

Return values

Same in Spring MVC

The table below shows supported controller method return values. Reactive types — Reactor, RxJava, or other are supported for all return values.

Controller method return value	Description
`@ResponseBody`	The return value is encoded through `HttpMessageWriter`s and written to the response.
`HttpEntity<B>`, `ResponseEntity<B>`	The return value specifies the full response including HTTP headers and body be encoded through `HttpMessageWriter`s and written to the response.
`HttpHeaders`	For returning a response with headers and no body.
`String`	A view name to be resolved with `ViewResolver`'s and used together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method may also programmatically enrich the model by declaring a `Model` argument (see above).
`View`	A `View` instance to use for rendering together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method may also programmatically enrich the model by declaring a `Model` argument (see above).
`java.util.Map`, `org.springframework.ui.Model`	Attributes to be added to the implicit model with the view name implicitly determined from the request path.
`Rendering`	An API for model and view rendering scenarios.
`void`	For use in method that don't write the response body; or methods where the view name is supposed to be determined implicitly from the request path.
`Flux<ServerSentEvent>`, `Observable<ServerSentEvent>`, or other reactive type	Emit server-sent events; the `SeverSentEvent` wrapper can be omitted when only data needs to be written (however `text/event-stream` must be requested or declared in the mapping through the produces attribute).
Any other return type	A single model attribute to be added to the implicit model with the view name implicitly determined through a `RequestToViewNameTranslator`; the attribute name may be specified through a method-level `@ModelAttribute` or otherwise a name is selected based on the class name of the return type.

1.5. Functional Endpoints

Spring WebFlux provides a lightweight, functional programming model where functions are used to route and handle requests and where contracts are designed for immutability. It is an alternative to the annotated-based programming model but runs on the same Reactive Spring Web foundation

1.5.1. HandlerFunction

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 is an `@RequestMapping` method.

`ServerRequest` and `ServerResponse` are immutable interfaces that offer JDK-8 friendly access to the underlying HTTP messages with Reactive Streams non-blocking back pressure. The request exposes the body as Reactor `Flux` or `Mono` types; the response accepts any Reactive Streams `Publisher` as body (see Reactive Libraries).

`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. For instance, this is how to extract the request body into a `Mono<String>`:

```
Mono<String> string = request.bodyToMono(String.class);
```

And here is how to extract the body into a `Flux`, where `Person` is a class that can be deserialised from the contents of the body (i.e. `Person` is supported by Jackson if the body contains JSON, or JAXB if XML).

```
Flux<Person> people = request.bodyToFlux(Person.class);
```

The above — `bodyToMono` and `bodyToFlux`, are, in fact, convenience methods that use the generic `ServerRequest.body(BodyExtractor)` method. `BodyExtractor` is a functional strategy interface that allows you to write your own extraction logic, but common `BodyExtractor` instances can be found in the `BodyExtractors` utility class. So, the above examples can be replaced with:

```
Mono<String> string = request.body(BodyExtractors.toMono(String.class);
Flux<Person> people = request.body(BodyExtractors.toFlux(Person.class);
```

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. For instance, this is how to create a response with a 200 OK status, a JSON content-type, and a body:

```
Mono<Person> person = ...
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(person);
```

And here is how to build a response with a 201 CREATED status, a `"Location"` header, and empty body:

```
URI location = ...
ServerResponse.created(location).build();
```

Putting these together allows us to create a `HandlerFunction`. For instance, here is an example of a simple "Hello World" handler lambda, that returns a response with a 200 status and a body based on a String:

```
HandlerFunction<ServerResponse> helloWorld =
  request -> ServerResponse.ok().body(fromObject("Hello World"));
```

Writing handler functions as lambda's, as we do above, is convenient, but perhaps lacks in readability and becomes less maintainable when dealing with multiple functions. Therefore, it is recommended to group related handler functions into a handler or controller class. For example, here is a class that exposes a reactive `Person` repository:

```
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) { ①
        Flux<Person> people = repository.allPeople();
        return ServerResponse.ok().contentType(APPLICATION_JSON).body(people, Person
.class);
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) { ②
        Mono<Person> person = request.bodyToMono(Person.class);
        return ServerResponse.ok().build(repository.savePerson(person));
    }

    public Mono<ServerResponse> getPerson(ServerRequest request) { ③
        int personId = Integer.valueOf(request.pathVariable("id"));
        Mono<ServerResponse> notFound = ServerResponse.notFound().build();
        Mono<Person> personMono = this.repository.getPerson(personId);
        return personMono
                .flatMap(person -> ServerResponse.ok().contentType(APPLICATION_JSON)
.body(fromObject(person)))
                .switchIfEmpty(notFound);
    }
}
```

① `listPeople` is a handler function that returns all `Person` objects found in the repository as JSON.

② `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.

③ `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 `switchIfEmpty(Mono<T>)` to return a 404 Not Found response.

1.5.2. RouterFunction

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. If the predicate applies, the request is routed to the given handler function; otherwise no routing is performed, resulting in a 404 Not Found response. Though you can write your own `RequestPredicate`, you do not have to: the `RequestPredicates` utility class offers commonly used predicates, such matching based on path, HTTP method, content-type, etc. Using `route`, we can route to our "Hello World" handler function:

```
RouterFunction<ServerResponse> helloWorldRoute =
    RouterFunctions.route(RequestPredicates.path("/hello-world"),
    request -> Response.ok().body(fromObject("Hello World")));
```

Two router functions can be composed into a new router function that routes to either handler function: if the predicate of the first route does not match, the second is evaluated. Composed router functions are evaluated in order, so it makes sense to put specific functions before generic ones. You can compose two router functions by calling `RouterFunction.and(RouterFunction)`, or by calling `RouterFunction.andRoute(RequestPredicate, HandlerFunction)`, which is a convenient combination of `RouterFunction.and()` with `RouterFunctions.route()`.

Given the `PersonHandler` we showed above, we can now define a router function that routes to the respective handler functions. We use method-references to refer to the handler functions:

```
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)`.

1.5.3. Running a server

How do you run a router function in an HTTP server? A simple option is to convert a router function to an `HttpHandler` via `RouterFunctions.toHttpHandler(RouterFunction)`. The `HttpHandler` can

then be used with a number of servers adapters. See HttpHandler for server-specific instructions.

it is also possible to run with a DispatcherHandler setup — side by side with annotated controllers. The easiest way to do that is through the WebFlux Java Config which creates the necessary configuration to handle requests with router and handler functions.

1.5.4. HandlerFilterFunction

Routes mapped by a router function can be filtered by calling `RouterFunction.filter(HandlerFilterFunction)`, where `HandlerFilterFunction` is essentially a function that takes a `ServerRequest` and `HandlerFunction`, and returns a `ServerResponse`. The handler function parameter represents the next element in the chain: this is typically the `HandlerFunction` that is routed to, but can also be another `FilterFunction` if multiple filters are applied. With annotations, similar functionality can be achieved using `@ControllerAdvice` and/or a `ServletFilter`. Let's add a simple security filter to our route, assuming that we have a `SecurityManager` that can determine whether a particular path is allowed:

```
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();
        }
  });
```

You can see in this example that invoking the `next.handle(ServerRequest)` is optional: we only allow the handler function to be executed when access is allowed.

1.6. WebFlux Java Config

Same in Spring MVC

The WebFlux Java config provides default configuration suitable for most applications along with a configuration API to customize it. For more advanced customizations, not available in the configuration API, see Advanced config mode.

You do not need to understand the underlying beans created by the Java config, but it's easy to seem them in `WebFluxConfigurationSupport`, and if you want to learn more, see Special bean types.

1.6.1. Enable the configuration

Same in Spring MVC

Use the `@EnableWebFlux` annotation in your Java config:

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

The above registers a number of Spring WebFlux infrastructure beans also adapting to dependencies available on the classpath — for JSON, XML, etc.

1.6.2. Configuration API

Same in Spring MVC

In your Java config implement the `WebFluxConfigurer` interface:

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    // Implement configuration methods...

}
```

1.6.3. Conversion, formatting

Same in Spring MVC

By default formatters for `Number` and `Date` types are installed, including support for the `@NumberFormat` and `@DateTimeFormat` annotations. Full support for the Joda Time formatting library is also installed if Joda Time is present on the classpath.

To register custom formatters and converters:

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        // ...
    }

}
```

See FormatterRegistrar SPI and the FormattingConversionServiceFactoryBean for more information on when to use FormatterRegistrars.

1.6.4. Validation

Same in Spring MVC

By default if Bean Validation is present on the classpath — e.g. Hibernate Validator, the LocalValidatorFactoryBean is registered as a global Validator for use with @Valid and Validated on @Controller method arguments.

In your Java config, you can customize the global Validator instance:

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public Validator getValidator(); {
        // ...
    }

}
```

Note that you can also register Validator's locally:

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(new FooValidator());
    }

}
```

If you need to have a `LocalValidatorFactoryBean` injected somewhere, create a bean and mark it with `@Primary` in order to avoid conflict with the one declared in the MVC config.

1.6.5. Content type resolvers

Same in Spring MVC

You can configure how Spring WebFlux determines the requested media types for `@Controller`'s from the request. By default only the "Accept" header is checked but you can also enable a query parameter based strategy.

To customize the requested content type resolution:

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureContentTypeResolver(RequestedContentTypeResolverBuilder
builder) {
        // ...
    }
}
```

1.6.6. HTTP message codecs

Same in Spring MVC

To customize how the request and response body are read and written:

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        // ...
    }
}
```

`ServerCodecConfigurer` provides a set of default readers and writers. You can use it to add more readers and writers, customize the default ones, or replace the default ones completely.

For Jackson JSON and XML, consider using the Jackson2ObjectMapperBuilder which customizes Jackson's default properties with the following ones:

1. `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled.

2. `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled.

It also automatically registers the following well-known modules if they are detected on the classpath:

1. jackson-datatype-jdk7: support for Java 7 types like `java.nio.file.Path`.
2. jackson-datatype-joda: support for Joda-Time types.
3. jackson-datatype-jsr310: support for Java 8 Date & Time API types.
4. jackson-datatype-jdk8: support for other Java 8 types like `Optional`.

1.6.7. View resolvers

Same in Spring MVC

To configure view resolution:

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        // ...
    }
}
```

Note that FreeMarker also requires configuration of the underlying view technology:

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    // ...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("classpath:/templates");
        return configurer;
    }

}
```

1.6.8. Static resources

Same in Spring MVC

This option provides a convenient way to serve static resources from a list of Resource-based locations.

In the example below, given a request that starts with `"/resources"`, the relative path is used to find and serve static resources relative to `"/static"` on the classpath. Resources will be served with a 1-year future expiration to ensure maximum use of the browser cache and a reduction in HTTP requests made by the browser. The `Last-Modified` header is also evaluated and if present a `304` status code is returned.

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("/public", "classpath:/static/")
            .setCachePeriod(31556926);
    }

}
```

The resource handler also supports a chain of ResourceResolver's and ResourceResolver's. which can be used to create a toolchain for working with optimized resources.

The `VersionResourceResolver` can be used for versioned resource URLs based on an MD5 hash computed from the content, a fixed application version, or other. A `ContentVersionStrategy` (MD5 hash) is a good choice with some notable exceptions such as JavaScript resources used with a module loader.

For example in your Java config;

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public/")
                .resourceChain(true)
                .addResolver(new VersionResourceResolver().addContentVersionStrategy(
"/**"));
    }

}
```

You can use `ResourceUrlProvider` to rewrite URLs and apply the full chain of resolvers and

transformers — e.g. to insert versions. The WebFlux config provides a `ResourceUrlProvider` so it can be injected into others.

Unlike Spring MVC at present in WebFlux there is no way to transparely rewrite static resource URLs since the are no view technologies that can make use of a non-blocking chain of resolvers and transformers (e.g. resources on Amazon S3). When serving only local resources the workaround is to use `ResourceUrlProvider` directly (e.g. through a custom tag) and block for 0 seconds.

WebJars is also supported via `WebJarsResourceResolver` and automatically registered when `"org.webjars:webjars-locator"` is present on the classpath. The resolver can re-write URLs to include the version of the jar and can also match to incoming URLs without versions — e.g. `"/jquery/jquery.min.js"` to `"/jquery/1.2.0/jquery.min.js"`.

1.6.9. Path Matching

Same in Spring MVC

Spring WebFlux uses parsed representation of path patterns — i.e. `PathPattern`, and also the incoming request path — i.e. `RequestPath`, which eliminates the need to indicate whether to decode the request path, or remove semicolon content, since `PathPattern` can now access decoded path segment values and match safely.

Spring WebFlux also does not support suffix pattern matching so effectively there are only two minor options to customize related to path matching — whether to match trailing slashes (`true` by default) and whether the match is case-sensitive (`false`).

To customize those options:

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        // ...
    }

}
```

1.6.10. Advanced config mode

Same in Spring MVC

`@EnableWebFlux` imports `DelegatingWebFluxConfiguration` that (1) provides default Spring configuration for WebFlux applications and (2) detects and delegates to `WebFluxConfigurer`'s to customize that configuration.

For advanced mode, remove `@EnableWebFlux` and extend directly from `DelegatingWebFluxConfiguration` instead of implementing `WebFluxConfigurer`:

```
@Configuration
public class WebConfig extends DelegatingWebFluxConfiguration {

    // ...

}
```

You can keep existing methods in `WebConfig` but you can now also override bean declarations from the base class and you can still have any number of other `WebMvcConfigurer`'s on the classpath.

1.7. WebClient

The `spring-webflux` module includes a non-blocking, reactive client for HTTP requests with Reactive Streams back pressure. It shares HTTP codecs and other infrastructure with the server functional web framework.

`WebClient` provides a higher level API over HTTP client libraries. By default it uses Reactor Netty but that is pluggable with a different `ClientHttpConnector`. The `WebClient` API returns Reactor `Flux` or `Mono` for output and accepts Reactive Streams `Publisher` as input (see Reactive Libraries).

By comparison to the RestTemplate, the `WebClient` offers a more functional and fluent API that taking full advantage of Java 8 lambdas. It supports both sync and async scenarios, including streaming, and brings the efficiency of non-blocking I/O.

1.7.1. Retrieve

The `retrieve()` method is the easiest way to get a response body and decode it:

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

    Mono<Person> result = client.get()
            .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
            .retrieve()
            .bodyToMono(Person.class);
```

You can also get a stream of objects decoded from the response:

```
    Flux<Quote> result = client.get()
            .uri("/quotes").accept(TEXT_EVENT_STREAM)
            .retrieve()
            .bodyToFlux(Quote.class);
```

By default, responses with 4xx or 5xx status codes result in an error of type `WebClientResponseException` but you can customize that:

```
Mono<Person> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .onStatus(HttpStatus::is4xxServerError, response -> ...)
        .onStatus(HttpStatus::is5xxServerError, response -> ...)
        .bodyToFlux(Person.class);
```

1.7.2. Exchange

The `exchange()` method provides more control. The below example is equivalent to `retrieve()` but also provides access to the `ClientResponse`:

```
Mono<Person> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .exchange()
        .flatMap(response -> response.bodyToMono(Person.class));
```

At this level you can also create a full `ResponseEntity`:

```
Mono<ResponseEntity<Person>> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .exchange()
        .flatMap(response -> response.bodyToEntity(Person.class));
```

Note that unlike `retrieve()`, with `exchange()` there are no automatic error signals for 4xx and 5xx responses. You have to check the status code and decide how to proceed.

When you use `exchange()`, you must call `response.close()` if you do not intend to read the response body in order to close the underlying HTTP connection. Not doing so can result in connection pool inconsistencies or memory leaks.

You do not have to call `response.close()` if you consume the body because forcing a connection to be closed negates the benefits of persistent connections and connection pooling.

1.7.3. Request body

The request body can be encoded from an Object:

```
Mono<Person> personMono = ... ;

Mono<Void> result = client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .body(personMono, Person.class)
        .retrieve()
        .bodyToMono(Void.class);
```

You can also have a stream of objects encoded:

```
Flux<Person> personFlux = ... ;

Mono<Void> result = client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_STREAM_JSON)
        .body(personFlux, Person.class)
        .retrieve()
        .bodyToMono(Void.class);
```

Or if you have the actual value, use the `syncBody` shortcut method:

```
Person person = ... ;

Mono<Void> result = client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .syncBody(person)
        .retrieve()
        .bodyToMono(Void.class);
```

1.7.4. Builder options

A simple way to create `WebClient` is through the static factory methods `create()` and `create(String)` with a base URL for all requests. You can also use `WebClient.builder()` for access to more options.

To customize the underlying HTTP client:

```
SslContext sslContext = ...

ClientHttpConnector connector = new ReactorClientHttpConnector(
        builder -> builder.sslContext(sslContext));

WebClient webClient = WebClient.builder()
        .clientConnector(connector)
        .build();
```

To customize the HTTP codecs used for encoding and decoding HTTP messages:

```
    ExchangeStrategies strategies = ExchangeStrategies.builder()
            .codecs(configurer -> {
                // ...
            })
            .build();

    WebClient webClient = WebClient.builder()
            .exchangeStrategies(strategies)
            .build();
```

The builder can be used to insert Filters.

Explore the `WebClient.Builder` in your IDE for other options related to URI building, default headers (and cookies), and more.

After the `WebClient` is built, you can always obtain a new builder from it, in order to build a new `WebClient`, based on, but without affecting the current instance:

```
    WebClient modifiedClient = client.mutate()
            // user builder methods...
            .build();
```

1.7.5. Filters

`WebClient` supports interception style request filtering:

```
    WebClient client = WebClient.builder()
        .filter((request, next) -> {

            ClientRequest filtered = ClientRequest.from(request)
                .header("foo", "bar")
                .build();

            return next.exchange(filtered);
        })
        .build();
```

`ExchangeFilterFunctions` provides a filter for basic authentication:

```
// static import of ExchangeFilterFunctions.basicAuthentication

    WebClient client = WebClient.builder()
        .filter(basicAuthentication("user", "pwd"))
        .build();
```

You can also mutate an existing `WebClient` instance without affecting the original:

```
WebClient filteredClient = client.mutate()
        .filter(basicAuthentication("user", "pwd")
        .build();
```

1.8. Reactive Libraries

Reactor is a required dependency for the `spring-webflux` module and is used internally for composing logic and for Reactive Streams support. An easy rule to remember is that WebFlux APIs return `Flux` or `Mono` — since that's what's used internally, and leniently accept any Reactive Streams `Publisher` implementation.

The use of `Flux` and `Mono` helps to express cardinality — e.g. whether a single or multiple async values are expected. This is important for API design but also essential in some cases, e.g. when encoding an HTTP message.

For annotated controllers, WebFlux adapts transparently to the reactive library in use with proper translation of cardinality. This is done with the help of the ReactiveAdapterRegistry from `spring-core` which provides pluggable support for reactive and async types. The registry has built-in support for RxJava and `CompletableFuture` but others can be registered.

For functional endpoints, the `WebClient`, and other functional APIs, the general rule of thumb for WebFlux APIs applies:

- `Flux` or `Mono` as return values — use them to compose logic or pass to any Reactive Streams library (both are `Publisher` implementations).
- Reactive Streams `Publisher` for input — if a `Publisher` from another reactive library is provided it can only be treated as a stream with unknown semantics (0..N). If the semantics are known — e.g. `io.reactivex.Single`, you can use `Mono.from(Publisher)` and pass that in instead of the raw `Publisher`.

For example, given a `Publisher` that is not a `Mono`, the Jackson JSON message writer expects multiple values. If the media type implies an infinite stream — e.g. `"application/json+stream"`, values are written and flushed individually; otherwise values are buffered into a list and rendered as a JSON array.