Web on Reactive Stack

Table of Contents

This part of the documentation covers support for reactive-stack web applications built on a Reactive Streams API to run on non-blocking servers, such as Netty, Undertow, and Servlet 3.1+ containers. Individual chapters cover the Spring WebFlux framework, the reactive `WebClient`, support for testing, and reactive libraries. For Servlet-stack web applications, see Web on Servlet Stack.

Spring WebFlux

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 such servers 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 can use one or the other module or, in some cases, both — for example, Spring MVC controllers with the reactive `WebClient`.

Overview

Why was Spring WebFlux created?

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 fewer 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 as the addition of annotations in Java 5 created opportunities (such as 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 annotated controllers.

Define “Reactive”

We touched on “non-blocking” and “functional” but what does reactive mean?

The term, “reactive,” refers to programming models that are built around reacting to change — network components reacting to I/O events, UI controllers reacting to mouse events, and others. 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 let the subscriber to control how quickly or how slowly the publisher produces data.

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

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. Applications need 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 (`Mono`) and 0..N (`Flux`) 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, a WebFlux API accepts a plain `Publisher` as input, adapts it to a Reactor type internally, uses that, and returns either a `Flux` or a `Mono` as output. So, you can pass any `Publisher` as input and you can apply operations on the output, but you need to adapt the output for use with another reactive library. Whenever feasible (for example, annotated controllers), WebFlux adapts transparently to the use of RxJava or another reactive library. See Reactive Libraries for more details.

In addition to Reactive APIs, WebFlux can also be used with Coroutines APIs in Kotlin which provides a more imperative style of programming. The following Kotlin code samples will be provided with Coroutines APIs.

Programming Models

The `spring-web` module contains the reactive foundation that underlies Spring WebFlux, including HTTP abstractions, Reactive Streams adapters for supported servers, codecs, and a core `WebHandler` API comparable to the Servlet API but with non-blocking contracts.

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 and 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, and functional programming model. You can 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 versus declaring intent through annotations and being called back.

Applicability

Spring MVC or WebFlux?

A natural question to ask but one that sets up an unsound dichotomy. Actually, both work together to expand the range of available options. The two are designed for continuity and consistency with each other, they are available side by side, and feedback from each side benefits both sides. The following diagram shows how the two relate, what they have in common, and what each supports uniquely:

[spring mvc and webflux venn] | *images/spring-mvc-and-webflux-venn.png*

We suggest that you consider the following specific points:

- 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, and 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, you can 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, 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 would not 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 (for example, concurrency on single-threaded Node.js) and its effects.

Servers

Spring WebFlux is supported on Tomcat, Jetty, Servlet 3.1+ containers, as well as on non-Servlet runtimes such as Netty and Undertow. All servers are adapted to a low-level, common API so that higher-level programming models can be supported across servers.

Spring WebFlux does not have built-in support to start or stop a server. However, it is easy to assemble an application from Spring configuration and WebFlux infrastructure and run it with a few lines of code.

Spring Boot has a WebFlux starter that automates these steps. By default, the starter uses Netty, but it is easy to switch to Tomcat, Jetty, or Undertow by changing your Maven or Gradle dependencies. Spring Boot defaults to Netty, because it is more widely used in the asynchronous, non-blocking space and lets a client and a server share resources.

Tomcat and Jetty can be used with both Spring MVC and WebFlux. Keep in mind, however, that the way they are used is very different. Spring MVC relies on Servlet blocking I/O and lets applications use the Servlet API directly if they need to. Spring WebFlux relies on Servlet 3.1 non-blocking I/O and uses the Servlet API behind a low-level adapter and not exposed for direct use.

For Undertow, Spring WebFlux uses Undertow APIs directly without the Servlet API.

Performance

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 is where the reactive stack begins to show its strengths, and the differences can be dramatic.

Concurrency Model

Both Spring MVC and Spring WebFlux support annotated controllers, but there is a key difference in the concurrency model and the default assumptions for blocking and threads.

In Spring MVC (and servlet applications in general), it is assumed that applications can block the current thread, (for example, for remote calls), and, for this reason, servlet containers use a large thread pool to absorb potential blocking during request handling.

In Spring WebFlux (and non-blocking servers in general), it is assumed that applications do not block, and, therefore, non-blocking servers use a small, fixed-size thread pool (event loop workers) to handle requests.

"To scale" and "small number of threads" may sound contradictory but to never block the current thread (and rely on callbacks instead) means that you do not need extra threads, as there are no blocking calls to absorb.

Invoking a Blocking API

What if you do need to use a blocking library? Both Reactor and RxJava provide the `publishOn` operator to continue processing on a different thread. That means there is an easy escape hatch. Keep in mind, however, that blocking APIs are not a good fit for this concurrency model.

Mutable State

In Reactor and RxJava, you declare logic through operators, and, at runtime, a reactive pipeline is formed where data is processed sequentially, in distinct stages. A key benefit of this is that it frees applications from having to protect mutable state because application code within that pipeline is never invoked concurrently.

Threading Model

What threads should you expect to see on a server running with Spring WebFlux?

- On a "vanilla" Spring WebFlux server (for example, no data access nor other optional dependencies), you can expect one thread for the server and several others for request processing (typically as many as the number of CPU cores). Servlet containers, however, may start with more threads (for example, 10 on Tomcat), in support of both servlet (blocking) I/O and servlet 3.1 (non-blocking) I/O usage.
- The reactive `WebClient` operates in event loop style. So you can see a small, fixed number of processing threads related to that (for example, `reactor-http-nio-` with the Reactor Netty connector). However, if Reactor Netty is used for both client and server, the two share event loop resources by default.
- Reactor and RxJava provide thread pool abstractions, called Schedulers, to use with the `publishOn` operator that is used to switch processing to a different thread pool. The schedulers have names that suggest a specific concurrency strategy — for example, "parallel" (for CPU-bound work with a limited number of threads) or "elastic" (for I/O-bound work with a large number of threads). If you see such threads, it means some code is using a specific thread pool `Scheduler` strategy.
- Data access libraries and other third party dependencies can also create and use threads of their own.

Configuring

The Spring Framework does not provide support for starting and stopping servers. To configure the threading model for a server, you need to use server-specific configuration APIs, or, if you use Spring Boot, check the Spring Boot configuration options for each server. You can configure the `WebClient` directly. For all other libraries, see their respective documentation.

Reactive Core

The `spring-web` module contains the following foundational support for reactive web applications:

- For server request processing there are two levels of support.
  - HttpHandler: Basic contract for HTTP request handling with non-blocking I/O and Reactive Streams back pressure, along with adapters for Reactor Netty, Undertow, Tomcat, Jetty, and any Servlet 3.1+ container.
  - `WebHandler` API: Slightly higher level, general-purpose web API for request handling, on top of which concrete programming models such as annotated controllers and functional endpoints are built.
- For the client side, there is a basic `ClientHttpConnector` contract to perform HTTP requests with non-blocking I/O and Reactive Streams back pressure, along with adapters for Reactor Netty and for the reactive Jetty HttpClient. The higher level WebClient used in applications builds on this basic contract.
- For client and server, codecs to use to serialize and deserialize HTTP request and response content.

`HttpHandler`

HttpHandler is a simple contract with a single method to handle a request and response. It is intentionally minimal, and its main, and only purpose is to be a minimal abstraction over different HTTP server APIs.

The following table describes the supported server APIs:

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

The following table describes server dependencies (also see supported versions):

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

The code snippets below show using the `HttpHandler` adapters with each server API:

Reactor Netty

Java

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

Kotlin

```
val handler: HttpHandler = ...
val adapter = ReactorHttpHandlerAdapter(handler)
HttpServer.create().host(host).port(port).handle(adapter).bind().block()
```

Undertow

Java

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

Kotlin

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

Tomcat

Java

```
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();
```

Kotlin

```
val handler: HttpHandler = ...
val servlet = TomcatHttpHandlerAdapter(handler)

val server = Tomcat()
val base = File(System.getProperty("java.io.tmpdir"))
val rootContext = server.addContext("", base.absolutePath)
Tomcat.addServlet(rootContext, "main", servlet)
rootContext.addServletMappingDecoded("/", "main")
server.host = host
server.setPort(port)
server.start()
```

Jetty

Java

```
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();
```

Kotlin

```
val handler: HttpHandler = ...
val servlet = JettyHttpHandlerAdapter(handler)

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

val connector = ServerConnector(server)
connector.host = host
connector.port = port
server.addConnector(connector)
server.start()
```

Servlet 3.1+ Container

To deploy as a WAR to any Servlet 3.1+ container, you can extend and include `AbstractReactiveWebInitializer` in the WAR. That class wraps an `HttpHandler` with `ServletHttpHandlerAdapter` and registers that as a `Servlet`.

`WebHandler` API

The `org.springframework.web.server` package builds on the `HttpHandler` contract to provide a general-purpose web API for processing requests through a chain of multiple `WebExceptionHandler`, multiple `WebFilter`, and a single `WebHandler` component. The chain can be put together with `WebHttpHandlerBuilder` by simply pointing to a Spring `ApplicationContext` where components are auto-detected, and/or by registering components with the builder.

While `HttpHandler` has a simple goal to abstract the use of different HTTP servers, the `WebHandler` API aims to provide a broader set of features commonly used in web applications such as:

- User session with attributes.
- Request attributes.
- Resolved `Locale` or `Principal` for the request.
- Access to parsed and cached form data.
- Abstractions for multipart data.
- and more..

Special bean types

The table below lists the components that `WebHttpHandlerBuilder` can auto-detect in a Spring ApplicationContext, or that can be registered directly with it:

Bean name	Bean type	Count	Description
<any>	`WebExceptionHandler`	0..N	Provide handling for exceptions from the chain of `WebFilter` instances and the target `WebHandler`. For more details, see Exceptions.
<any>	`WebFilter`	0..N	Apply interception style logic to before and after the rest of the filter chain and the target `WebHandler`. For more details, see Filters.
`webHandler`	`WebHandler`	1	The handler for the request.
`webSessionManager`	`WebSessionManager`	0..1	The manager for `WebSession` instances exposed through a method on `ServerWebExchange`. `DefaultWebSessionManager` by default.

Bean name	Bean type	Count	Description
`serverCodecConfigurer`	`ServerCodecConfigurer`	0..1	For access to `HttpMessageReader` instances for parsing form data and multipart data that is then exposed through methods on `ServerWebExchange`. `ServerCodecConfigurer.create()` by default.
`localeContextResolver`	`LocaleContextResolver`	0..1	The resolver for `LocaleContext` exposed through a method on `ServerWebExchange`. `AcceptHeaderLocaleContextResolver` by default.
`forwardedHeaderTransformer`	`ForwardedHeaderTransformer`	0..1	For processing forwarded type headers, either by extracting and removing them or by removing them only. Not used by default.

Form Data

`ServerWebExchange` exposes the following method for access to form data:

Java

```
Mono<MultiValueMap<String, String>> getFormData();
```

Kotlin

```
suspend fun getFormData(): MultiValueMap<String, String>
```

The `DefaultServerWebExchange` uses the configured `HttpMessageReader` to parse form data (`application/x-www-form-urlencoded`) into a `MultiValueMap`. By default, `FormHttpMessageReader` is configured for use by the `ServerCodecConfigurer` bean (see the Web Handler API).

Multipart Data

Same as in Spring MVC

`ServerWebExchange` exposes the following method for access to multipart data:

Java

```
Mono<MultiValueMap<String, Part>> getMultipartData();
```

Kotlin

```
suspend fun getMultipartData(): MultiValueMap<String, Part>
```

The `DefaultServerWebExchange` uses the configured `HttpMessageReader<MultiValueMap<String, Part>>` to parse `multipart/form-data` content into a `MultiValueMap`. At present, Synchronoss NIO Multipart is the only third-party library supported and the only library we know for non-blocking parsing of multipart requests. It is enabled through the `ServerCodecConfigurer` bean (see the Web Handler API).

To parse multipart data in streaming fashion, you can use the `Flux<Part>` returned from an `HttpMessageReader<Part>` instead. For example, in an annotated controller, use of `@RequestPart` implies `Map`-like access to individual parts by name and, hence, requires parsing multipart data in full. By contrast, you can use `@RequestBody` to decode the content to `Flux<Part>` without collecting to a `MultiValueMap`.

Forwarded Headers

Same as in Spring MVC

As a request goes through proxies (such as load balancers), the host, port, and scheme may change, and that makes it a challenge, from a client perspective, to create links that point to the correct host, port, and scheme.

RFC 7239 defines the `Forwarded` HTTP header that proxies can use to provide information about the original request. There are other non-standard headers, too, including `X-Forwarded-Host`, `X-Forwarded-Port`, `X-Forwarded-Proto`, `X-Forwarded-Ssl`, and `X-Forwarded-Prefix`.

`ForwardedHeaderTransformer` is a component that modifies the host, port, and scheme of the request, based on forwarded headers, and then removes those headers. You can declare it as a bean with a name of `forwardedHeaderTransformer`, and it is detected and used.

There are security considerations for forwarded headers, since an application cannot know if the headers were added by a proxy, as intended, or by a malicious client. This is why a proxy at the boundary of trust should be configured to remove untrusted forwarded traffic coming from the outside. You can also configure the `ForwardedHeaderTransformer` with `removeOnly=true`, in which case it removes but does not use the headers.

In 5.1 `ForwardedHeaderFilter` was deprecated and superceded by `ForwardedHeaderTransformer` so forwarded headers can be processed earlier, before the exchange is created. If the filter is configured anyway, it is taken out of the list of filters, and `ForwardedHeaderTransformer` is used instead.

Filters

Same as in Spring MVC

In the `WebHandler` API, you can use a `WebFilter` to apply interception-style logic before and after the rest of the processing chain of filters and the target `WebHandler`. When using the WebFlux Config, registering a `WebFilter` is as simple as declaring it as a Spring bean and (optionally) expressing

precedence by using `@Order` on the bean declaration or by implementing `Ordered`.

CORS

Same as in Spring MVC

Spring WebFlux provides fine-grained support for CORS configuration through annotations on controllers. However, when you use it with Spring Security, we advise relying on the built-in `CorsFilter`, which must be ordered ahead of Spring Security's chain of filters.

See the section on CORS and the CORS `WebFilter` for more details.

Exceptions

Same as in Spring MVC

In the `WebHandler` API, you can use a `WebExceptionHandler` to handle exceptions from the chain of `WebFilter` instances and the target `WebHandler`. When using the WebFlux Config, registering a `WebExceptionHandler` is as simple as declaring it as a Spring bean and (optionally) expressing precedence by using `@Order` on the bean declaration or by implementing `Ordered`.

The following table describes the available `WebExceptionHandler` implementations:

Exception Handler	Description
`ResponseStatusExceptionHandler`	Provides handling for exceptions of type `ResponseStatusException` by setting the response to the HTTP status code of the exception.
`WebFluxResponseStatusExceptionHandler`	Extension of `ResponseStatusExceptionHandler` that can also determine the HTTP status code of a `@ResponseStatus` annotation on any exception. This handler is declared in the WebFlux Config.

Codecs

Same as in Spring MVC

The `spring-web` and `spring-core` modules provide support for serializing and deserializing byte content to and from higher level objects through non-blocking I/O with Reactive Streams back pressure. The following describes this support:

- `Encoder` and `Decoder` are low level contracts to encode and decode content independent of HTTP.
- `HttpMessageReader` and `HttpMessageWriter` are contracts to encode and decode HTTP message content.
- An `Encoder` can be wrapped with `EncoderHttpMessageWriter` to adapt it for use in a web application, while a `Decoder` can be wrapped with `DecoderHttpMessageReader`.
- `DataBuffer` abstracts different byte buffer representations (e.g. Netty `ByteBuf`, `java.nio.ByteBuffer`, etc.) and is what all codecs work on. See Data Buffers and Codecs in the

"Spring Core" section for more on this topic.

The `spring-core` module provides `byte[]`, `ByteBuffer`, `DataBuffer`, `Resource`, and `String` encoder and decoder implementations. The `spring-web` module provides Jackson JSON, Jackson Smile, JAXB2, Protocol Buffers and other encoders and decoders along with web-only HTTP message reader and writer implementations for form data, multipart content, server-sent events, and others.

`ClientCodecConfigurer` and `ServerCodecConfigurer` are typically used to configure and customize the codecs to use in an application. See the section on configuring HTTP message codecs.

Jackson JSON

JSON and binary JSON (Smile) are both supported when the Jackson library is present.

The `Jackson2Decoder` works as follows:

- Jackson's asynchronous, non-blocking parser is used to aggregate a stream of byte chunks into `TokenBuffer`'s each representing a JSON object.
- Each `TokenBuffer` is passed to Jackson's `ObjectMapper` to create a higher level object.
- When decoding to a single-value publisher (e.g. `Mono`), there is one `TokenBuffer`.
- When decoding to a multi-value publisher (e.g. `Flux`), each `TokenBuffer` is passed to the `ObjectMapper` as soon as enough bytes are received for a fully formed object. The input content can be a JSON array, or line-delimited JSON if the content-type is "application/stream+json".

The `Jackson2Encoder` works as follows:

- For a single value publisher (e.g. `Mono`), simply serialize it through the `ObjectMapper`.
- For a multi-value publisher with "application/json", by default collect the values with `Flux#collectToList()` and then serialize the resulting collection.
- For a multi-value publisher with a streaming media type such as `application/stream+json` or `application/stream+x-jackson-smile`, encode, write, and flush each value individually using a line-delimited JSON format.
- For SSE the `Jackson2Encoder` is invoked per event and the output is flushed to ensure delivery without delay.

By default both `Jackson2Encoder` and `Jackson2Decoder` do not support elements of type `String`. Instead the default assumption is that a string or a sequence of strings represent serialized JSON content, to be rendered by the `CharSequenceEncoder`. If what you need is to render a JSON array from `Flux<String>`, use `Flux#collectToList()` and encode a `Mono<List<String>>`.

Form Data

`FormHttpMessageReader` and `FormHttpMessageWriter` support decoding and encoding "application/x-www-form-urlencoded" content.

On the server side where form content often needs to be accessed from multiple places,

`ServerWebExchange` provides a dedicated `getFormData()` method that parses the content through `FormHttpMessageReader` and then caches the result for repeated access. See Form Data in the `WebHandler` API section.

Once `getFormData()` is used, the original raw content can no longer be read from the request body. For this reason, applications are expected to go through `ServerWebExchange` consistently for access to the cached form data versus reading from the raw request body.

Multipart

`MultipartHttpMessageReader` and `MultipartHttpMessageWriter` support decoding and encoding "multipart/form-data" content. In turn `MultipartHttpMessageReader` delegates to another `HttpMessageReader` for the actual parsing to a `Flux<Part>` and then simply collects the parts into a `MultiValueMap`. At present the Synchronoss NIO Multipart is used for the actual parsing.

On the server side where multipart form content may need to be accessed from multiple places, `ServerWebExchange` provides a dedicated `getMultipartData()` method that parses the content through `MultipartHttpMessageReader` and then caches the result for repeated access. See Multipart Data in the `WebHandler` API section.

Once `getMultipartData()` is used, the original raw content can no longer be read from the request body. For this reason applications have to consistently use `getMultipartData()` for repeated, map-like access to parts, or otherwise rely on the `SynchronossPartHttpMessageReader` for a one-time access to `Flux<Part>`.

Streaming

Same as in Spring MVC

When streaming to the HTTP response (for example, `text/event-stream`, `application/stream+json`), it is important to send data periodically, in order to reliably detect a disconnected client sooner rather than later. Such a send could be an comment-only, empty SSE event or any other "no-op" data that would effectively serve as a heartbeat.

`DataBuffer`

`DataBuffer` is the representation for a byte buffer in WebFlux. The Spring Core part of the reference has more on that in the section on Data Buffers and Codecs. The key point to understand is that on some servers like Netty, byte buffers are pooled and reference counted, and must be released when consumed to avoid memory leaks.

WebFlux applications generally do not need to be concerned with such issues, unless they consume or produce data buffers directly, as opposed to relying on codecs to convert to and from higher level objects. Or unless they choose to create custom codecs. For such cases please review the the information in Data Buffers and Codecs, especially the section on Using DataBuffer.

Logging

Same as in Spring MVC

DEBUG level logging in Spring WebFlux is designed to be compact, minimal, and human-friendly. It focuses on high value bits of information that are useful over and over again vs others that are useful only when debugging a specific issue.

TRACE level logging generally follows the same principles as DEBUG (and for example also should not be a firehose) but can be used for debugging any issue. In addition some log messages may show a different level of detail at TRACE vs DEBUG.

Good logging comes from the experience of using the logs. If you spot anything that does not meet the stated goals, please let us know.

Log Id

In WebFlux, a single request can be executed over multiple threads and the thread ID is not useful for correlating log messages that belong to a specific request. This is why WebFlux log messages are prefixed with a request-specific ID by default.

On the server side, the log ID is stored in the `ServerWebExchange` attribute (`LOG_ID_ATTRIBUTE`), while a fully formatted prefix based on that ID is available from `ServerWebExchange#getLogPrefix()`. On the `WebClient` side, the log ID is stored in the `ClientRequest` attribute (`LOG_ID_ATTRIBUTE`) ,while a fully formatted prefix is available from `ClientRequest#logPrefix()`.

Sensitive Data

Same as in Spring MVC

`DEBUG` and `TRACE` logging can log sensitive information. This is why form parameters and headers are masked by default and you must explicitly enable their logging in full.

The following example shows how to do so for server-side requests:

Java

```
@Configuration
@EnableWebFlux
class MyConfig implements WebFluxConfigurer {

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        configurer.defaultCodecs().enableLoggingRequestDetails(true);
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class MyConfig : WebFluxConfigurer {

    override fun configureHttpMessageCodecs(configurer: ServerCodecConfigurer) {
        configurer.defaultCodecs().enableLoggingRequestDetails(true)
    }
}
```

The following example shows how to do so for client-side requests:

Java

```
Consumer<ClientCodecConfigurer> consumer = configurer ->
        configurer.defaultCodecs().enableLoggingRequestDetails(true);

WebClient webClient = WebClient.builder()
        .exchangeStrategies(ExchangeStrategies.builder().codecs(consumer).build())
        .build();
```

Kotlin

```
val consumer: (ClientCodecConfigurer) -> Unit  = { configurer ->
configurer.defaultCodecs().enableLoggingRequestDetails(true) }

val webClient = WebClient.builder()
        .exchangeStrategies(ExchangeStrategies.builder().codecs(consumer).build())
        .build()
```

DispatcherHandler

Same as in Spring MVC

Spring WebFlux, similarly to 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 in which it runs. If `DispatcherHandler` is declared with a bean name of `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, as the following example shows:

Java

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

Kotlin

```
val context: ApplicationContext = ...
val handler = WebHttpHandlerBuilder.applicationContext(context)
```

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

Special Bean Types

Same as 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 WebFlux framework contracts. Those usually come with built-in contracts, but you can customize their properties, extend them, or replace them.

The following table lists the special beans detected by the `DispatcherHandler`. Note that there are also some other beans detected at a lower level (see Special bean types in the Web Handler API).

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, and others. The main `HandlerMapping` implementations are `RequestMappingHandlerMapping` for `@RequestMapping` annotated methods, `RouterFunctionMapping` for functional endpoint routes, and `SimpleUrlHandlerMapping` for explicit registrations of URI path patterns and `WebHandler` instances.

Bean type	Explanation
`HandlerAdapter`	Help 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 annotations. The main purpose of a `HandlerAdapter` is to shield the `DispatcherHandler` from such details.
`HandlerResultHandler`	Process the result from the handler invocation and finalize the response. See Result Handling.

WebFlux Config

Same as in Spring MVC

Applications can declare the infrastructure beans (listed under Web Handler API and `DispatcherHandler`) that are required to process requests. However, in most cases, the WebFlux Config is the best starting point. It declares the required beans and provides a higher-level configuration callback API to customize it.

Spring Boot relies on the WebFlux config to configure Spring WebFlux and also provides many extra convenient options.

Processing

Same as in Spring MVC

`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 by using a view to render.

Result Handling

The return value from the invocation of a handler, through a `HandlerAdapter`, is wrapped as a `HandlerResult`, along with some additional context, and passed to the first `HandlerResultHandler` that claims support for it. The following table shows the available `HandlerResultHandler` implementations, all of which are declared in the WebFlux Config:

Result Handler Type	Return Values	Default Order
`ResponseEntityResultHandler`	`ResponseEntity`, typically from `@Controller` instances.	0
`ServerResponseResultHandler`	`ServerResponse`, typically from functional endpoints.	0

Result Handler Type	Return Values	Default Order
`ResponseBodyResultHandler`	Handle return values from `@ResponseBody` methods or `@RestController` classes.	100
`ViewResolutionResultHandler`	`CharSequence`, `View`, Model, `Map`, Rendering, or any other `Object` is treated as a model attribute. See also View Resolution.	`Integer.MAX_VALUE`

Exceptions

Same as in Spring MVC

The `HandlerResult` returned from a `HandlerAdapter` can expose a function for error handling based on some handler-specific mechanism. This error function is called if:

- The handler (for example, `@Controller`) invocation fails.
- The handling of the handler return value through a `HandlerResultHandler` fails.

The error function can change the response (for example, to an error status), as long as an error signal occurs before the reactive type returned from the handler produces any data items.

This is how `@ExceptionHandler` methods in `@Controller` classes are supported. By contrast, support for the same in Spring MVC is built on a `HandlerExceptionResolver`. This generally should not matter. However, keep in mind that, in WebFlux, you cannot use a `@ControllerAdvice` to handle exceptions that occur before a handler is chosen.

See also Managing Exceptions in the "Annotated Controller" section or Exceptions in the WebHandler API section.

View Resolution

Same as in Spring MVC

View resolution enables rendering to a browser with an HTML template and a model without tying you to a specific view technology. In Spring WebFlux, view resolution is supported through a dedicated HandlerResultHandler that uses `ViewResolver` instances to map a String (representing a logical view name) to a `View` instance. The `View` is then used to render the response.

Handling

Same as in Spring MVC

The `HandlerResult` passed into `ViewResolutionResultHandler` contains the return value from the handler and the model that contains attributes added during request handling. The return value is processed as one of the following:

- `String`, `CharSequence`: A logical view name to be resolved to a `View` through the list of configured `ViewResolver` implementations.

- `void`: Select a default view name based on the request path, minus the leading and trailing slash, and resolve it to a `View`. The same also happens when a view name was not provided (for example, model attribute was returned) or an async return value (for example, `Mono` completed empty).
- Rendering: API for view resolution scenarios. Explore the options in your IDE with code completion.
- `Model`, `Map`: Extra model attributes to be added to the model for the request.
- Any other: Any other return value (except for simple types, as determined by BeanUtils#isSimpleProperty) is treated as a model attribute to be added to the model. The attribute name is derived from the class name by using conventions, unless a handler method `@ModelAttribute` annotation is present.

The model can contain asynchronous, reactive types (for example, from Reactor or RxJava). Prior to rendering, `AbstractView` resolves such model attributes into concrete values and updates the model. Single-value reactive types are resolved to a single value or no value (if empty), while multi-value reactive types (for example, `Flux<T>`) are collected and resolved to `List<T>`.

To configure view resolution is as simple as adding a `ViewResolutionResultHandler` bean to your Spring configuration. WebFlux Config provides a dedicated configuration API for view resolution.

See View Technologies for more on the view technologies integrated with Spring WebFlux.

Redirecting

Same as in Spring MVC

The special `redirect:` prefix in a view name lets you perform a redirect. The `UrlBasedViewResolver` (and sub-classes) recognize this as an instruction that a redirect is needed. The rest of the view name is the redirect URL.

The net effect is the same as if the controller had returned a `RedirectView` or `Rendering.redirectTo("abc").build()`, but now the controller itself can operate in terms of logical view names. A view name such as `redirect:/some/resource` is relative to the current application, while a view name such as `redirect:https://example.com/arbitrary/path` redirects to an absolute URL.

Content Negotiation

Same as in Spring MVC

`ViewResolutionResultHandler` supports content negotiation. It compares the request media types with the media types supported by each selected `View`. The first `View` that supports the requested media type(s) is used.

In order to support media types such as JSON and XML, Spring WebFlux provides `HttpMessageWriterView`, which is a special `View` that renders through an HttpMessageWriter. Typically, you would configure these as default views through the WebFlux Configuration. Default views are always selected and used if they match the requested media type.

Annotated Controllers

Same as in Spring MVC

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

The following listing shows a basic example:

Java

```
@RestController
public class HelloController {

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

Kotlin

```
@RestController
class HelloController {

    @GetMapping("/hello")
    fun handle() = "Hello WebFlux"
}
```

In the preceding example, the method returns a `String` to be written to the response body.

`@Controller`

Same as in Spring MVC

You can define controller beans by using a standard Spring bean definition. The `@Controller` stereotype allows for auto-detection and is 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, as the following example shows:

Java

```
@Configuration
@ComponentScan("org.example.web") ①
public class WebConfig {

    // ...
}
```

① Scan the `org.example.web` package.

Kotlin

```
@Configuration
@ComponentScan("org.example.web") ①
class WebConfig {

    // ...
}
```

① Scan the `org.example.web` package.

`@RestController` is a composed annotation that is itself meta-annotated with `@Controller` and `@ResponseBody`, indicating a controller whose every method inherits the type-level `@ResponseBody` annotation and, therefore, writes directly to the response body versus view resolution and rendering with an HTML template.

Request Mapping

Same as 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. You can use it 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 preceding annotations are Custom Annotations that are provided because, arguably, most controller methods should be mapped to a specific HTTP method versus using `@RequestMapping`, which, by default, matches to all HTTP methods. At the same time, a `@RequestMapping` is still needed at the class level to express shared mappings.

The following example uses type and method level mappings:

Java

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

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

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

Kotlin

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

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

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

URI Patterns

Same as in Spring MVC

You can map requests by 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`, as the following example shows:

Java

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

Kotlin

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

You can declare URI variables at the class and method levels, as the following example shows:

Java

```
@Controller
@RequestMapping("/owners/{ownerId}") ①
public class OwnerController {

    @GetMapping("/pets/{petId}") ②
    public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
        // ...
    }
}
```

① Class-level URI mapping.

② Method-level URI mapping.

Kotlin

```
@Controller
@RequestMapping("/owners/{ownerId}") ①
class OwnerController {

    @GetMapping("/pets/{petId}") ②
    fun findPet(@PathVariable ownerId: Long, @PathVariable petId: Long): Pet {
        // ...
    }
}
```

① Class-level URI mapping.

② Method-level URI mapping.

URI variables are automatically converted to the appropriate type or a `TypeMismatchException` is raised. Simple types (`int`, `long`, `Date`, and so on) are supported by default and you can register support for any other data type. See Type Conversion and `DataBinder`.

URI variables can be named explicitly (for example, `@PathVariable("customId")`), but you can leave that detail out if the names are the same and you compile your code 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 expression that has the syntax: `{varName:regex}`. For example, given a URL of `/spring-web-3.0.5 .jar`, the following method extracts the name, version, and file extension:

Java

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

Kotlin

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

URI path patterns can also have embedded `${…}` placeholders that are resolved on startup through `PropertyPlaceHolderConfigurer` against local, system, environment, and other property sources. You ca use this to, for example, parameterize a base URL based on some external configuration.

Spring WebFlux uses `PathPattern` and the `PathPatternParser` for URI path matching support. Both classes are located in `spring-web` and are 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 simpler, more explicit, and less vulnerable to URL path based exploits.

Pattern Comparison

Same as 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 are more specific.

For every pattern, a score is computed, based on 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, the longer is chosen.

Catch-all patterns (for example, `**`, `{*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 as in Spring MVC

You can narrow the request mapping based on the `Content-Type` of the request, as the following example shows:

Java

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

Kotlin

```
@PostMapping("/pets", consumes = ["application/json"])
fun addPet(@RequestBody pet: Pet) {
    // ...
}
```

The consumes attribute also supports negation expressions — for example, `!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 overrides rather than extends the class-level declaration.

`MediaType` provides constants for commonly used media types — for example, `APPLICATION_JSON_VALUE` and `APPLICATION_XML_VALUE`.

Producible Media Types

Same as 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, as the following example shows:

Java

```
@GetMapping(path = "/pets/{petId}", produces = "application/json")
@ResponseBody
public Pet getPet(@PathVariable String petId) {
    // ...
}
```

Kotlin

```
@GetMapping("/pets/{petId}", produces = ["application/json"])
@ResponseBody
fun getPet(@PathVariable String petId): Pet {
    // ...
}
```

The media type can specify a character set. Negated expressions are supported — for example, `!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 overrides rather than extend the class level declaration.

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

Parameters and Headers

Same as 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 its absence (`!myParam`), or for a specific value (`myParam=myValue`). The following examples tests for a parameter with a value:

Java

```
@GetMapping(path = "/pets/{petId}", params = "myParam=myValue") ①
public void findPet(@PathVariable String petId) {
    // ...
}
```

① Check that `myParam` equals `myValue`.

Kotlin

```
@GetMapping("/pets/{petId}", params = ["myParam=myValue"]) ①
fun findPet(@PathVariable petId: String) {
    // ...
}
```

① Check that `myParam` equals `myValue`.

You can also use the same with request header conditions, as the follwing example shows:

Java

```
@GetMapping(path = "/pets", headers = "myHeader=myValue") ①
public void findPet(@PathVariable String petId) {
    // ...
}
```

① Check that `myHeader` equals `myValue`.

Kotlin

```
@GetMapping("/pets", headers = ["myHeader=myValue"]) ①
fun findPet(@PathVariable petId: String) {
    // ...
}
```

① Check that `myHeader` equals `myValue`.

HTTP HEAD, OPTIONS

Same as in Spring MVC

`@GetMapping` and `@RequestMapping(method=HttpMethod.GET)` support HTTP HEAD transparently for request mapping purposes. Controller methods need not change. A response wrapper, applied in the `HttpHandler` server adapter, ensures a `Content-Length` header is set to the number of bytes written 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`, and others).

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

Custom Annotations

Same as in Spring MVC

Spring WebFlux supports the use of composed annotations for request mapping. Those are annotations that are themselves meta-annotated with `@RequestMapping` and composed to redeclare a subset (or all) of the `@RequestMapping` attributes with a narrower, more specific purpose.

`@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, and `@PatchMapping` are examples of composed annotations. They are provided, because, arguably, most controller methods should be mapped to a specific HTTP method versus using `@RequestMapping`, which, by default, matches to all HTTP methods. If you need an example of composed annotations, look at how those are declared.

Spring WebFlux also supports custom request mapping attributes with custom request matching logic. This is a more advanced option that requires sub-classing `RequestMappingHandlerMapping` and overriding the `getCustomMethodCondition` method, where you can check the custom attribute and return your own `RequestCondition`.

Explicit Registrations

Same as in Spring MVC

You can programmatically register Handler methods, which can be used for dynamic registrations or for advanced cases, such as different instances of the same handler under different URLs. The following example shows how to do so:

Java

```
@Configuration
public class MyConfig {

    @Autowired
    public void setHandlerMapping(RequestMappingHandlerMapping mapping, UserHandler
handler) ①
            throws NoSuchMethodException {

        RequestMappingInfo info = RequestMappingInfo
                .paths("/user/{id}").methods(RequestMethod.GET).build(); ②

        Method method = UserHandler.class.getMethod("getUser", Long.class); ③

        mapping.registerMapping(info, handler, method); ④
    }

}
```

① Inject target handlers and the handler mapping for controllers.

② Prepare the request mapping metadata.

③ Get the handler method.

④ Add the registration.

Kotlin

```
@Configuration
class MyConfig {

    @Autowired
    fun setHandlerMapping(mapping: RequestMappingHandlerMapping, handler: UserHandler)
{ ①

        val info =
RequestMappingInfo.paths("/user/{id}").methods(RequestMethod.GET).build() ②

        val method = UserHandler::class.java.getMethod("getUser", Long::class.java) ③

        mapping.registerMapping(info, handler, method) ④
    }
}
```

① Inject target handlers and the handler mapping for controllers.

② Prepare the request mapping metadata.

③ Get the handler method.

④ Add the registration.

Handler Methods

Same as 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 as in Spring MVC

The following table shows the supported controller method arguments.

Reactive types (Reactor, RxJava, or other) are supported on arguments that require blocking I/O (for example, reading the request body) to be resolved. This is marked in the Description column. Reactive types are not expected on arguments that do not require blocking.

JDK 1.8's `java.util.Optional` is supported as a method argument in combination with annotations that have a `required` attribute (for example, `@RequestParam`, `@RequestHeader`, and others) 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.

Controller method argument	Description
`ServerHttpRequest`, `ServerHttpResponse`	Access to the HTTP request or response.
`WebSession`	Access to the session. This does not force the start of a new session unless attributes are added. Supports reactive types.
`java.security.Principal`	The currently authenticated user — possibly a specific `Principal` implementation class if known. Supports reactive types.
`org.springframework.http.HttpMethod`	The HTTP method of the request.
`java.util.Locale`	The current request locale, determined by the most specific `LocaleResolver` available — in effect, the configured `LocaleResolver`/`LocaleContextResolver`.
`java.util.TimeZone` + `java.time.ZoneId`	The time zone associated with the current request, as determined by a `LocaleContextResolver`.
`@PathVariable`	For access to URI template variables. See URI Patterns.
`@MatrixVariable`	For access to name-value pairs in URI path segments. See Matrix Variables.
`@RequestParam`	For access to Servlet request parameters. Parameter values are converted to the declared method argument type. See `@RequestParam`. Note that use of `@RequestParam` is optional — for example, to set its attributes. See “Any other argument” later in this table.
`@RequestHeader`	For access to request headers. Header values are converted to the declared method argument type. See `@RequestHeader`.
`@CookieValue`	For access to cookies. Cookie values are converted to the declared method argument type. See `@CookieValue`.
`@RequestBody`	For access to the HTTP request body. Body content is converted to the declared method argument type by using `HttpMessageReader` instances. Supports reactive types. See `@RequestBody`.
`HttpEntity<B>`	For access to request headers and body. The body is converted with `HttpMessageReader` instances. Supports reactive types. See `HttpEntity`.
`@RequestPart`	For access to a part in a `multipart/form-data` request. Supports reactive types. See Multipart Content and Multipart Data.
`java.util.Map`, `org.springframework.ui.Model`, and `org.springframework.ui.ModelMap`.	For access to the model that is used in HTML controllers and is exposed to templates as part of view rendering.

Controller method argument	Description
`@ModelAttribute`	For access to an existing attribute in the model (instantiated if not present) with data binding and validation applied. See `@ModelAttribute` as well as `Model` and `DataBinder`. Note that use of `@ModelAttribute` is optional — for example, to set its attributes. See “Any other argument” later in this table.
`Errors`, `BindingResult`	For access to errors from validation and data binding for a command object (that is, a `@ModelAttribute` argument) or errors from the validation of a `@RequestBody` or `@RequestPart` argument. An `Errors`, or `BindingResult` argument must be declared immediately after the validated method argument.
`SessionStatus` + class-level `@SessionAttributes`	For marking form processing complete, which triggers cleanup of session attributes declared through a class-level `@SessionAttributes` annotation. See `@SessionAttributes` for more details.
`UriComponentsBuilder`	For preparing a URL relative to the current request’s host, port, scheme, and path. See URI Links.
`@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. See `@SessionAttribute` for more details.
`@RequestAttribute`	For access to request attributes. See `@RequestAttribute` for more details.
Any other argument	If a method argument is not matched to any of the above, it is, by default, resolved as a `@RequestParam` if it is a simple type, as determined by BeanUtils#isSimpleProperty, or as a `@ModelAttribute`, otherwise.

Return Values

Same as in Spring MVC

The following table shows the supported controller method return values. Note that reactive types from libraries such as Reactor, RxJava, or other are generally supported for all return values.

Controller method return value	Description
`@ResponseBody`	The return value is encoded through `HttpMessageWriter` instances and written to the response. See `@ResponseBody`.
`HttpEntity<B>`, `ResponseEntity<B>`	The return value specifies the full response, including HTTP headers, and the body is encoded through `HttpMessageWriter` instances and written to the response. See `ResponseEntity`.
`HttpHeaders`	For returning a response with headers and no body.

Controller method return value	Description
`String`	A view name to be resolved with `ViewResolver` instances and used together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method can also programmatically enrich the model by declaring a `Model` argument (described earlier).
`View`	A `View` instance to use for rendering together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method can also programmatically enrich the model by declaring a `Model` argument (described earlier).
`java.util.Map`, `org.springframework.ui.Model`	Attributes to be added to the implicit model, with the view name implicitly determined based on the request path.
`@ModelAttribute`	An attribute to be added to the model, with the view name implicitly determined based on the request path. Note that `@ModelAttribute` is optional. See “Any other return value” later in this table.
`Rendering`	An API for model and view rendering scenarios.
`void`	A method with a `void`, possibly asynchronous (for example, `Mono<Void>`), return type (or a `null` return value) is considered to have fully handled the response if it also has a `ServerHttpResponse`, a `ServerWebExchange` argument, or an `@ResponseStatus` annotation. The same is also true if the controller has made a positive ETag or `lastModified` timestamp check. // TODO: See Controllers for details. If none of the above is true, a `void` return type can also indicate “no response body” for REST controllers or default view name selection for HTML controllers.
`Flux<ServerSentEvent>`, `Observable<ServerSentEvent>`, or other reactive type	Emit server-sent events. The `ServerSentEvent` 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 value	If a return value is not matched to any of the above, it is, by default, treated as a view name, if it is `String` or `void` (default view name selection applies), or as a model attribute to be added to the model, unless it is a simple type, as determined by BeanUtils#isSimpleProperty, in which case it remains unresolved.

Type Conversion

Same as in Spring MVC

Some annotated controller method arguments that represent String-based request input (for example, `@RequestParam`, `@RequestHeader`, `@PathVariable`, `@MatrixVariable`, and `@CookieValue`) can require type conversion if the argument is declared as something other than `String`.

For such cases, type conversion is automatically applied based on the configured converters. By default, simple types (such as `int`, `long`, `Date`, and others) are supported. Type conversion can be customized through a `WebDataBinder` (see [mvc-ann-initbinder]) or by registering `Formatters` with the `FormattingConversionService` (see Spring Field Formatting).

Matrix Variables

Same as in Spring MVC

RFC 3986 discusses name-value pairs in path segments. In Spring WebFlux, we refer to those as "matrix variables" based on an "old post" by Tim Berners-Lee, but they can be also be referred to as URI path parameters.

Matrix variables can appear in any path segment, with each variable separated by a semicolon and multiple values separated by commas — for example, `"/cars;color=red,green;year=2012"`. Multiple values can also be specified through repeated variable names — for example, `"color=red;color=green;color=blue"`.

Unlike Spring MVC, in WebFlux, the presence or absence of matrix variables in a URL does not affect request mappings. In other words, you are not required to use a URI variable to mask variable content. That said, if you want to access matrix variables from a controller method, you need to add a URI variable to the path segment where matrix variables are expected. The following example shows how to do so:

Java

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
public void findPet(@PathVariable String petId, @MatrixVariable int q) {

    // petId == 42
    // q == 11
}
```

Kotlin

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
fun findPet(@PathVariable petId: String, @MatrixVariable q: Int) {

    // petId == 42
    // q == 11
}
```

Given that all path segments can contain matrix variables, you may sometimes need to disambiguate which path variable the matrix variable is expected to be in, as the following example shows:

Java

```
// GET /owners/42;q=11/pets/21;q=22

@GetMapping("/owners/{ownerId}/pets/{petId}")
public void findPet(
        @MatrixVariable(name="q", pathVar="ownerId") int q1,
        @MatrixVariable(name="q", pathVar="petId") int q2) {

    // q1 == 11
    // q2 == 22
}
```

Kotlin

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
fun findPet(
        @MatrixVariable(name = "q", pathVar = "ownerId") q1: Int,
        @MatrixVariable(name = "q", pathVar = "petId") q2: Int) {

    // q1 == 11
    // q2 == 22
}
```

You can define a matrix variable may be defined as optional and specify a default value as the following example shows:

Java

```
// GET /pets/42

@GetMapping("/pets/{petId}")
public void findPet(@MatrixVariable(required=false, defaultValue="1") int q) {

    // q == 1
}
```

Kotlin

```
// GET /pets/42

@GetMapping("/pets/{petId}")
fun findPet(@MatrixVariable(required = false, defaultValue = "1") q: Int) {

    // q == 1
}
```

To get all matrix variables, use a `MultiValueMap`, as the following example shows:

Java

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@GetMapping("/owners/{ownerId}/pets/{petId}")
public void findPet(
        @MatrixVariable MultiValueMap<String, String> matrixVars,
        @MatrixVariable(pathVar="petId") MultiValueMap<String, String> petMatrixVars)
{

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]
    // petMatrixVars: ["q" : 22, "s" : 23]
}
```

Kotlin

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@GetMapping("/owners/{ownerId}/pets/{petId}")
fun findPet(
        @MatrixVariable matrixVars: MultiValueMap<String, String>,
        @MatrixVariable(pathVar="petId") petMatrixVars: MultiValueMap<String, String>)
{

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]
    // petMatrixVars: ["q" : 22, "s" : 23]
}
```

`@RequestParam`

Same as in Spring MVC

You can use the `@RequestParam` annotation to bind query parameters to a method argument in a controller. The following code snippet shows the usage:

Java

```
@Controller
@RequestMapping("/pets")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, Model model) { ①
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...
}
```

① Using `@RequestParam`.

Kotlin

```
import org.springframework.ui.set

@Controller
@RequestMapping("/pets")
class EditPetForm {

    // ...

    @GetMapping
    fun setupForm(@RequestParam("petId") petId: Int, model: Model): String { ①
        val pet = clinic.loadPet(petId)
        model["pet"] = pet
        return "petForm"
    }

    // ...
}
```

① Using `@RequestParam`.

The Servlet API "request parameter" concept conflates query parameters, form data, and multiparts into one. However, in WebFlux, each is accessed individually through `ServerWebExchange`. While `@RequestParam` binds to query parameters only, you can use data binding to apply query parameters, form data, and multiparts to a command object.

Method parameters that use the `@RequestParam` annotation are required by default, but you can specify that a method parameter is optional by setting the required flag of a `@RequestParam` to `false` or by declaring the argument with a `java.util.Optional` wrapper.

Type conversion is applied automatically if the target method parameter type is not `String`. See [mvc-ann-typeconversion].

When a `@RequestParam` annotation is declared on a `Map<String, String>` or `MultiValueMap<String, String>` argument, the map is populated with all query parameters.

Note that use of `@RequestParam` is optional — for example, to set its attributes. By default, any argument that is a simple value type (as determined by BeanUtils#isSimpleProperty) and is not resolved by any other argument resolver is treated as if it were annotated with `@RequestParam`.

`@RequestHeader`

Same as in Spring MVC

You can use the `@RequestHeader` annotation to bind a request header to a method argument in a controller.

The following example shows a request with headers:

```
Host                    localhost:8080
Accept                  text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Language         fr,en-gb;q=0.7,en;q=0.3
Accept-Encoding         gzip,deflate
Accept-Charset          ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive              300
```

The following example gets the value of the `Accept-Encoding` and `Keep-Alive` headers:

Java

```
@GetMapping("/demo")
public void handle(
        @RequestHeader("Accept-Encoding") String encoding, ①
        @RequestHeader("Keep-Alive") long keepAlive) { ②
    //...
}
```

① Get the value of the `Accept-Encoging` header.

② Get the value of the `Keep-Alive` header.

Kotlin

```
@GetMapping("/demo")
fun handle(
        @RequestHeader("Accept-Encoding") encoding: String, ①
        @RequestHeader("Keep-Alive") keepAlive: Long) { ②
    //...
}
```

① Get the value of the `Accept-Encoging` header.

② Get the value of the `Keep-Alive` header.

Type conversion is applied automatically if the target method parameter type is not `String`. See [mvc-ann-typeconversion].

When a `@RequestHeader` annotation is used on a `Map<String, String>`, `MultiValueMap<String, String>`, or `HttpHeaders` argument, the map is populated with all header values.

Built-in support is available for converting a comma-separated string into an array or collection of strings or other types known to the type conversion system. For example, a method parameter annotated with `@RequestHeader("Accept")` may be of type `String` but also of `String[]` or `List<String>`.

`@CookieValue`

Same as in Spring MVC

You can use the `@CookieValue` annotation to bind the value of an HTTP cookie to a method argument in a controller.

The following example shows a request with a cookie:

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

The following code sample demonstrates how to get the cookie value:

Java

```
@GetMapping("/demo")
public void handle(@CookieValue("JSESSIONID") String cookie) { ①
    //...
}
```

① Get the cookie value.

Kotlin

```
@GetMapping("/demo")
fun handle(@CookieValue("JSESSIONID") cookie: String) { ①
    //...
}
```

① Get the cookie value.

Type conversion is applied automatically if the target method parameter type is not `String`. See [mvc-ann-typeconversion].

`@ModelAttribute`

Same as in Spring MVC

You can use the `@ModelAttribute` annotation on a method argument to access an attribute from the model or have it instantiated if not present. The model attribute is also overlain with the values of query parameters and form fields whose names match to field names. This is referred to as data binding, and it saves you from having to deal with parsing and converting individual query parameters and form fields. The following example binds an instance of `Pet`:

Java

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute Pet pet) { } ①
```

① Bind an instance of `Pet`.

Kotlin

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@ModelAttribute pet: Pet): String { } ①
```

① Bind an instance of `Pet`.

The `Pet` instance in the preceding example is resolved as follows:

- From the model if already added through `Model`.
- From the HTTP session through `@SessionAttributes`.
- From the invocation of a default constructor.
- From the invocation of a "primary constructor" with arguments that match query parameters or form fields. Argument names are determined through JavaBeans `@ConstructorProperties` or through runtime-retained parameter names in the bytecode.

After the model attribute instance is obtained, data binding is applied. The `WebExchangeDataBinder` class matches names of query parameters and form fields to field names on the target `Object`. Matching fields are populated after type conversion is applied where necessary. For more on data binding (and validation), see Validation. For more on customizing data binding, see `DataBinder`.

Data binding can result in errors. By default, a `WebExchangeBindException` is raised, but, to check for such errors in the controller method, you can add a `BindingResult` argument immediately next to the `@ModelAttribute`, as the following example shows:

Java

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) { ①
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

① Adding a `BindingResult`.

Kotlin

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@ModelAttribute("pet") pet: Pet, result: BindingResult): String { ①
    if (result.hasErrors()) {
        return "petForm"
    }
    // ...
}
```

① Adding a `BindingResult`.

You can automatically apply validation after data binding by adding the `javax.validation.Valid` annotation or Spring's `@Validated` annotation (see also Bean Validation and Spring validation). The following example uses the `@Valid` annotation:

Java

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult
result) { ①
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

① Using `@Valid` on a model attribute argument.

Kotlin

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@Valid @ModelAttribute("pet") pet: Pet, result: BindingResult):
String { ①
    if (result.hasErrors()) {
        return "petForm"
    }
    // ...
}
```

① Using `@Valid` on a model attribute argument.

Spring WebFlux, unlike Spring MVC, supports reactive types in the model — for example, `Mono<Account>` or `io.reactivex.Single<Account>`. You can declare a `@ModelAttribute` argument with or without a reactive type wrapper, and it will be resolved accordingly, to the actual value if necessary. However, note that, to use a `BindingResult` argument, you must declare the `@ModelAttribute` argument before it without a reactive type wrapper, as shown earlier. Alternatively, you can handle any errors through the reactive type, as the following example shows:

Java

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public Mono<String> processSubmit(@Valid @ModelAttribute("pet") Mono<Pet> petMono) {
    return petMono
        .flatMap(pet -> {
            // ...
        })
        .onErrorResume(ex -> {
            // ...
        });
}
```

Kotlin

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@Valid @ModelAttribute("pet") petMono: Mono<Pet>): Mono<String> {
    return petMono
            .flatMap { pet ->
                // ...
            }
            .onErrorResume{ ex ->
                // ...
            }
}
```

Note that use of `@ModelAttribute` is optional — for example, to set its attributes. By default, any argument that is not a simple value type(as determined by BeanUtils#isSimpleProperty) and is not resolved by any other argument resolver is treated as if it were annotated with `@ModelAttribute`.

`@SessionAttributes`

Same as in Spring MVC

`@SessionAttributes` is used to store model attributes in the `WebSession` between requests. It is a type-level annotation that declares session attributes used by a specific controller. This typically lists the names of model attributes or types of model attributes that should be transparently stored in the session for subsequent requests to access.

Consider the following example:

Java

```
@Controller
@SessionAttributes("pet") ①
public class EditPetForm {
    // ...
}
```

① Using the `@SessionAttributes` annotation.

Kotlin

```
@Controller
@SessionAttributes("pet") ①
class EditPetForm {
    // ...
}
```

① Using the `@SessionAttributes` annotation.

On the first request, when a model attribute with the name, `pet`, is added to the model, it is automatically promoted to and saved in the `WebSession`. It remains there until another controller

method uses a `SessionStatus` method argument to clear the storage, as the following example shows:

Java

```
@Controller
@SessionAttributes("pet") ①
public class EditPetForm {

    // ...

    @PostMapping("/pets/{id}")
    public String handle(Pet pet, BindingResult errors, SessionStatus status) { ②
        if (errors.hasErrors()) {
            // ...
        }
            status.setComplete();
            // ...
        }
    }
}
```

① Using the `@SessionAttributes` annotation.

② Using a `SessionStatus` variable.

Kotlin

```
@Controller
@SessionAttributes("pet") ①
class EditPetForm {

    // ...

    @PostMapping("/pets/{id}")
    fun handle(pet: Pet, errors: BindingResult, status: SessionStatus): String { ②
        if (errors.hasErrors()) {
            // ...
        }
        status.setComplete()
        // ...
    }
}
```

① Using the `@SessionAttributes` annotation.

② Using a `SessionStatus` variable.

`@SessionAttribute`

Same as in Spring MVC

If you need access to pre-existing session attributes that are managed globally (that is, outside the controller — for example, by a filter) and may or may not be present, you can use the `@SessionAttribute` annotation on a method parameter, as the following example shows:

Java

```
@GetMapping("/")
public String handle(@SessionAttribute User user) { ①
    // ...
}
```

① Using `@SessionAttribute`.

Kotlin

```
@GetMapping("/")
fun handle(@SessionAttribute user: User): String { ①
    // ...
}
```

① Using `@SessionAttribute`.

For use cases that require adding or removing session attributes, consider injecting `WebSession` into the controller method.

For temporary storage of model attributes in the session as part of a controller workflow, consider using `SessionAttributes`, as described in `@SessionAttributes`.

`@RequestAttribute`

Same as in Spring MVC

Similarly to `@SessionAttribute`, you can use the `@RequestAttribute` annotation to access pre-existing request attributes created earlier (for example, by a `WebFilter`), as the following example shows:

Java

```
@GetMapping("/")
public String handle(@RequestAttribute Client client) { ①
    // ...
}
```

① Using `@RequestAttribute`.

Kotlin

```
@GetMapping("/")
fun handle(@RequestAttribute client: Client): String { ①
    // ...
}
```

① Using `@RequestAttribute`.

Multipart Content

Same as in Spring MVC

As explained in Multipart Data, `ServerWebExchange` provides access to multipart content. The best way to handle a file upload form (for example, from a browser) in a controller is through data binding to a command object, as the following example shows:

Java

```
class MyForm {

    private String name;

    private MultipartFile file;

    // ...

}

@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(MyForm form, BindingResult errors) {
        // ...
    }

}
```

Kotlin

```
class MyForm(
        val name: String,
        val file: MultipartFile)

@Controller
class FileUploadController {

    @PostMapping("/form")
    fun handleFormUpload(form: MyForm, errors: BindingResult): String {
        // ...
    }

}
```

You can also submit multipart requests from non-browser clients in a RESTful service scenario. The following example uses a file along with JSON:

```
POST /someUrl
Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="meta-data"
Content-Type: application/json; charset=UTF-8
Content-Transfer-Encoding: 8bit

{
    "name": "value"
}
--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="file-data"; filename="file.properties"
Content-Type: text/xml
Content-Transfer-Encoding: 8bit
... File Data ...
```

You can access individual parts with `@RequestPart`, as the following example shows:

Java

```
@PostMapping("/")
public String handle(@RequestPart("meta-data") Part metadata, ①
        @RequestPart("file-data") FilePart file) { ②
    // ...
}
```

① Using `@RequestPart` to get the metadata.

② Using `@RequestPart` to get the file.

Kotlin

```
@PostMapping("/")
fun handle(@RequestPart("meta-data") Part metadata, ①
        @RequestPart("file-data") FilePart file): String { ②
    // ...
}
```

① Using `@RequestPart` to get the metadata.

② Using `@RequestPart` to get the file.

To deserialize the raw part content (for example, to JSON — similar to `@RequestBody`), you can declare a concrete target `Object`, instead of `Part`, as the following example shows:

Java

```
@PostMapping("/")
public String handle(@RequestPart("meta-data") MetaData metadata) { ①
    // ...
}
```

① Using `@RequestPart` to get the metadata.

Kotlin

```
@PostMapping("/")
fun handle(@RequestPart("meta-data") metadata: MetaData): String { ①
    // ...
}
```

① Using `@RequestPart` to get the metadata.

You can use `@RequestPart` combination with `javax.validation.Valid` or Spring's `@Validated` annotation, which causes Standard Bean Validation to be applied. By default, validation errors cause a `WebExchangeBindException`, which is turned into a 400 (`BAD_REQUEST`) response. Alternatively, you can handle validation errors locally within the controller through an `Errors` or `BindingResult` argument, as the following example shows:

Java

```
@PostMapping("/")
public String handle(@Valid @RequestPart("meta-data") MetaData metadata, ①
        BindingResult result) { ②
    // ...
}
```

① Using a `@Valid` annotation.

② Using a `BindingResult` argument.

Kotlin

```
@PostMapping("/")
fun handle(@Valid @RequestPart("meta-data") metadata: MetaData, ①
        result: BindingResult): String { ②
    // ...
}
```

① Using a `@Valid` annotation.

② Using a `BindingResult` argument.

To access all multipart data as a `MultiValueMap`, you can use `@RequestBody`, as the following example shows:

Java

```
@PostMapping("/")
public String handle(@RequestBody Mono<MultiValueMap<String, Part>> parts) { ①
    // ...
}
```

① Using `@RequestBody`.

Kotlin

```
@PostMapping("/")
fun handle(@RequestBody parts: MultiValueMap<String, Part>): String { ①
    // ...
}
```

① Using `@RequestBody`.

To access multipart data sequentially, in streaming fashion, you can use `@RequestBody` with `Flux<Part>` (or `Flow<Part>` in Kotlin) instead, as the following example shows:

Java

```
@PostMapping("/")
public String handle(@RequestBody Flux<Part> parts) { ①
    // ...
}
```

① Using `@RequestBody`.

Kotlin

```
@PostMapping("/")
fun handle(@RequestBody parts: Flow<Part>): String { ①
    // ...
}
```

① Using `@RequestBody`.

`@RequestBody`

Same as in Spring MVC

You can use the `@RequestBody` annotation to have the request body read and deserialized into an `Object` through an HttpMessageReader. The following example uses a `@RequestBody` argument:

Java

```
@PostMapping("/accounts")
public void handle(@RequestBody Account account) {
    // ...
}
```

Kotlin

```
@PostMapping("/accounts")
fun handle(@RequestBody account: Account) {
    // ...
}
```

Unlike Spring MVC, in WebFlux, the `@RequestBody` method argument supports reactive types and fully non-blocking reading and (client-to-server) streaming.

Java

```
@PostMapping("/accounts")
public void handle(@RequestBody Mono<Account> account) {
    // ...
}
```

Kotlin

```
@PostMapping("/accounts")
fun handle(@RequestBody accounts: Flow<Account>) {
    // ...
}
```

You can use the HTTP message codecs option of the WebFlux Config to configure or customize message readers.

You can use `@RequestBody` in combination with `javax.validation.Valid` or Spring's `@Validated` annotation, which causes Standard Bean Validation to be applied. By default, validation errors cause a `WebExchangeBindException`, which is turned into a 400 (`BAD_REQUEST`) response. Alternatively, you can handle validation errors locally within the controller through an `Errors` or a `BindingResult` argument. The following example uses a `BindingResult` argument`:

Java

```
@PostMapping("/accounts")
public void handle(@Valid @RequestBody Account account, BindingResult result) {
    // ...
}
```

Kotlin

```
@PostMapping("/accounts")
fun handle(@Valid @RequestBody account: Account, result: BindingResult) {
    // ...
}
```

`HttpEntity`

Same as in Spring MVC

`HttpEntity` is more or less identical to using `@RequestBody` but is based on a container object that exposes request headers and the body. The following example uses an `HttpEntity`:

Java

```
@PostMapping("/accounts")
public void handle(HttpEntity<Account> entity) {
    // ...
}
```

Kotlin

```
@PostMapping("/accounts")
fun handle(entity: HttpEntity<Account>) {
    // ...
}
```

`@ResponseBody`

Same as in Spring MVC

You can use the `@ResponseBody` annotation on a method to have the return serialized to the response body through an HttpMessageWriter. The following example shows how to do so:

Java

```
@GetMapping("/accounts/{id}")
@ResponseBody
public Account handle() {
    // ...
}
```

Kotlin

```
@GetMapping("/accounts/{id}")
@ResponseBody
fun handle(): Account {
    // ...
}
```

`@ResponseBody` is also supported at the class level, in which case it is inherited by all controller methods. This is the effect of `@RestController`, which is nothing more than a meta-annotation marked with `@Controller` and `@ResponseBody`.

`@ResponseBody` supports reactive types, which means you can return Reactor or RxJava types and have the asynchronous values they produce rendered to the response. For additional details, see Streaming and JSON rendering.

You can combine `@ResponseBody` methods with JSON serialization views. See Jackson JSON for details.

You can use the HTTP message codecs option of the WebFlux Config to configure or customize message writing.

`ResponseEntity`

Same as in Spring MVC

`ResponseEntity` is like `@ResponseBody` but with status and headers. For example:

Java

```
@GetMapping("/something")
public ResponseEntity<String> handle() {
    String body = ... ;
    String etag = ... ;
    return ResponseEntity.ok().eTag(etag).build(body);
}
```

Kotlin

```
@GetMapping("/something")
fun handle(): ResponseEntity<String> {
    val body: String = ...
    val etag: String = ...
    return ResponseEntity.ok().eTag(etag).build(body)
}
```

WebFlux supports using a single value reactive type to produce the `ResponseEntity` asynchronously, and/or single and multi-value reactive types for the body.

Jackson JSON

Spring offers support for the Jackson JSON library.

JSON Views

Same as in Spring MVC

Spring WebFlux provides built-in support for Jackson's Serialization Views, which allows rendering only a subset of all fields in an `Object`. To use it with `@ResponseBody` or `ResponseEntity` controller methods, you can use Jackson's `@JsonView` annotation to activate a serialization view class, as the following example shows:

Java

```
@RestController
public class UserController {

    @GetMapping("/user")
    @JsonView(User.WithoutPasswordView.class)
    public User getUser() {
        return new User("eric", "7!jd#h23");
    }
}

public class User {

    public interface WithoutPasswordView {};
    public interface WithPasswordView extends WithoutPasswordView {};

    private String username;
    private String password;

    public User() {
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    @JsonView(WithoutPasswordView.class)
    public String getUsername() {
        return this.username;
    }

    @JsonView(WithPasswordView.class)
    public String getPassword() {
        return this.password;
    }
}
```

Kotlin

```
@RestController
class UserController {

    @GetMapping("/user")
    @JsonView(User.WithoutPasswordView::class)
    fun getUser(): User {
        return User("eric", "7!jd#h23")
    }
}

class User(
        @JsonView(WithoutPasswordView::class) val username: String,
        @JsonView(WithPasswordView::class) val password: String
) {
    interface WithoutPasswordView
    interface WithPasswordView : WithoutPasswordView
}
```

`@JsonView` allows an array of view classes but you can only specify only one per controller method. Use a composite interface if you need to activate multiple views.

Model

Same as in Spring MVC

You can use the `@ModelAttribute` annotation:

- On a method argument in `@RequestMapping` methods to create or access an Object from the model and to bind it to the request through a `WebDataBinder`.
- As a method-level annotation in `@Controller` or `@ControllerAdvice` classes, helping to initialize the model prior to any `@RequestMapping` method invocation.
- On a `@RequestMapping` method to mark its return value as a model attribute.

This section discusses `@ModelAttribute` methods, or the second item from the preceding list. A controller can have any number of `@ModelAttribute` methods. All such methods are invoked before `@RequestMapping` methods in the same controller. A `@ModelAttribute` method can also be shared across controllers through `@ControllerAdvice`. See the section on Controller Advice for more details.

`@ModelAttribute` methods have flexible method signatures. They support many of the same arguments as `@RequestMapping` methods (except for `@ModelAttribute` itself and anything related to the request body).

The following example uses a `@ModelAttribute` method:

Java

```
@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
    model.addAttribute(accountRepository.findAccount(number));
    // add more ...
}
```

Kotlin

```
@ModelAttribute
fun populateModel(@RequestParam number: String, model: Model) {
    model.addAttribute(accountRepository.findAccount(number))
    // add more ...
}
```

The following example adds one attribute only:

Java

```
@ModelAttribute
public Account addAccount(@RequestParam String number) {
    return accountRepository.findAccount(number);
}
```

Kotlin

```
@ModelAttribute
fun addAccount(@RequestParam number: String): Account {
    return accountRepository.findAccount(number);
}
```

When a name is not explicitly specified, a default name is chosen based on the type, as explained in the javadoc for `Conventions`. You can always assign an explicit name by using the overloaded `addAttribute` method or through the name attribute on `@ModelAttribute` (for a return value).

Spring WebFlux, unlike Spring MVC, explicitly supports reactive types in the model (for example, `Mono<Account>` or `io.reactivex.Single<Account>`). Such asynchronous model attributes can be transparently resolved (and the model updated) to their actual values at the time of `@RequestMapping` invocation, provided a `@ModelAttribute` argument is declared without a wrapper, as the following example shows:

Java

```
@ModelAttribute
public void addAccount(@RequestParam String number) {
    Mono<Account> accountMono = accountRepository.findAccount(number);
    model.addAttribute("account", accountMono);
}

@PostMapping("/accounts")
public String handle(@ModelAttribute Account account, BindingResult errors) {
    // ...
}
```

Kotlin

```
import org.springframework.ui.set

@ModelAttribute
fun addAccount(@RequestParam number: String) {
    val accountMono: Mono<Account> = accountRepository.findAccount(number)
    model["account"] = accountMono
}

@PostMapping("/accounts")
fun handle(@ModelAttribute account: Account, errors: BindingResult): String {
    // ...
}
```

In addition, any model attributes that have a reactive type wrapper are resolved to their actual values (and the model updated) just prior to view rendering.

You can also use `@ModelAttribute` as a method-level annotation on `@RequestMapping` methods, in which case the return value of the `@RequestMapping` method is interpreted as a model attribute. This is typically not required, as it is the default behavior in HTML controllers, unless the return value is a `String` that would otherwise be interpreted as a view name. `@ModelAttribute` can also help to customize the model attribute name, as the following example shows:

Java

```
@GetMapping("/accounts/{id}")
@ModelAttribute("myAccount")
public Account handle() {
    // ...
    return account;
}
```

Kotlin

```
@GetMapping("/accounts/{id}")
@ModelAttribute("myAccount")
fun handle(): Account {
    // ...
    return account
}
```

DataBinder

Same as in Spring MVC

`@Controller` or `@ControllerAdvice` classes can have `@InitBinder` methods, to initialize instances of `WebDataBinder`. Those, in turn, are used to:

- Bind request parameters (that is, form data or query) to a model object.
- Convert `String`-based request values (such as request parameters, path variables, headers, cookies, and others) to the target type of controller method arguments.
- Format model object values as `String` values when rendering HTML forms.

`@InitBinder` methods can register controller-specific `java.bean.PropertyEditor` or Spring `Converter` and `Formatter` components. In addition, you can use the WebFlux Java configuration to register `Converter` and `Formatter` types in a globally shared `FormattingConversionService`.

`@InitBinder` methods support many of the same arguments that `@RequestMapping` methods do, except for `@ModelAttribute` (command object) arguments. Typically, they are declared with a `WebDataBinder` argument, for registrations, and a `void` return value. The following example uses the `@InitBinder` annotation:

Java

```
@Controller
public class FormController {

    @InitBinder ①
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat,
false));
    }

    // ...
}
```

① Using the `@InitBinder` annotation.

Kotlin

```
@Controller
class FormController {

    @InitBinder ①
    fun initBinder(binder: WebDataBinder) {
        val dateFormat = SimpleDateFormat("yyyy-MM-dd")
        dateFormat.isLenient = false
        binder.registerCustomEditor(Date::class.java, CustomDateEditor(dateFormat,
false))
    }

    // ...
}
```

Alternatively, when using a `Formatter`-based setup through a shared `FormattingConversionService`, you could re-use the same approach and register controller-specific `Formatter` instances, as the following example shows:

Java

```
@Controller
public class FormController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd")); ①
    }

    // ...
}
```

① Adding a custom formatter (a `DateFormatter`, in this case).

Kotlin

```
@Controller
class FormController {

    @InitBinder
    fun initBinder(binder: WebDataBinder) {
        binder.addCustomFormatter(DateFormatter("yyyy-MM-dd")) ①
    }

    // ...
}
```

① Adding a custom formatter (a `DateFormatter`, in this case).

Managing Exceptions

Same as in Spring MVC

`@Controller` and `@ControllerAdvice` classes can have `@ExceptionHandler` methods to handle exceptions from controller methods. The following example includes such a handler method:

Java

```
@Controller
public class SimpleController {

    // ...

    @ExceptionHandler ①
    public ResponseEntity<String> handle(IOException ex) {
        // ...
    }
}
```

① Declaring an `@ExceptionHandler`.

Kotlin

```
@Controller
class SimpleController {

    // ...

    @ExceptionHandler ①
    fun handle(ex: IOException): ResponseEntity<String> {
        // ...
    }
}
```

① Declaring an `@ExceptionHandler`.

The exception can match against a top-level exception being propagated (that is, a direct `IOException` being thrown) or against the immediate cause within a top-level wrapper exception (for example, an `IOException` wrapped inside an `IllegalStateException`).

For matching exception types, preferably declare the target exception as a method argument, as shown in the preceding example. Alternatively, the annotation declaration can narrow the exception types to match. We generally recommend being as specific as possible in the argument signature and to declare your primary root exception mappings on a `@ControllerAdvice` prioritized with a corresponding order. See the MVC section for details.

An `@ExceptionHandler` method in WebFlux supports the same method arguments and return values as a `@RequestMapping` method, with the exception of request body- and `@ModelAttribute`-related method arguments.

Support for `@ExceptionHandler` methods in Spring WebFlux is provided by the `HandlerAdapter` for `@RequestMapping` methods. See `DispatcherHandler` for more detail.

REST API exceptions

Same as in Spring MVC

A common requirement for REST services is to include error details in the body of the response. The Spring Framework does not automatically do so, because the representation of error details in the response body is application-specific. However, a `@RestController` can use `@ExceptionHandler` methods with a `ResponseEntity` return value to set the status and the body of the response. Such methods can also be declared in `@ControllerAdvice` classes to apply them globally.

Note that Spring WebFlux does not have an equivalent for the Spring MVC `ResponseEntityExceptionHandler`, because WebFlux raises only `ResponseStatusException` (or subclasses thereof), and those do not need to be translated to an HTTP status code.

Controller Advice

Same as in Spring MVC

Typically, the `@ExceptionHandler`, `@InitBinder`, and `@ModelAttribute` methods apply within the `@Controller` class (or class hierarchy) in which they are declared. If you want such methods to apply more globally (across controllers), you can declare them in a class annotated with `@ControllerAdvice` or `@RestControllerAdvice`.

`@ControllerAdvice` is annotated with `@Component`, which means that such classes can be registered as Spring beans through component scanning. `@RestControllerAdvice` is a composed annotation that is annotated with both `@ControllerAdvice` and `@ResponseBody`, which essentially means `@ExceptionHandler` methods are rendered to the response body through message conversion (versus view resolution or template rendering).

On startup, the infrastructure classes for `@RequestMapping` and `@ExceptionHandler` methods detect Spring beans annotated with `@ControllerAdvice` and then apply their methods at runtime. Global `@ExceptionHandler` methods (from a `@ControllerAdvice`) are applied *after* local ones (from the `@Controller`). By contrast, global `@ModelAttribute` and `@InitBinder` methods are applied *before* local ones.

By default, `@ControllerAdvice` methods apply to every request (that is, all controllers), but you can narrow that down to a subset of controllers by using attributes on the annotation, as the following example shows:

Java

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class ExampleAdvice1 {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = {ControllerInterface.class, AbstractController
.class})
public class ExampleAdvice3 {}
```

Kotlin

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = [RestController::class])
public class ExampleAdvice1 {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = [ControllerInterface::class,
AbstractController::class])
public class ExampleAdvice3 {}
```

The selectors in the preceding example are evaluated at runtime and may negatively impact performance if used extensively. See the `@ControllerAdvice` javadoc for more details.

Functional Endpoints

Same as in Spring MVC

Spring WebFlux includes WebFlux.fn, a lightweight functional programming model in which functions are used to route and handle requests and contracts are designed for immutability. It is an alternative to the annotation-based programming model but otherwise runs on the same Reactive Core foundation.

Overview

Same as in Spring MVC

In WebFlux.fn, an HTTP request is handled with a `HandlerFunction`: a function that takes `ServerRequest` and returns a delayed `ServerResponse` (i.e. `Mono<ServerResponse>`). Both the request as the response object have immutable contracts that offer JDK 8-friendly access to the HTTP request

and response. `HandlerFunction` is the equivalent of the body of a `@RequestMapping` method in the annotation-based programming model.

Incoming requests are routed to a handler function with a `RouterFunction`: a function that takes `ServerRequest` and returns a delayed `HandlerFunction` (i.e. `Mono<HandlerFunction>`). When the router function matches, a handler function is returned; otherwise an empty Mono. `RouterFunction` is the equivalent of a `@RequestMapping` annotation, but with the major difference that router functions provide not just data, but also behavior.

`RouterFunctions.route()` provides a router builder that facilitates the creation of routers, as the following example shows:

Java

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

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

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


public class PersonHandler {

    // ...

    public Mono<ServerResponse> listPeople(ServerRequest request) {
        // ...
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) {
        // ...
    }

    public Mono<ServerResponse> getPerson(ServerRequest request) {
        // ...
    }
}
```

Kotlin

```
val repository: PersonRepository = ...
val handler = PersonHandler(repository)

val route = coRouter { ①
    accept(APPLICATION_JSON).nest {
        GET("/person/{id}", handler::getPerson)
        GET("/person", handler::listPeople)
    }
    POST("/person", handler::createPerson)
}


class PersonHandler(private val repository: PersonRepository) {

    // ...

    suspend fun listPeople(request: ServerRequest): ServerResponse {
        // ...
    }

    suspend fun createPerson(request: ServerRequest): ServerResponse {
        // ...
    }

    suspend fun getPerson(request: ServerRequest): ServerResponse {
        // ...
    }
}
```

① Create router using Coroutines router DSL, a Reactive alternative is also available via `router { }`.

One way to run a `RouterFunction` is to turn it into an `HttpHandler` and install it through one of the built-in server adapters:

- `RouterFunctions.toHttpHandler(RouterFunction)`
- `RouterFunctions.toHttpHandler(RouterFunction, HandlerStrategies)`

Most applications can run through the WebFlux Java configuration, see Running a Server.

HandlerFunction

Same as in Spring MVC

`ServerRequest` and `ServerResponse` are immutable interfaces that offer JDK 8-friendly access to the HTTP request and response. Both request and response provide Reactive Streams back pressure against the body streams. The request body is represented with a Reactor `Flux` or `Mono`. The response body is represented with any Reactive Streams `Publisher`, including `Flux` and `Mono`. For more on that, see Reactive Libraries.

`ServerRequest`

`ServerRequest` provides access to the HTTP method, URI, headers, and query parameters, while access to the body is provided through the `body` methods.

The following example extracts the request body to a `Mono<String>`:

Java

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

Kotlin

```
val string = request.awaitBody<String>()
```

The following example extracts the body to a `Flux<Person>` (or a `Flow<Person>` in Kotlin), where `Person` objects are decoded from someserialized form, such as JSON or XML:

Java

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

Kotlin

```
val people = request.bodyToFlow<Person>()
```

The preceding examples are shortcuts that use the more general `ServerRequest.body(BodyExtractor)`, which accepts the `BodyExtractor` functional strategy interface. The utility class `BodyExtractors` provides access to a number of instances. For example, the preceding examples can also be written as follows:

Java

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

Kotlin

```
    val string = request.body(BodyExtractors.toMono(String::class.java)).awaitFirst()
    val people = request.body(BodyExtractors.toFlux(Person::class.java)).asFlow()
```

The following example shows how to access form data:

Java

```
Mono<MultiValueMap<String, String> map = request.formData();
```

Kotlin

```
val map = request.awaitFormData()
```

The following example shows how to access multipart data as a map:

Java

```
Mono<MultiValueMap<String, Part> map = request.multipartData();
```

Kotlin

```
val map = request.awaitMultipartData()
```

The following example shows how to access multiparts, one at a time, in streaming fashion:

Java

```
Flux<Part> parts = request.body(BodyExtractors.toParts());
```

Kotlin

```
val parts = request.body(BodyExtractors.toParts()).asFlow()
```

`ServerResponse`

`ServerResponse` provides access to the HTTP response and, since it is immutable, you can use a `build` method to create it. You can use the builder to set the response status, to add response headers, or to provide a body. The following example creates a 200 (OK) response with JSON content:

Java

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

Kotlin

```
val person: Person = ...
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).bodyValue(person)
```

The following example shows how to build a 201 (CREATED) response with a `Location` header and no body:

Java

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

Kotlin

```
val location: URI = ...
ServerResponse.created(location).build()
```

Depending on the codec used, it is possible to pass hint parameters to customize how the body is serialized or deserialized. For example, to specify a Jackson JSON view:

Java

```
ServerResponse.ok().hint(Jackson2CodecSupport.JSON_VIEW_HINT, MyJacksonView.class
).body(...);
```

Kotlin

```
ServerResponse.ok().hint(Jackson2CodecSupport.JSON_VIEW_HINT,
MyJacksonView::class.java).body(...)
```

Handler Classes

We can write a handler function as a lambda, as the following example shows:

Java

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

Kotlin

```
val helloWorld = HandlerFunction<ServerResponse> {
ServerResponse.ok().bodyValue("Hello World") }
```

That is convenient, but in an application we need multiple functions, and multiple inline lambda's can get messy. Therefore, it is useful to group related handler functions together into a handler class, which has a similar role as `@Controller` in an annotation-based application. For example, the following class exposes a reactive `Person` repository:

Java

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.ServerResponse.ok;

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 ok().contentType(APPLICATION_JSON).body(people, Person.class);
    }

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

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

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 (that is, when the `Person` has been saved).

③ `getPerson` is a handler function that returns a single person, identified by the `id` path variable. We retrieve that `Person` from 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.

Kotlin

```
class PersonHandler(private val repository: PersonRepository) {

    suspend fun listPeople(request: ServerRequest): ServerResponse { ①
        val people: Flow<Person> = repository.allPeople()
        return ok().contentType(APPLICATION_JSON).bodyAndAwait(people);
    }

    suspend fun createPerson(request: ServerRequest): ServerResponse { ②
        val person = request.awaitBody<Person>()
        repository.savePerson(person)
        return ok().buildAndAwait()
    }

    suspend fun getPerson(request: ServerRequest): ServerResponse { ③
        val personId = request.pathVariable("id").toInt()
        return repository.getPerson(personId)?.let {
ok().contentType(APPLICATION_JSON).bodyValueAndAwait(it) }
                ?: ServerResponse.notFound().buildAndAwait()

    }
}
```

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)` is a suspending function with no return type.

③ `getPerson` is a handler function that returns a single person, identified by the `id` path variable. We retrieve that `Person` from the repository and create a JSON response, if it is found. If it is not found, we return a 404 Not Found response.

Validation

A functional endpoint can use Spring's validation facilities to apply validation to the request body. For example, given a custom Spring Validator implementation for a `Person`:

Java

```
public class PersonHandler {

    private final Validator validator = new PersonValidator(); ①

    // ...

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

    private void validate(Person person) {
        Errors errors = new BeanPropertyBindingResult(person, "person");
        validator.validate(person, errors);
        if (errors.hasErrors()) {
            throw new ServerWebInputException(errors.toString()); ③
        }
    }
}
```

① Create `Validator` instance.

② Apply validation.

③ Raise exception for a 400 response.

Kotlin

```
class PersonHandler(private val repository: PersonRepository) {

    private val validator = PersonValidator() ①

    // ...

    suspend fun createPerson(request: ServerRequest): ServerResponse {
        val person = request.awaitBody<Person>()
        validate(person) ②
        repository.savePerson(person)
        return ok().buildAndAwait()
    }

    private fun validate(person: Person) {
        val errors: Errors = BeanPropertyBindingResult(person, "person");
        validator.validate(person, errors);
        if (errors.hasErrors()) {
            throw ServerWebInputException(errors.toString()) ③
        }
    }
}
```

① Create `Validator` instance.

② Apply validation.

③ Raise exception for a 400 response.

Handlers can also use the standard bean validation API (JSR-303) by creating and injecting a global `Validator` instance based on `LocalValidatorFactoryBean`. See Spring Validation.

`RouterFunction`

Same as in Spring MVC

Router functions are used to route the requests to the corresponding `HandlerFunction`. Typically, you do not write router functions yourself, but rather use a method on the `RouterFunctions` utility class to create one. `RouterFunctions.route()` (no parameters) provides you with a fluent builder for creating a router function, whereas `RouterFunctions.route(RequestPredicate, HandlerFunction)` offers a direct way to create a router.

Generally, it is recommended to use the `route()` builder, as it provides convenient short-cuts for typical mapping scenarios without requiring hard-to-discover static imports. For instance, the router function builder offers the method `GET(String, HandlerFunction)` to create a mapping for GET requests; and `POST(String, HandlerFunction)` for POSTs.

Besides HTTP method-based mapping, the route builder offers a way to introduce additional predicates when mapping to requests. For each HTTP method there is an overloaded variant that takes a `RequestPredicate` as a parameter, though which additional constraints can be expressed.

Predicates

You can write your own `RequestPredicate`, but the `RequestPredicates` utility class offers commonly used implementations, based on the request path, HTTP method, content-type, and so on. The following example uses a request predicate to create a constraint based on the `Accept` header:

Java

```
RouterFunction<ServerResponse> route = RouterFunctions.route()
    .GET("/hello-world", accept(MediaType.TEXT_PLAIN),
        request -> ServerResponse.ok().bodyValue("Hello World"));
```

Kotlin

```
val route = coRouter {
    GET("/hello-world", accept(TEXT_PLAIN)) {
        ServerResponse.ok().bodyValueAndAwait("Hello World")
    }
}
```

You can compose multiple request predicates together by using:

- `RequestPredicate.and(RequestPredicate)` — both must match.
- `RequestPredicate.or(RequestPredicate)` — either can match.

Many of the predicates from `RequestPredicates` are composed. For example, `RequestPredicates.GET(String)` is composed from `RequestPredicates.method(HttpMethod)` and `RequestPredicates.path(String)`. The example shown above also uses two request predicates, as the builder uses `RequestPredicates.GET` internally, and composes that with the `accept` predicate.

Routes

Router functions are evaluated in order: if the first route does not match, the second is evaluated, and so on. Therefore, it makes sense to declare more specific routes before general ones. Note that this behavior is different from the annotation-based programming model, where the "most specific" controller method is picked automatically.

When using the router function builder, all defined routes are composed into one `RouterFunction` that is returned from `build()`. There are also other ways to compose multiple router functions together:

- `add(RouterFunction)` on the `RouterFunctions.route()` builder
- `RouterFunction.and(RouterFunction)`
- `RouterFunction.andRoute(RequestPredicate, HandlerFunction)` — shortcut for `RouterFunction.and()` with nested `RouterFunctions.route()`.

The following example shows the composition of four routes:

Java

```
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> otherRoute = ...

RouterFunction<ServerResponse> route = route()
    .GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson) ①
    .GET("/person", accept(APPLICATION_JSON), handler::listPeople) ②
    .POST("/person", handler::createPerson) ③
    .add(otherRoute) ④
    .build();
```

① `GET /person/{id}` with an `Accept` header that matches JSON is routed to `PersonHandler.getPerson`

② `GET /person` with an `Accept` header that matches JSON is routed to `PersonHandler.listPeople`

③ `POST /person` with no additional predicates is mapped to `PersonHandler.createPerson`, and

④ `otherRoute` is a router function that is created elsewhere, and added to the route built.

Kotlin

```
import org.springframework.http.MediaType.APPLICATION_JSON

val repository: PersonRepository = ...
val handler = PersonHandler(repository);

val otherRoute: RouterFunction<ServerResponse> = coRouter {  }

val route = coRouter {
    GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson) ①
    GET("/person", accept(APPLICATION_JSON), handler::listPeople) ②
    POST("/person", handler::createPerson) ③
}.and(otherRoute) ④
```

① `GET /person/{id}` with an `Accept` header that matches JSON is routed to `PersonHandler.getPerson`

② `GET /person` with an `Accept` header that matches JSON is routed to `PersonHandler.listPeople`

③ `POST /person` with no additional predicates is mapped to `PersonHandler.createPerson`, and

④ `otherRoute` is a router function that is created elsewhere, and added to the route built.

Nested Routes

It is common for a group of router functions to have a shared predicate, for instance a shared path. In the example above, the shared predicate would be a path predicate that matches `/person`, used by three of the routes. When using annotations, you would remove this duplication by using a type-level `@RequestMapping` annotation that maps to `/person`. In WebFlux.fn, path predicates can be

shared through the `path` method on the router function builder. For instance, the last few lines of the example above can be improved in the following way by using nested routes:

Java

```
RouterFunction<ServerResponse> route = route()
    .path("/person", builder -> builder ①
        .GET("/{id}", accept(APPLICATION_JSON), handler::getPerson)
        .GET("", accept(APPLICATION_JSON), handler::listPeople)
        .POST("/person", handler::createPerson))
    .build();
```

① Note that second parameter of `path` is a consumer that takes the a router builder.

Kotlin

```
val route = coRouter {
    "/person".nest {
        GET("/{id}", accept(APPLICATION_JSON), handler::getPerson)
        GET("", accept(APPLICATION_JSON), handler::listPeople)
        POST("/person", handler::createPerson)
    }
}
```

Though path-based nesting is the most common, you can nest on any kind of predicate by using the `nest` method on the builder. The above still contains some duplication in the form of the shared `Accept`-header predicate. We can further improve by using the `nest` method together with `accept`:

Java

```
RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET("", handler::listPeople))
        .POST("/person", handler::createPerson))
    .build();
```

Kotlin

```
val route = coRouter {
    "/person".nest {
        accept(APPLICATION_JSON).nest {
            GET("/{id}", handler::getPerson)
            GET("", handler::listPeople)
            POST("/person", handler::createPerson)
        }
    }
}
```

Running a Server

Same as in Spring MVC

How do you run a router function in an HTTP server? A simple option is to convert a router function to an `HttpHandler` by using one of the following:

- `RouterFunctions.toHttpHandler(RouterFunction)`
- `RouterFunctions.toHttpHandler(RouterFunction, HandlerStrategies)`

You can then use the returned `HttpHandler` with a number of server adapters by following HttpHandler for server-specific instructions.

A more typical option, also used by Spring Boot, is to run with a `DispatcherHandler`-based setup through the WebFlux Config, which uses Spring configuration to declare the components required to process requests. The WebFlux Java configuration declares the following infrastructure components to support functional endpoints:

- `RouterFunctionMapping`: Detects one or more `RouterFunction<?>` beans in the Spring configuration, combines them through `RouterFunction.andOther`, and routes requests to the resulting composed `RouterFunction`.
- `HandlerFunctionAdapter`: Simple adapter that lets `DispatcherHandler` invoke a `HandlerFunction` that was mapped to a request.
- `ServerResponseResultHandler`: Handles the result from the invocation of a `HandlerFunction` by invoking the `writeTo` method of the `ServerResponse`.

The preceding components let functional endpoints fit within the `DispatcherHandler` request processing lifecycle and also (potentially) run side by side with annotated controllers, if any are declared. It is also how functional endpoints are enabled by the Spring Boot WebFlux starter.

The following example shows a WebFlux Java configuration (see DispatcherHandler for how to run it):

Java

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

    @Bean
    public RouterFunction<?> routerFunctionA() {
        // ...
    }

    @Bean
    public RouterFunction<?> routerFunctionB() {
        // ...
    }

    // ...

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        // configure message conversion...
    }

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        // configure CORS...
    }

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        // configure view resolution for HTML rendering...
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    @Bean
    fun routerFunctionA(): RouterFunction<<strong>> {
        // ...
    }

    @Bean
    fun routerFunctionB(): RouterFunction<</strong>> {
        // ...
    }

    // ...

    override fun configureHttpMessageCodecs(configurer: ServerCodecConfigurer) {
        // configure message conversion...
    }

    override fun addCorsMappings(registry: CorsRegistry) {
        // configure CORS...
    }

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        // configure view resolution for HTML rendering...
    }
}
```

Filtering Handler Functions

Same as in Spring MVC

You can filter handler functions by using the `before`, `after`, or `filter` methods on the routing function builder. With annotations, you can achieve similar functionality by using `@ControllerAdvice`, a `ServletFilter`, or both. The filter will apply to all routes that are built by the builder. This means that filters defined in nested routes do not apply to "top-level" routes. For instance, consider the following example:

Java

```
RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET("", handler::listPeople)
            .before(request -> ServerRequest.from(request) ①
                .header("X-RequestHeader", "Value")
                .build()))
        .POST("/person", handler::createPerson))
    .after((request, response) -> logResponse(response)) ②
    .build();
```

① The `before` filter that adds a custom request header is only applied to the two GET routes.

② The `after` filter that logs the response is applied to all routes, including the nested ones.

Kotlin

```
val route = router {
    "/person".nest {
        GET("/{id}", handler::getPerson)
        GET("", handler::listPeople)
        before { ①
            ServerRequest.from(it)
                    .header("X-RequestHeader", "Value").build()
        }
        POST("/person", handler::createPerson)
        after { _, response -> ②
            logResponse(response)
        }
    }
}
```

① The `before` filter that adds a custom request header is only applied to the two GET routes.

② The `after` filter that logs the response is applied to all routes, including the nested ones.

The `filter` method on the router builder takes a `HandlerFilterFunction`: 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 handler that is routed to, but it can also be another filter if multiple are applied.

Now we can add a simple security filter to our route, assuming that we have a `SecurityManager` that can determine whether a particular path is allowed. The following example shows how to do so:

Java

```
SecurityManager securityManager = ...

RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET("", handler::listPeople))
        .POST("/person", handler::createPerson))
    .filter((request, next) -> {
        if (securityManager.allowAccessTo(request.path())) {
            return next.handle(request);
        }
        else {
            return ServerResponse.status(UNAUTHORIZED).build();
        }
    })
    .build();
```

Kotlin

```
val securityManager: SecurityManager = ...

val route = router {
        ("/person" and accept(APPLICATION_JSON)).nest {
            GET("/{id}", handler::getPerson)
            GET("", handler::listPeople)
            POST("/person", handler::createPerson)
            filter { request, next ->
                if (securityManager.allowAccessTo(request.path())) {
                    next(request)
                }
                else {
                    status(UNAUTHORIZED).build();
                }
            }
        }
    }
```

The preceding example demonstrates that invoking the `next.handle(ServerRequest)` is optional. We allow only the handler function to be executed when access is allowed.

Besides using the `filter` method on the router function builder, it is possible to apply a filter to an existing router function via `RouterFunction.filter(HandlerFilterFunction)`.

CORS support for functional endpoints is provided through a dedicated `CorsWebFilter`.

URI Links

Same as in Spring MVC

This section describes various options available in the Spring Framework to prepare URIs.

UriComponents

Spring MVC and Spring WebFlux

`UriComponentsBuilder` helps to build URI's from URI templates with variables, as the following example shows:

Java

```
UriComponents uriComponents = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")  ①
        .queryParam("q", "{q}")  ②
        .encode() ③
        .build(); ④

URI uri = uriComponents.expand("Westin", "123").toUri();  ⑤
```

① Static factory method with a URI template.

② Add or replace URI components.

③ Request to have the URI template and URI variables encoded.

④ Build a `UriComponents`.

⑤ Expand variables and obtain the `URI`.

Kotlin

```
val uriComponents = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")  ①
        .queryParam("q", "{q}")  ②
        .encode() ③
        .build() ④

val uri = uriComponents.expand("Westin", "123").toUri()  ⑤
```

① Static factory method with a URI template.

② Add or replace URI components.

③ Request to have the URI template and URI variables encoded.

④ Build a `UriComponents`.

⑤ Expand variables and obtain the `URI`.

The preceding example can be consolidated into one chain and shortened with `buildAndExpand`, as the following example shows:

Java

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("Westin", "123")
        .toUri();
```

Kotlin

```
val uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("Westin", "123")
        .toUri()
```

You can shorten it further by going directly to a URI (which implies encoding), as the following example shows:

Java

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

Kotlin

```
val uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123")
```

You shorter it further still with a full URI template, as the following example shows:

Java

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}?q={q}")
        .build("Westin", "123");
```

Kotlin

```
val uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}?q={q}")
        .build("Westin", "123")
```

UriBuilder

Spring MVC and Spring WebFlux

`UriComponentsBuilder` implements `UriBuilder`. You can create a `UriBuilder`, in turn, with a `UriBuilderFactory`. Together, `UriBuilderFactory` and `UriBuilder` provide a pluggable mechanism to build URIs from URI templates, based on shared configuration, such as a base URL, encoding preferences, and other details.

You can configure `RestTemplate` and `WebClient` with a `UriBuilderFactory` to customize the preparation of URIs. `DefaultUriBuilderFactory` is a default implementation of `UriBuilderFactory` that uses `UriComponentsBuilder` internally and exposes shared configuration options.

The following example shows how to configure a `RestTemplate`:

Java

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);
```

Kotlin

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode

val baseUrl = "https://example.org"
val factory = DefaultUriBuilderFactory(baseUrl)
factory.encodingMode = EncodingMode.TEMPLATE_AND_VALUES

val restTemplate = RestTemplate()
restTemplate.uriTemplateHandler = factory
```

The following example configures a `WebClient`:

Java

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

Kotlin

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode

val baseUrl = "https://example.org"
val factory = DefaultUriBuilderFactory(baseUrl)
factory.encodingMode = EncodingMode.TEMPLATE_AND_VALUES

val client = WebClient.builder().uriBuilderFactory(factory).build()
```

In addition, you can also use `DefaultUriBuilderFactory` directly. It is similar to using `UriComponentsBuilder` but, instead of static factory methods, it is an actual instance that holds configuration and preferences, as the following example shows:

Java

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory uriBuilderFactory = new DefaultUriBuilderFactory(baseUrl);

URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

Kotlin

```
val baseUrl = "https://example.com"
val uriBuilderFactory = DefaultUriBuilderFactory(baseUrl)

val uri = uriBuilderFactory.uriString("/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123")
```

URI Encoding

Spring MVC and Spring WebFlux

`UriComponentsBuilder` exposes encoding options at two levels:

- UriComponentsBuilder#encode(): Pre-encodes the URI template first and then strictly encodes URI variables when expanded.
- UriComponents#encode(): Encodes URI components *after* URI variables are expanded.

Both options replace non-ASCII and illegal characters with escaped octets. However, the first option also replaces characters with reserved meaning that appear in URI variables.

Consider ";", which is legal in a path but has reserved meaning. The first option replaces ";" with "%3B" in URI variables but not in the URI template. By contrast, the second option never replaces ";", since it is a legal character in a path.

For most cases, the first option is likely to give the expected result, because it treats URI variables as opaque data to be fully encoded, while option 2 is useful only if URI variables intentionally contain reserved characters.

The following example uses the first option:

Java

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("New York", "foo+bar")
        .toUri();

// Result is "/hotel%20list/New%20York?q=foo%2Bbar"
```

Kotlin

```
val uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("New York", "foo+bar")
        .toUri()

// Result is "/hotel%20list/New%20York?q=foo%2Bbar"
```

You can shorten the preceding example by going directly to the URI (which implies encoding), as the following example shows:

Java

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .build("New York", "foo+bar")
```

Kotlin

```
val uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .build("New York", "foo+bar")
```

You can shorten it further still with a full URI template, as the following example shows:

Java

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}?q={q}")
        .build("New York", "foo+bar")
```

Kotlin

```
val uri = UriComponentsBuilder.fromPath("/hotel list/{city}?q={q}")
        .build("New York", "foo+bar")
```

The `WebClient` and the `RestTemplate` expand and encode URI templates internally through the `UriBuilderFactory` strategy. Both can be configured with a custom strategy. as the following example shows:

Java

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl)
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

// Customize the RestTemplate..
RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);

// Customize the WebClient..
WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

Kotlin

```
val baseUrl = "https://example.com"
val factory = DefaultUriBuilderFactory(baseUrl).apply {
    encodingMode = EncodingMode.TEMPLATE_AND_VALUES
}

// Customize the RestTemplate..
val restTemplate = RestTemplate().apply {
    uriTemplateHandler = factory
}

// Customize the WebClient..
val client = WebClient.builder().uriBuilderFactory(factory).build()
```

The `DefaultUriBuilderFactory` implementation uses `UriComponentsBuilder` internally to expand and encode URI templates. As a factory, it provides a single place to configure the approach to encoding, based on one of the below encoding modes:

- `TEMPLATE_AND_VALUES`: Uses `UriComponentsBuilder#encode()`, corresponding to the first option in the earlier list, to pre-encode the URI template and strictly encode URI variables when expanded.
- `VALUES_ONLY`: Does not encode the URI template and, instead, applies strict encoding to URI variables through `UriUtils#encodeUriUriVariables` prior to expanding them into the template.
- `URI_COMPONENTS`: Uses `UriComponents#encode()`, corresponding to the second option in the earlier list, to encode URI component value *after* URI variables are expanded.
- `NONE`: No encoding is applied.

The `RestTemplate` is set to `EncodingMode.URI_COMPONENTS` for historic reasons and for backwards compatibility. The `WebClient` relies on the default value in `DefaultUriBuilderFactory`, which was changed from `EncodingMode.URI_COMPONENTS` in 5.0.x to `EncodingMode.TEMPLATE_AND_VALUES` in 5.1.

CORS

Same as in Spring MVC

Spring WebFlux lets you handle CORS (Cross-Origin Resource Sharing). This section describes how to do so.

Introduction

Same as in Spring MVC

For security reasons, browsers prohibit AJAX calls to resources outside the current origin. For example, you could have your bank account in one tab and evil.com in another. Scripts from evil.com should not be able to make AJAX requests to your bank API with your credentials — for example, withdrawing money from your account!

Cross-Origin Resource Sharing (CORS) is a W3C specification implemented by most browsers that lets you specify what kind of cross-domain requests are authorized, rather than using less secure and less powerful workarounds based on IFRAME or JSONP.

Processing

Same as in Spring MVC

The CORS specification distinguishes between preflight, simple, and actual requests. To learn how CORS works, you can read this article, among many others, or see the specification for more details.

Spring WebFlux `HandlerMapping` implementations provide built-in support for CORS. After successfully mapping a request to a handler, a `HandlerMapping` checks the CORS configuration for the given request and handler and takes further actions. Preflight requests are handled directly, while simple and actual CORS requests are intercepted, validated, and have the required CORS response headers set.

In order to enable cross-origin requests (that is, the `Origin` header is present and differs from the host of the request), you need to have some explicitly declared CORS configuration. If no matching CORS configuration is found, preflight requests are rejected. No CORS headers are added to the responses of simple and actual CORS requests and, consequently, browsers reject them.

Each `HandlerMapping` can be configured individually with URL pattern-based `CorsConfiguration` mappings. In most cases, applications use the WebFlux Java configuration to declare such mappings, which results in a single, global map passed to all `HadlerMappping` implementations.

You can combine global CORS configuration at the `HandlerMapping` level with more fine-grained, handler-level CORS configuration. For example, annotated controllers can use class- or method-level `@CrossOrigin` annotations (other handlers can implement `CorsConfigurationSource`).

The rules for combining global and local configuration are generally additive — for example, all global and all local origins. For those attributes where only a single value can be accepted, such as `allowCredentials` and `maxAge`, the local overrides the global value. See `CorsConfiguration#combine(CorsConfiguration)` for more details.

To learn more from the source or to make advanced customizations, see:

- `CorsConfiguration`
- `CorsProcessor` and `DefaultCorsProcessor`
- `AbstractHandlerMapping`

`@CrossOrigin`

Same as in Spring MVC

The `@CrossOrigin` annotation enables cross-origin requests on annotated controller methods, as the following example shows:

Java

```
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin
    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) {
        // ...
    }
}
```

Kotlin

```
@RestController
@RequestMapping("/account")
class AccountController {

    @CrossOrigin
    @GetMapping("/{id}")
    suspend fun retrieve(@PathVariable id: Long): Account {
        // ...
    }

    @DeleteMapping("/{id}")
    suspend fun remove(@PathVariable id: Long) {
        // ...
    }
}
```

By default, `@CrossOrigin` allows:

- All origins.
- All headers.
- All HTTP methods to which the controller method is mapped.

`allowedCredentials` is not enabled by default, since that establishes a trust level that exposes sensitive user-specific information (such as cookies and CSRF tokens) and should be used only where appropriate.

`maxAge` is set to 30 minutes.

`@CrossOrigin` is supported at the class level, too, and inherited by all methods. The following

example specifies a certain domain and sets `maxAge` to an hour:

Java

```
@CrossOrigin(origins = "https://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) {
        // ...
    }
}
```

Kotlin

```
@CrossOrigin("https://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
class AccountController {

    @GetMapping("/{id}")
    suspend fun retrieve(@PathVariable id: Long): Account {
        // ...
    }

    @DeleteMapping("/{id}")
    suspend fun remove(@PathVariable id: Long) {
        // ...
    }
}
```

You can use `@CrossOrigin` at both the class and the method level, as the following example shows:

Java

```
@CrossOrigin(maxAge = 3600) ①
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin("https://domain2.com") ②
    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) {
        // ...
    }
}
```

① Using `@CrossOrigin` at the class level.

② Using `@CrossOrigin` at the method level.

Kotlin

```
@CrossOrigin(maxAge = 3600) ①
@RestController
@RequestMapping("/account")
class AccountController {

    @CrossOrigin("https://domain2.com") ②
    @GetMapping("/{id}")
    suspend fun retrieve(@PathVariable id: Long): Account {
        // ...
    }

    @DeleteMapping("/{id}")
    suspend fun remove(@PathVariable id: Long) {
        // ...
    }
}
```

① Using `@CrossOrigin` at the class level.

② Using `@CrossOrigin` at the method level.

Global Configuration

Same as in Spring MVC

In addition to fine-grained, controller method-level configuration, you probably want to define some global CORS configuration, too. You can set URL-based `CorsConfiguration` mappings

individually on any `HandlerMapping`. Most applications, however, use the WebFlux Java configuration to do that.

By default global configuration enables the following:

- All origins.
- All headers.
- `GET`, `HEAD`, and `POST` methods.

`allowedCredentials` is not enabled by default, since that establishes a trust level that exposes sensitive user-specific information(such as cookies and CSRF tokens) and should be used only where appropriate.

`maxAge` is set to 30 minutes.

To enable CORS in the WebFlux Java configuration, you can use the `CorsRegistry` callback, as the following example shows:

Java

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

    @Override
    public void addCorsMappings(CorsRegistry registry) {

        registry.addMapping("/api/**")
            .allowedOrigins("https://domain2.com")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("header1", "header2", "header3")
            .exposedHeaders("header1", "header2")
            .allowCredentials(true).maxAge(3600);

        // Add more mappings...
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun addCorsMappings(registry: CorsRegistry) {

        registry.addMapping("/api/**")
                .allowedOrigins("https://domain2.com")
                .allowedMethods("PUT", "DELETE")
                .allowedHeaders("header1", "header2", "header3")
                .exposedHeaders("header1", "header2")
                .allowCredentials(true).maxAge(3600)

        // Add more mappings...
    }
}
```

CORS `WebFilter`

Same as in Spring MVC

You can apply CORS support through the built-in `CorsWebFilter`, which is a good fit with functional endpoints.

If you try to use the `CorsFilter` with Spring Security, keep in mind that Spring Security has built-in support for CORS.

To configure the filter, you can declare a `CorsWebFilter` bean and pass a `CorsConfigurationSource` to its constructor, as the following example shows:

Java

```
@Bean
CorsWebFilter corsFilter() {

    CorsConfiguration config = new CorsConfiguration();

    // Possibly...
    // config.applyPermitDefaultValues()

    config.setAllowCredentials(true);
    config.addAllowedOrigin("https://domain1.com");
    config.addAllowedHeader("*");
    config.addAllowedMethod("*");

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", config);

    return new CorsWebFilter(source);
}
```

Kotlin

```
@Bean
fun corsFilter(): CorsWebFilter {

    val config = CorsConfiguration()

    // Possibly...
    // config.applyPermitDefaultValues()

    config.allowCredentials = true
    config.addAllowedOrigin("https://domain1.com")
    config.addAllowedHeader("*")
    config.addAllowedMethod("*")

    val source = UrlBasedCorsConfigurationSource().apply {
        registerCorsConfiguration("/**", config)
    }
    return CorsWebFilter(source)
}
```

Web Security

Same as in Spring MVC

The Spring Security project provides support for protecting web applications from malicious exploits. See the Spring Security reference documentation, including:

- WebFlux Security
- WebFlux Testing Support
- CSRF Protection
- Security Response Headers

View Technologies

Same as in Spring MVC

The use of view technologies in Spring WebFlux is pluggable. Whether you decide to use Thymeleaf, FreeMarker, or some other view technology is primarily a matter of a configuration change. This chapter covers the view technologies integrated with Spring WebFlux. We assume you are already familiar with View Resolution.

Thymeleaf

Same as in Spring MVC

Thymeleaf is a modern server-side Java template engine that emphasizes natural HTML templates that can be previewed in a browser by double-clicking, which is very helpful for independent work on UI templates (for example, by a designer) without the need for a running server. Thymeleaf offers an extensive set of features, and it is actively developed and maintained. For a more complete introduction, see the Thymeleaf project home page.

The Thymeleaf integration with Spring WebFlux is managed by the Thymeleaf project. The configuration involves a few bean declarations, such as `SpringResourceTemplateResolver`, `SpringWebFluxTemplateEngine`, and `ThymeleafReactiveViewResolver`. For more details, see Thymeleaf+Spring and the WebFlux integration announcement.

FreeMarker

Same as in Spring MVC

Apache FreeMarker is a template engine for generating any kind of text output from HTML to email and others. The Spring Framework has built-in integration for using Spring WebFlux with FreeMarker templates.

View Configuration

Same as in Spring MVC

The following example shows how to configure FreeMarker as a view technology:

Java

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

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();
    }

    // Configure FreeMarker...

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

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.freeMarker()
    }

    // Configure FreeMarker...

    @Bean
    fun freeMarkerConfigurer() = FreeMarkerConfigurer().apply {
        setTemplateLoaderPath("classpath:/templates/freemarker")
    }
}
```

Your templates need to be stored in the directory specified by the `FreeMarkerConfigurer`, shown in the preceding example. Given the preceding configuration, if your controller returns the view name, `welcome`, the resolver looks for the `classpath:/templates/freemarker/welcome.ftl` template.

FreeMarker Configuration

Same as in Spring MVC

You can pass FreeMarker 'Settings' and 'SharedVariables' directly to the FreeMarker `Configuration` object (which is managed by Spring) by setting the appropriate bean properties on the `FreeMarkerConfigurer` bean. The `freemarkerSettings` property requires a `java.util.Properties` object,

and the `freemarkerVariables` property requires a `java.util.Map`. The following example shows how to use a `FreeMarkerConfigurer`:

Java

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

    // ...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        Map<String, Object> variables = new HashMap<>();
        variables.put("xml_escape", new XmlEscape());

        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("classpath:/templates");
        configurer.setFreemarkerVariables(variables);
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    // ...

    @Bean
    fun freeMarkerConfigurer() = FreeMarkerConfigurer().apply {
        setTemplateLoaderPath("classpath:/templates")
        setFreemarkerVariables(mapOf("xml_escape" to XmlEscape()))
    }
}
```

See the FreeMarker documentation for details of settings and variables as they apply to the `Configuration` object.

Form Handling

Same as in Spring MVC

Spring provides a tag library for use in JSPs that contains, among others, a `<spring:bind/>` element. This element primarily lets forms display values from form-backing objects and show the results of failed validations from a `Validator` in the web or business tier. Spring also has support for the same functionality in FreeMarker, with additional convenience macros for generating form input elements themselves.

The Bind Macros

Same as in Spring MVC

A standard set of macros are maintained within the `spring-webflux.jar` file for FreeMarker, so they are always available to a suitably configured application.

Some of the macros defined in the Spring templating libraries are considered internal (private), but no such scoping exists in the macro definitions, making all macros visible to calling code and user templates. The following sections concentrate only on the macros you need to directly call from within your templates. If you wish to view the macro code directly, the file is called `spring.ftl` and is in the `org.springframework.web.reactive.result.view.freemarker` package.

For additional details on binding support, see Simple Binding for Spring MVC.

Form Macros

For details on Spring's form macro support for FreeMarker templates, consult the following sections of the Spring MVC documentation.

- Input Macros
- Input Fields
- Selection Fields
- HTML Escaping

Script Views

Same as in Spring MVC

The Spring Framework has a built-in integration for using Spring WebFlux with any templating library that can run on top of the JSR-223 Java scripting engine. The following table shows the templating libraries that we have tested on different script engines:

Scripting Library	Scripting Engine
Handlebars	Nashorn
Mustache	Nashorn
React	Nashorn
EJS	Nashorn
ERB	JRuby
String templates	Jython
Kotlin Script templating	Kotlin

The basic rule for integrating any other script engine is that it must implement the `ScriptEngine` and `Invocable` interfaces.

Requirements

Same as in Spring MVC

You need to have the script engine on your classpath, the details of which vary by script engine:

- The Nashorn JavaScript engine is provided with Java 8+. Using the latest update release available is highly recommended.
- JRuby should be added as a dependency for Ruby support.
- Jython should be added as a dependency for Python support.
- `org.jetbrains.kotlin:kotlin-script-util` dependency and a `META-INF/services/javax.script.ScriptEngineFactory` file containing a `org.jetbrains.kotlin.script.jsr223.KotlinJsr223JvmLocalScriptEngineFactory` line should be added for Kotlin script support. See this example for more detail.

You need to have the script templating library. One way to do that for Javascript is through WebJars.

Script Templates

Same as in Spring MVC

You can declare a `ScriptTemplateConfigurer` bean to specify the script engine to use, the script files to load, what function to call to render templates, and so on. The following example uses Mustache templates and the Nashorn JavaScript engine:

Java

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

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("mustache.js");
        configurer.setRenderObject("Mustache");
        configurer.setRenderFunction("render");
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.scriptTemplate()
    }

    @Bean
    fun configurer() = ScriptTemplateConfigurer().apply {
        engineName = "nashorn"
        setScripts("mustache.js")
        renderObject = "Mustache"
        renderFunction = "render"
    }
}
```

The `render` function is called with the following parameters:

- `String template`: The template content
- `Map model`: The view model
- `RenderingContext renderingContext`: The `RenderingContext` that gives access to the application context, the locale, the template loader, and the URL (since 5.0)

`Mustache.render()` is natively compatible with this signature, so you can call it directly.

If your templating technology requires some customization, you can provide a script that implements a custom render function. For example, Handlerbars needs to compile templates before using them and requires a polyfill in order to emulate some browser facilities not available in the server-side script engine. The following example shows how to set a custom render function:

Java

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

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("polyfill.js", "handlebars.js", "render.js");
        configurer.setRenderFunction("render");
        configurer.setSharedEngine(false);
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.scriptTemplate()
    }

    @Bean
    fun configurer() = ScriptTemplateConfigurer().apply {
        engineName = "nashorn"
        setScripts("polyfill.js", "handlebars.js", "render.js")
        renderFunction = "render"
        isSharedEngine = false
    }
}
```

Setting the `sharedEngine` property to `false` is required when using non-thread-safe script engines with templating libraries not designed for concurrency, such as Handlebars or React running on Nashorn. In that case, Java SE 8 update 60 is required, due to this bug, but it is generally recommended to use a recent Java SE patch release in any case.

`polyfill.js` defines only the `window` object needed by Handlebars to run properly, as the following snippet shows:

```
var window = {};
```

This basic `render.js` implementation compiles the template before using it. A production ready implementation should also store and reused cached templates or pre-compiled templates. This can be done on the script side, as well as any customization you need (managing template engine configuration for example). The following example shows how compile a template:

```
function render(template, model) {
    var compiledTemplate = Handlebars.compile(template);
    return compiledTemplate(model);
}
```

Check out the Spring Framework unit tests, Java, and resources, for more configuration examples.

JSON and XML

Same as in Spring MVC

For Content Negotiation purposes, it is useful to be able to alternate between rendering a model with an HTML template or as other formats (such as JSON or XML), depending on the content type requested by the client. To support doing so, Spring WebFlux provides the `HttpMessageWriterView`, which you can use to plug in any of the available Codecs from `spring-web`, such as `Jackson2JsonEncoder`, `Jackson2SmileEncoder`, or `Jaxb2XmlEncoder`.

Unlike other view technologies, `HttpMessageWriterView` does not require a `ViewResolver` but is instead configured as a default view. You can configure one or more such default views, wrapping different `HttpMessageWriter` instances or `Encoder` instances. The one that matches the requested content type is used at runtime.

In most cases, a model contains multiple attributes. To determine which one to serialize, you can configure `HttpMessageWriterView` with the name of the model attribute to use for rendering. If the model contains only one attribute, that one is used.

HTTP Caching

Same as in Spring MVC

HTTP caching can significantly improve the performance of a web application. HTTP caching revolves around the `Cache-Control` response header and subsequent conditional request headers, such as `Last-Modified` and `ETag`. `Cache-Control` advises private (for example, browser) and public (for example, proxy) caches how to cache and re-use responses. An `ETag` header is used to make a conditional request that may result in a 304 (NOT_MODIFIED) without a body, if the content has not changed. `ETag` can be seen as a more sophisticated successor to the `Last-Modified` header.

This section describes the HTTP caching related options available in Spring WebFlux.

CacheControl

Same as in Spring MVC

`CacheControl` provides support for configuring settings related to the `Cache-Control` header and is accepted as an argument in a number of places:

- Controllers
- Static Resources

While RFC 7234 describes all possible directives for the `Cache-Control` response header, the `CacheControl` type takes a use case-oriented approach that focuses on the common scenarios, as the following example shows:

Java

```
// Cache for an hour - "Cache-Control: max-age=3600"
CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// Prevent caching - "Cache-Control: no-store"
CacheControl ccNoStore = CacheControl.noStore();

// Cache for ten days in public and private caches,
// public caches should not transform the response
// "Cache-Control: max-age=864000, public, no-transform"
CacheControl ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS).noTransform()
.cachePublic();
```

Kotlin

```
// Cache for an hour - "Cache-Control: max-age=3600"
val ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS)

// Prevent caching - "Cache-Control: no-store"
val ccNoStore = CacheControl.noStore()

// Cache for ten days in public and private caches,
// public caches should not transform the response
// "Cache-Control: max-age=864000, public, no-transform"
val ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS).noTransform().cachePublic()
```

Controllers

Same as in Spring MVC

Controllers can add explicit support for HTTP caching. We recommend doing so, since the `lastModified` or `ETag` value for a resource needs to be calculated before it can be compared against conditional request headers. A controller can add an `ETag` and `Cache-Control` settings to a `ResponseEntity`, as the following example shows:

Java

```
@GetMapping("/book/{id}")
public ResponseEntity<Book> showBook(@PathVariable Long id) {

    Book book = findBook(id);
    String version = book.getVersion();

    return ResponseEntity
            .ok()
            .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
            .eTag(version) // lastModified is also available
            .body(book);
}
```

Kotlin

```
@GetMapping("/book/{id}")
fun showBook(@PathVariable id: Long): ResponseEntity<Book> {

    val book = findBook(id)
    val version = book.getVersion()

    return ResponseEntity
            .ok()
            .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
            .eTag(version) // lastModified is also available
            .body(book)
}
```

The preceding example sends a 304 (NOT_MODIFIED) response with an empty body if the comparison to the conditional request headers indicates the content has not changed. Otherwise, the `ETag` and `Cache-Control` headers are added to the response.

You can also make the check against conditional request headers in the controller, as the following example shows:

Java

```
@RequestMapping
public String myHandleMethod(ServerWebExchange exchange, Model model) {

    long eTag = ... ①

    if (exchange.checkNotModified(eTag)) {
        return null; ②
    }

    model.addAttribute(...); ③
    return "myViewName";
}
```

① Application-specific calculation.

② Response has been set to 304 (NOT_MODIFIED). No further processing.

③ Continue with request processing.

Kotlin

```
@RequestMapping
fun myHandleMethod(exchange: ServerWebExchange, model: Model): String? {

    val eTag: Long = ... ①

    if (exchange.checkNotModified(eTag)) {
        return null②
    }

    model.addAttribute(...) ③
    return "myViewName"
}
```

① Application-specific calculation.

② Response has been set to 304 (NOT_MODIFIED). No further processing.

③ Continue with request processing.

There are three variants for checking conditional requests against `eTag` values, `lastModified` values, or both. For conditional `GET` and `HEAD` requests, you can set the response to 304 (NOT_MODIFIED). For conditional `POST`, `PUT`, and `DELETE`, you can instead set the response to 409 (PRECONDITION_FAILED) to prevent concurrent modification.

Static Resources

Same as in Spring MVC

You should serve static resources with a `Cache-Control` and conditional response headers for optimal performance. See the section on configuring Static Resources.

WebFlux Config

Same as in Spring MVC

The WebFlux Java configuration declares the components that are required to process requests with annotated controllers or functional endpoints, and it offers an API to customize the configuration. That means you do not need to understand the underlying beans created by the Java configuration. However, if you want to understand them, you can see them in `WebFluxConfigurationSupport` or read more about what they are in Special Bean Types.

For more advanced customizations, not available in the configuration API, you can gain full control over the configuration through the Advanced Configuration Mode.

Enabling WebFlux Config

Same as in Spring MVC

You can use the `@EnableWebFlux` annotation in your Java config, as the following example shows:

Java

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

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig
```

The preceding example registers a number of Spring WebFlux infrastructure beans and adapts to dependencies available on the classpath — for JSON, XML, and others.

WebFlux config API

Same as in Spring MVC

In your Java configuration, you can implement the `WebFluxConfigurer` interface, as the following example shows:

Java

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

    // Implement configuration methods...
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    // Implement configuration methods...
}
```

Conversion, formatting

Same as 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.

The following example shows how to register custom formatters and converters:

Java

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

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

}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun addFormatters(registry: FormatterRegistry) {
        // ...
    }
}
```

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

Validation

Same as in Spring MVC

By default, if Bean Validation is present on the classpath (for example, the Hibernate Validator), the `LocalValidatorFactoryBean` is registered as a global validator for use with `@Valid` and `Validated` on `@Controller` method arguments.

In your Java configuration, you can customize the global `Validator` instance, as the following example shows:

Java

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

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

}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun getValidator(): Validator {
        // ...
    }

}
```

Note that you can also register `Validator` implementations locally, as the following example shows:

Java

```
@Controller
public class MyController {

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

}
```

Kotlin

```
@Controller
class MyController {

    @InitBinder
    protected fun initBinder(binder: WebDataBinder) {
        binder.addValidators(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.

Content Type Resolvers

Same as in Spring MVC

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

The following example shows how to customize the requested content type resolution:

Java

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

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

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureContentTypeResolver(builder:
RequestedContentTypeResolverBuilder) {
        // ...
    }
}
```

HTTP message codecs

Same as in Spring MVC

The following example shows how to customize how the request and response body are read and written:

Java

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

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

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureHttpMessageCodecs(configurer: ServerCodecConfigurer) {
        // ...
    }
}
```

`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 `Jackson2ObjectMapperBuilder`, which customizes Jackson's default properties with the following ones:

- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled.
- `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled.

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

- `jackson-datatype-jdk7`: Support for Java 7 types like `java.nio.file.Path`.
- `jackson-datatype-joda`: Support for Joda-Time types.
- `jackson-datatype-jsr310`: Support for Java 8 Date and Time API types.
- `jackson-datatype-jdk8`: Support for other Java 8 types, such as `Optional`.

View Resolvers

Same as in Spring MVC

The following example shows how to configure view resolution:

Java

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

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

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        // ...
    }
}
```

The `ViewResolverRegistry` has shortcuts for view technologies with which the Spring Framework integrates. The following example uses FreeMarker (which also requires configuring the underlying FreeMarker view technology):

Java

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


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();
    }

    // Configure Freemarker...

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

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.freeMarker()
    }

    // Configure Freemarker...

    @Bean
    fun freeMarkerConfigurer() = FreeMarkerConfigurer().apply {
        setTemplateLoaderPath("classpath:/templates")
    }
}
```

You can also plug in any `ViewResolver` implementation, as the following example shows:

Java

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


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        ViewResolver resolver = ... ;
        registry.viewResolver(resolver);
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        val resolver: ViewResolver = ...
        registry.viewResolver(resolver
    }
}
```

To support Content Negotiation and rendering other formats through view resolution (besides HTML), you can configure one or more default views based on the `HttpMessageWriterView` implementation, which accepts any of the available Codecs from `spring-web`. The following example

shows how to do so:

Java

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


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();

        Jackson2JsonEncoder encoder = new Jackson2JsonEncoder();
        registry.defaultViews(new HttpMessageWriterView(encoder));
    }

    // ...
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {


    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.freeMarker()

        val encoder = Jackson2JsonEncoder()
        registry.defaultViews(HttpMessageWriterView(encoder))
    }

    // ...
}
```

See View Technologies for more on the view technologies that are integrated with Spring WebFlux.

Static Resources

Same as in Spring MVC

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

In the next example, 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 are served with a one-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. The following list shows the example:

Java

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

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("/public", "classpath:/static/")
            .setCacheControl(CacheControl.maxAge(365, TimeUnit.DAYS));
    }

}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun addResourceHandlers(registry: ResourceHandlerRegistry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public", "classpath:/static/")
                .setCacheControl(CacheControl.maxAge(365, TimeUnit.DAYS))
    }
}
```

The resource handler also supports a chain of `ResourceResolver` implementations and `ResourceTransformer` implementations, which can be used to create a toolchain for working with optimized resources.

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

The following example shows how to use `VersionResourceResolver` in your Java configuration:

Java

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

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

}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun addResourceHandlers(registry: ResourceHandlerRegistry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public/")
                .resourceChain(true)

.addResolver(VersionResourceResolver().addContentVersionStrategy("/**"))
    }

}
```

You can use `ResourceUrlProvider` to rewrite URLs and apply the full chain of resolvers and transformers (for example, to insert versions). The WebFlux configuration provides a `ResourceUrlProvider` so that it can be injected into others.

Unlike Spring MVC, at present, in WebFlux, there is no way to transparently rewrite static resource URLs, since there are no view technologies that can make use of a non-blocking chain of resolvers and transformers. When serving only local resources, the workaround is to use `ResourceUrlProvider` directly (for example, through a custom element) and block.

Note that, when using both `EncodedResourceResolver` (for example, Gzip, Brotli encoded) and `VersionedResourceResolver`, they must be registered in that order, to ensure content-based versions are always computed reliably based on the unencoded file.

WebJars are also supported through the `WebJarsResourceResolver` which is automatically registered when the `org.webjars:webjars-locator-core` library is present on the classpath. The resolver can re-write URLs to include the version of the jar and can also match against incoming URLs without

versions — for example, from `/jquery/jquery.min.js` to `/jquery/1.2.0/jquery.min.js`.

Path Matching

Same as in Spring MVC

You can customize options related to path matching. For details on the individual options, see the `PathMatchConfigurer` javadoc. The following example shows how to use `PathMatchConfigurer`:

Java

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

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer
            .setUseCaseSensitiveMatch(true)
            .setUseTrailingSlashMatch(false)
            .addPathPrefix("/api",
                    HandlerTypePredicate.forAnnotation(RestController.class));
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    @Override
    fun configurePathMatch(configurer: PathMatchConfigurer) {
        configurer
            .setUseCaseSensitiveMatch(true)
            .setUseTrailingSlashMatch(false)
            .addPathPrefix("/api",
                    HandlerTypePredicate.forAnnotation(RestController::class.java))
    }
}
```

Spring WebFlux relies on a parsed representation of the request path called `RequestPath` for access to decoded path segment values, with semicolon content removed (that is, path or matrix variables). That means, unlike in Spring MVC, you need not indicate whether to decode the request path nor whether to remove semicolon content for path matching purposes.

Spring WebFlux also does not support suffix pattern matching, unlike in Spring MVC, where we are also recommend moving away from reliance on it.

Advanced Configuration Mode

Same as in Spring MVC

`@EnableWebFlux` imports `DelegatingWebFluxConfiguration` that:

- Provides default Spring configuration for WebFlux applications
- detects and delegates to `WebFluxConfigurer` implementations to customize that configuration.

For advanced mode, you can remove `@EnableWebFlux` and extend directly from `DelegatingWebFluxConfiguration` instead of implementing `WebFluxConfigurer`, as the following example shows:

Java

```
@Configuration
public class WebConfig extends DelegatingWebFluxConfiguration {

    // ...
}
```

Kotlin

```
@Configuration
class WebConfig : DelegatingWebFluxConfiguration {

    // ...
}
```

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

HTTP/2

Same as in Spring MVC

Servlet 4 containers are required to support HTTP/2, and Spring Framework 5 is compatible with Servlet API 4. From a programming model perspective, there is nothing specific that applications need to do. However, there are considerations related to server configuration. For more details, see the HTTP/2 wiki page.

Currently, Spring WebFlux does not support HTTP/2 with Netty. There is also no support for pushing resources programmatically to the client.

WebClient

Spring WebFlux includes a reactive, non-blocking `WebClient` for HTTP requests. The client has a functional, fluent API with reactive types for declarative composition, see Reactive Libraries. WebFlux client and server rely on the same non-blocking codecs to encode and decode request and response content.

Internally `WebClient` delegates to an HTTP client library. By default, it uses Reactor Netty, there is built-in support for the Jetty reactive HttpClient, and others can be plugged in through a `ClientHttpConnector`.

Configuration

The simplest way to create a `WebClient` is through one of the static factory methods:

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

The above methods use the Reactor Netty `HttpClient` with default settings and expect `io.projectreactor.netty:reactor-netty` to be on the classpath.

You can also use `WebClient.builder()` with further options:

- `uriBuilderFactory`: Customized `UriBuilderFactory` to use as a base URL.
- `defaultHeader`: Headers for every request.
- `defaultCookie`: Cookies for every request.
- `defaultRequest`: `Consumer` to customize every request.
- `filter`: Client filter for every request.
- `exchangeStrategies`: HTTP message reader/writer customizations.
- `clientConnector`: HTTP client library settings.

The following example configures HTTP codecs:

Java

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

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

Kotlin

```
val strategies = ExchangeStrategies.builder()
        .codecs {
            // ...
        }
        .build()

val client = WebClient.builder()
        .exchangeStrategies(strategies)
        .build()
```

Once built, a `WebClient` instance is immutable. However, you can clone it and build a modified copy without affecting the original instance, as the following example shows:

Java

```
WebClient client1 = WebClient.builder()
        .filter(filterA).filter(filterB).build();

WebClient client2 = client1.mutate()
        .filter(filterC).filter(filterD).build();

// client1 has filterA, filterB

// client2 has filterA, filterB, filterC, filterD
```

Kotlin

```
val client1 = WebClient.builder()
        .filter(filterA).filter(filterB).build()

val client2 = client1.mutate()
        .filter(filterC).filter(filterD).build()

// client1 has filterA, filterB

// client2 has filterA, filterB, filterC, filterD
```

Reactor Netty

To customize Reactor Netty settings, simple provide a pre-configured `HttpClient`:

Java

```
HttpClient httpClient = HttpClient.create().secure(sslSpec -> ...);

WebClient webClient = WebClient.builder()
        .clientConnector(new ReactorClientHttpConnector(httpClient))
        .build();
```

Kotlin

```
val httpClient = HttpClient.create().secure { ... }

val webClient = WebClient.builder()
    .clientConnector(ReactorClientHttpConnector(httpClient))
    .build()
```

Resources

By default, `HttpClient` participates in the global Reactor Netty resources held in `reactor.netty.http.HttpResources`, including event loop threads and a connection pool. This is the recommended mode, since fixed, shared resources are preferred for event loop concurrency. In this mode global resources remain active until the process exits.

If the server is timed with the process, there is typically no need for an explicit shutdown. However, if the server can start or stop in-process (for example, a Spring MVC application deployed as a WAR), you can declare a Spring-managed bean of type `ReactorResourceFactory` with `globalResources=true` (the default) to ensure that the Reactor Netty global resources are shut down when the Spring `ApplicationContext` is closed, as the following example shows:

Java

```
@Bean
public ReactorResourceFactory reactorResourceFactory() {
    return new ReactorResourceFactory();
}
```

Kotlin

```
@Bean
fun reactorResourceFactory() = ReactorResourceFactory()
```

You can also choose not to participate in the global Reactor Netty resources. However, in this mode, the burden is on you to ensure that all Reactor Netty client and server instances use shared resources, as the following example shows:

Java

```
@Bean
public ReactorResourceFactory resourceFactory() {
    ReactorResourceFactory factory = new ReactorResourceFactory();
    factory.setUseGlobalResources(false); ①
    return factory;
}

@Bean
public WebClient webClient() {

    Function<HttpClient, HttpClient> mapper = client -> {
        // Further customizations...
    };

    ClientHttpConnector connector =
            new ReactorClientHttpConnector(resourceFactory(), mapper); ②

    return WebClient.builder().clientConnector(connector).build(); ③
}
```

① Create resources independent of global ones.

② Use the `ReactorClientHttpConnector` constructor with resource factory.

③ Plug the connector into the `WebClient.Builder`.

Kotlin

```
@Bean
fun resourceFactory() = ReactorResourceFactory().apply {
    isUseGlobalResources = false ①
}

@Bean
fun webClient(): WebClient {

    val mapper: (HttpClient) -> HttpClient = {
        // Further customizations...
    }

    val connector = ReactorClientHttpConnector(resourceFactory(), mapper) ②

    return WebClient.builder().clientConnector(connector).build() ③
}
```

① Create resources independent of global ones.

② Use the `ReactorClientHttpConnector` constructor with resource factory.

③ Plug the connector into the `WebClient.Builder`.

Timeouts

To configure a connection timeout:

Java

```
import io.netty.channel.ChannelOption;

HttpClient httpClient = HttpClient.create()
        .tcpConfiguration(client ->
                client.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000));
```

Kotlin

```
import io.netty.channel.ChannelOption

val httpClient = HttpClient.create()
        .tcpConfiguration { it.option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000)}
```

To configure a read and/or write timeout values:

Java

```
import io.netty.handler.timeout.ReadTimeoutHandler;
import io.netty.handler.timeout.WriteTimeoutHandler;

HttpClient httpClient = HttpClient.create()
        .tcpConfiguration(client ->
                client.doOnConnected(conn -> conn
                        .addHandlerLast(new ReadTimeoutHandler(10))
                        .addHandlerLast(new WriteTimeoutHandler(10))));
```

Kotlin

```
import io.netty.handler.timeout.ReadTimeoutHandler
import io.netty.handler.timeout.WriteTimeoutHandler

val httpClient = HttpClient.create().tcpConfiguration {
    it.doOnConnected { conn -> conn
            .addHandlerLast(ReadTimeoutHandler(10))
            .addHandlerLast(WriteTimeoutHandler(10))
    }
}
```

Jetty

The following example shows how to customize Jetty `HttpClient` settings:

Java

```
HttpClient httpClient = new HttpClient();
httpClient.setCookieStore(...);
ClientHttpConnector connector = new JettyClientHttpConnector(httpClient);

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

Kotlin

```
val httpClient = HttpClient()
httpClient.cookieStore = ...
val connector = JettyClientHttpConnector(httpClient)

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

By default, `HttpClient` creates its own resources (`Executor`, `ByteBufferPool`, `Scheduler`), which remain active until the process exits or `stop()` is called.

You can share resources between multiple instances of the Jetty client (and server) and ensure that the resources are shut down when the Spring `ApplicationContext` is closed by declaring a Spring-managed bean of type `JettyResourceFactory`, as the following example shows:

Java

```
@Bean
public JettyResourceFactory resourceFactory() {
    return new JettyResourceFactory();
}

@Bean
public WebClient webClient() {

    HttpClient httpClient = new HttpClient();
    // Further customizations...

    ClientHttpConnector connector =
            new JettyClientHttpConnector(httpClient, resourceFactory()); ①

    return WebClient.builder().clientConnector(connector).build(); ②
}
```

① Use the `JettyClientHttpConnector` constructor with resource factory.

② Plug the connector into the `WebClient.Builder`.

Kotlin

```
@Bean
fun resourceFactory() = JettyResourceFactory()

@Bean
fun webClient(): WebClient {

    val httpClient = HttpClient()
    // Further customizations...

    val connector = JettyClientHttpConnector(httpClient, resourceFactory()) ①

    return WebClient.builder().clientConnector(connector).build() ②
}
```

① Use the `JettyClientHttpConnector` constructor with resource factory.

② Plug the connector into the `WebClient.Builder`.

`retrieve()`

The `retrieve()` method is the easiest way to get a response body and decode it. The following example shows how to do so:

Java

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

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

Kotlin

```
val client = WebClient.create("https://example.org")

val result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .awaitBody<Person>()
```

You can also get a stream of objects decoded from the response, as the following example shows:

Java

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

Kotlin

```
val result = client.get()
        .uri("/quotes").accept(MediaType.TEXT_EVENT_STREAM)
        .retrieve()
        .bodyToFlow<Quote>()
```

By default, responses with 4xx or 5xx status codes result in an `WebClientResponseException` or one of its HTTP status specific sub-classes, such as `WebClientResponseException.BadRequest`, `WebClientResponseException.NotFound`, and others. You can also use the `onStatus` method to customize the resulting exception, as the following example shows:

Java

```
Mono<Person> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .onStatus(HttpStatus::is4xxClientError, response -> ...)
        .onStatus(HttpStatus::is5xxServerError, response -> ...)
        .bodyToMono(Person.class);
```

Kotlin

```
val result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .onStatus(HttpStatus::is4xxClientError) { ... }
        .onStatus(HttpStatus::is5xxServerError) { ... }
        .awaitBody<Person>()
```

When `onStatus` is used, if the response is expected to have content, then the `onStatus` callback should consume it. If not, the content will be automatically drained to ensure resources are released.

exchange()

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

Java

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

Kotlin

```
val result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .awaitExchange()
        .awaitBody<Person>()
```

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

Java

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

Kotlin

```
val result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .awaitExchange()
        .toEntity<Person>()
```

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 using `exchange()`, you have to make sure that the body is always consumed or released, even when an exception occurs (see Using DataBuffer). Typically, you do this by invoking either `bodyTo*` or `toEntity*` on `ClientResponse` to convert the body into an object of the desired type, but you can also invoke `releaseBody()` to discard the body contents without consuming it or `toBodilessEntity()` to get just the status and headers (while discarding the body).

Finally, there is `bodyToMono(Void.class)`, which should only be used if no response content is expected. If the response does have content, the connection is closed and is not placed back in the pool, because it is not left in a reusable state.

Request Body

The request body can be encoded from any asynchronous type handled by `ReactiveAdapterRegistry`,

like `Mono` or Kotlin Coroutines `Deferred` as the following example shows:

Java

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

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

Kotlin

```
val personDeferred: Deferred<Person> = ...

client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .body<Person>(personDeferred)
        .retrieve()
        .awaitBody<Unit>()
```

You can also have a stream of objects be encoded, as the following example shows:

Java

```
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);
```

Kotlin

```
val people: Flow<Person> = ...

client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .body(people)
        .retrieve()
        .awaitBody<Unit>()
```

Alternatively, if you have the actual value, you can use the `bodyValue` shortcut method, as the

following example shows:

Java

```
Person person = ... ;

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

Kotlin

```
val person: Person = ...

client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .bodyValue(person)
        .retrieve()
        .awaitBody<Unit>()
```

Form Data

To send form data, you can provide a `MultiValueMap<String, String>` as the body. Note that the content is automatically set to `application/x-www-form-urlencoded` by the `FormHttpMessageWriter`. The following example shows how to use `MultiValueMap<String, String>`:

Java

```
MultiValueMap<String, String> formData = ... ;

Mono<Void> result = client.post()
        .uri("/path", id)
        .bodyValue(formData)
        .retrieve()
        .bodyToMono(Void.class);
```

Kotlin

```
val formData: MultiValueMap<String, String> = ...

client.post()
        .uri("/path", id)
        .bodyValue(formData)
        .retrieve()
        .awaitBody<Unit>()
```

You can also supply form data in-line by using `BodyInserters`, as the following example shows:

Java

```
import static org.springframework.web.reactive.function.BodyInserters.*;

Mono<Void> result = client.post()
        .uri("/path", id)
        .body(fromFormData("k1", "v1").with("k2", "v2"))
        .retrieve()
        .bodyToMono(Void.class);
```

Kotlin

```
import org.springframework.web.reactive.function.BodyInserters.*

client.post()
        .uri("/path", id)
        .body(fromFormData("k1", "v1").with("k2", "v2"))
        .retrieve()
        .awaitBody<Unit>()
```

Multipart Data

To send multipart data, you need to provide a `MultiValueMap<String, ?>` whose values are either `Object` instances that represent part content or `HttpEntity` instances that represent the content and headers for a part. `MultipartBodyBuilder` provides a convenient API to prepare a multipart request. The following example shows how to create a `MultiValueMap<String, ?>`:

Java

```
MultipartBodyBuilder builder = new MultipartBodyBuilder();
builder.part("fieldPart", "fieldValue");
builder.part("filePart1", new FileSystemResource("...logo.png"));
builder.part("jsonPart", new Person("Jason"));
builder.part("myPart", part); // Part from a server request

MultiValueMap<String, HttpEntity<?>> parts = builder.build();
```

Kotlin

```
val builder = MultipartBodyBuilder().apply {
    part("fieldPart", "fieldValue")
    part("filePart1", new FileSystemResource("...logo.png"))
    part("jsonPart", new Person("Jason"))
    part("myPart", part) // Part from a server request
}

val parts = builder.build()
```

In most cases, you do not have to specify the `Content-Type` for each part. The content type is determined automatically based on the `HttpMessageWriter` chosen to serialize it or, in the case of a `Resource`, based on the file extension. If necessary, you can explicitly provide the `MediaType` to use for each part through one of the overloaded builder `part` methods.

Once a `MultiValueMap` is prepared, the easiest way to pass it to the the `WebClient` is through the `body` method, as the following example shows:

Java

```
MultipartBodyBuilder builder = ...;

Mono<Void> result = client.post()
        .uri("/path", id)
        .body(builder.build())
        .retrieve()
        .bodyToMono(Void.class);
```

Kotlin

```
val builder: MultipartBodyBuilder = ...

client.post()
        .uri("/path", id)
        .body(builder.build())
        .retrieve()
        .awaitBody<Unit>()
```

If the `MultiValueMap` contains at least one non-`String` value, which could also represent regular form data (that is, `application/x-www-form-urlencoded`), you need not set the `Content-Type` to `multipart/form-data`. This is always the case when using `MultipartBodyBuilder`, which ensures an `HttpEntity` wrapper.

As an alternative to `MultipartBodyBuilder`, you can also provide multipart content, inline-style, through the built-in `BodyInserters`, as the following example shows:

Java

```
import static org.springframework.web.reactive.function.BodyInserters.*;

Mono<Void> result = client.post()
        .uri("/path", id)
        .body(fromMultipartData("fieldPart", "value").with("filePart", resource))
        .retrieve()
        .bodyToMono(Void.class);
```

Kotlin

```
import org.springframework.web.reactive.function.BodyInserters.*

client.post()
        .uri("/path", id)
        .body(fromMultipartData("fieldPart", "value").with("filePart", resource))
        .retrieve()
        .awaitBody<Unit>()
```

Client Filters

You can register a client filter (`ExchangeFilterFunction`) through the `WebClient.Builder` in order to intercept and modify requests, as the following example shows:

Java

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

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

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

Kotlin

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

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

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

This can be used for cross-cutting concerns, such as authentication. The following example uses a filter for basic authentication through a static factory method:

Java

```
import static
org.springframework.web.reactive.function.client.ExchangeFilterFunctions.basicAuthenti
cation;

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

Kotlin

```
import
org.springframework.web.reactive.function.client.ExchangeFilterFunctions.basicAuthenti
cation

val client = WebClient.builder()
        .filter(basicAuthentication("user", "password"))
        .build()
```

Filters apply globally to every request. To change a filter's behavior for a specific request, you can add request attributes to the `ClientRequest` that can then be accessed by all filters in the chain, as the following example shows:

Java

```
WebClient client = WebClient.builder()
        .filter((request, next) -> {
            Optional<Object> usr = request.attribute("myAttribute");
            // ...
        })
        .build();

client.get().uri("https://example.org/")
        .attribute("myAttribute", "...")
        .retrieve()
        .bodyToMono(Void.class);

    }
```

Kotlin

```
val client = WebClient.builder()
            .filter { request, _ ->
        val usr = request.attributes()["myAttribute"];
        // ...
    }.build()

    client.get().uri("https://example.org/")
            .attribute("myAttribute", "...")
            .retrieve()
            .awaitBody<Unit>()
```

You can also replicate an existing `WebClient`, insert new filters, or remove already registered filters. The following example, inserts a basic authentication filter at index 0:

Java

```
import static
org.springframework.web.reactive.function.client.ExchangeFilterFunctions.basicAuthenti
cation;

WebClient client = webClient.mutate()
        .filters(filterList -> {
            filterList.add(0, basicAuthentication("user", "password"));
        })
        .build();
```

Kotlin

```
val client = webClient.mutate()
        .filters { it.add(0, basicAuthentication("user", "password")) }
        .build()
```

Synchronous Use

`WebClient` can be used in synchronous style by blocking at the end for the result:

Java

```
Person person = client.get().uri("/person/{id}", i).retrieve()
    .bodyToMono(Person.class)
    .block();

List<Person> persons = client.get().uri("/persons").retrieve()
    .bodyToFlux(Person.class)
    .collectList()
    .block();
```

Kotlin

```
val person = runBlocking {
    client.get().uri("/person/{id}", i).retrieve()
            .awaitBody<Person>()
}

val persons = runBlocking {
    client.get().uri("/persons").retrieve()
            .bodyToFlow<Person>()
            .toList()
}
```

However if multiple calls need to be made, it's more efficient to avoid blocking on each response individually, and instead wait for the combined result:

Java

```
Mono<Person> personMono = client.get().uri("/person/{id}", personId)
        .retrieve().bodyToMono(Person.class);

Mono<List<Hobby>> hobbiesMono = client.get().uri("/person/{id}/hobbies", personId)
        .retrieve().bodyToFlux(Hobby.class).collectList();

Map<String, Object> data = Mono.zip(personMono, hobbiesMono, (person, hobbies) -> {
            Map<String, String> map = new LinkedHashMap<>();
            map.put("person", person);
            map.put("hobbies", hobbies);
            return map;
        })
        .block();
```

Kotlin

```
val data = runBlocking {
        val personDeferred = async {
            client.get().uri("/person/{id}", personId)
                    .retrieve().awaitBody<Person>()
        }

        val hobbiesDeferred = async {
            client.get().uri("/person/{id}/hobbies", personId)
                    .retrieve().bodyToFlow<Hobby>().toList()
        }

        mapOf("person" to personDeferred.await(), "hobbies" to
hobbiesDeferred.await())
    }
```

The above is merely one example. There are lots of other patterns and operators for putting together a reactive pipeline that makes many remote calls, potentially some nested, inter-dependent, without ever blocking until the end.

With `Flux` or `Mono`, you should never have to block in a Spring MVC or Spring WebFlux controller. Simply return the resulting reactive type from the controller method. The same principle apply to Kotlin Coroutines and Spring WebFlux, just use suspending function or return `Flow` in your controller method .

Testing

To test code that uses the `WebClient`, you can use a mock web server, such as the OkHttp MockWebServer. To see an example of its use, check out `WebClientIntegrationTests` in the Spring Framework test suite or the `static-server` sample in the OkHttp repository.

WebSockets

Same as in the Servlet stack

This part of the reference documentation covers support for reactive-stack WebSocket messaging.

Introduction to WebSocket

The WebSocket protocol, RFC 6455, provides a standardized way to establish a full-duplex, two-way communication channel between client and server over a single TCP connection. It is a different TCP protocol from HTTP but is designed to work over HTTP, using ports 80 and 443 and allowing re-use of existing firewall rules.

A WebSocket interaction begins with an HTTP request that uses the HTTP `Upgrade` header to upgrade or, in this case, to switch to the WebSocket protocol. The following example shows such an interaction:

```
GET /spring-websocket-portfolio/portfolio HTTP/1.1
Host: localhost:8080
Upgrade: websocket ①
Connection: Upgrade ②
Sec-WebSocket-Key: Uc9l9TMkWGbHFD2qnFHltg==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
```

① The `Upgrade` header.

② Using the `Upgrade` connection.

Instead of the usual 200 status code, a server with WebSocket support returns output similar to the following:

```
HTTP/1.1 101 Switching Protocols ①
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1qVdfYHU9hPOl4JYYNXF623Gzn0=
Sec-WebSocket-Protocol: v10.stomp
```

① Protocol switch

After a successful handshake, the TCP socket underlying the HTTP upgrade request remains open for both the client and the server to continue to send and receive messages.

A complete introduction of how WebSockets work is beyond the scope of this document. See RFC 6455, the WebSocket chapter of HTML5, or any of the many introductions and tutorials on the Web.

Note that, if a WebSocket server is running behind a web server (e.g. nginx), you likely need to configure it to pass WebSocket upgrade requests on to the WebSocket server. Likewise, if the

application runs in a cloud environment, check the instructions of the cloud provider related to WebSocket support.

HTTP Versus WebSocket

Even though WebSocket is designed to be HTTP-compatible and starts with an HTTP request, it is important to understand that the two protocols lead to very different architectures and application programming models.

In HTTP and REST, an application is modeled as many URLs. To interact with the application, clients access those URLs, request-response style. Servers route requests to the appropriate handler based on the HTTP URL, method, and headers.

By contrast, in WebSockets, there is usually only one URL for the initial connect. Subsequently, all application messages flow on that same TCP connection. This points to an entirely different asynchronous, event-driven, messaging architecture.

WebSocket is also a low-level transport protocol, which, unlike HTTP, does not prescribe any semantics to the content of messages. That means that there is no way to route or process a message unless the client and the server agree on message semantics.

WebSocket clients and servers can negotiate the use of a higher-level, messaging protocol (for example, STOMP), through the `Sec-WebSocket-Protocol` header on the HTTP handshake request. In the absence of that, they need to come up with their own conventions.

When to Use WebSockets

WebSockets can make a web page be dynamic and interactive. However, in many cases, a combination of Ajax and HTTP streaming or long polling can provide a simple and effective solution.

For example, news, mail, and social feeds need to update dynamically, but it may be perfectly okay to do so every few minutes. Collaboration, games, and financial apps, on the other hand, need to be much closer to real-time.

Latency alone is not a deciding factor. If the volume of messages is relatively low (for example, monitoring network failures) HTTP streaming or polling can provide an effective solution. It is the combination of low latency, high frequency, and high volume that make the best case for the use of WebSocket.

Keep in mind also that over the Internet, restrictive proxies that are outside of your control may preclude WebSocket interactions, either because they are not configured to pass on the `Upgrade` header or because they close long-lived connections that appear idle. This means that the use of WebSocket for internal applications within the firewall is a more straightforward decision than it is for public facing applications.

WebSocket API

Same as in the Servlet stack

The Spring Framework provides a WebSocket API that you can use to write client- and server-side applications that handle WebSocket messages.

Server

Same as in the Servlet stack

To create a WebSocket server, you can first create a `WebSocketHandler`. The following example shows how to do so:

Java

```
import org.springframework.web.reactive.socket.WebSocketHandler;
import org.springframework.web.reactive.socket.WebSocketSession;

public class MyWebSocketHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {
        // ...
    }
}
```

Kotlin

```
import org.springframework.web.reactive.socket.WebSocketHandler
import org.springframework.web.reactive.socket.WebSocketSession

class MyWebSocketHandler : WebSocketHandler {

    override fun handle(session: WebSocketSession): Mono<Void> {
        // ...
    }
}
```

Then you can map it to a URL and add a `WebSocketHandlerAdapter`, as the following example shows:

Java

```
@Configuration
class WebConfig {

    @Bean
    public HandlerMapping handlerMapping() {
        Map<String, WebSocketHandler> map = new HashMap<>();
        map.put("/path", new MyWebSocketHandler());
        int order = -1; // before annotated controllers

        return new SimpleUrlHandlerMapping(map, order);
    }

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

Kotlin

```
@Configuration
class WebConfig {

    @Bean
    fun handlerMapping(): HandlerMapping {
        val map = mapOf("/path" to MyWebSocketHandler())
        val order = -1 // before annotated controllers

        return SimpleUrlHandlerMapping(map, order)
    }

    @Bean
    fun handlerAdapter() =  WebSocketHandlerAdapter()
}
```

WebSocketHandler

The `handle` method of `WebSocketHandler` takes `WebSocketSession` and returns `Mono<Void>` to indicate when application handling of the session is complete. The session is handled through two streams, one for inbound and one for outbound messages. The following table describes the two methods that handle the streams:

`WebSocketSession` **method**	**Description**
`Flux<WebSocketMessage> receive()`	Provides access to the inbound message stream and completes when the connection is closed.

`WebSocketSession` method	Description
`Mono<Void> send(Publisher<WebSocketMessage>)`	Takes a source for outgoing messages, writes the messages, and returns a `Mono<Void>` that completes when the source completes and writing is done.

A `WebSocketHandler` must compose the inbound and outbound streams into a unified flow and return a `Mono<Void>` that reflects the completion of that flow. Depending on application requirements, the unified flow completes when:

- Either the inbound or the outbound message stream completes.
- The inbound stream completes (that is, the connection closed), while the outbound stream is infinite.
- At a chosen point, through the `close` method of `WebSocketSession`.

When inbound and outbound message streams are composed together, there is no need to check if the connection is open, since Reactive Streams signals terminate activity. The inbound stream receives a completion or error signal, and the outbound stream receives a cancellation signal.

The most basic implementation of a handler is one that handles the inbound stream. The following example shows such an implementation:

Java

```
class ExampleHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {
        return session.receive()            ①
                .doOnNext(message -> {
                    // ...                  ②
                })
                .concatMap(message -> {
                    // ...                  ③
                })
                .then();                    ④
    }
}
```

① Access the stream of inbound messages.

② Do something with each message.

③ Perform nested asynchronous operations that use the message content.

④ Return a `Mono<Void>` that completes when receiving completes.

Kotlin

```
class ExampleHandler : WebSocketHandler {

    override fun handle(session: WebSocketSession): Mono<Void> {
        return session.receive()            ①
                .doOnNext {
                    // ...                  ②
                }
                .concatMap {
                    // ...                  ③
                }
                .then()                     ④
    }
}
```

① Access the stream of inbound messages.

② Do something with each message.

③ Perform nested asynchronous operations that use the message content.

④ Return a `Mono<Void>` that completes when receiving completes.

For nested, asynchronous operations, you may need to call `message.retain()` on underlying servers that use pooled data buffers (for example, Netty). Otherwise, the data buffer may be released before you have had a chance to read the data. For more background, see Data Buffers and Codecs.

The following implementation combines the inbound and outbound streams:

Java

```
class ExampleHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {

        Flux<WebSocketMessage> output = session.receive()              ①
                .doOnNext(message -> {
                    // ...
                })
                .concatMap(message -> {
                    // ...
                })
                .map(value -> session.textMessage("Echo " + value));    ②

        return session.send(output);                                   ③
    }
}
```

① Handle the inbound message stream.

② Create the outbound message, producing a combined flow.

③ Return a `Mono<Void>` that does not complete while we continue to receive.

Kotlin

```
class ExampleHandler : WebSocketHandler {

    override fun handle(session: WebSocketSession): Mono<Void> {

        val output = session.receive()                          ①
                .doOnNext {
                    // ...
                }
                .concatMap {
                    // ...
                }
                .map { session.textMessage("Echo $it") }        ②

        return session.send(output)                             ③
    }
}
```

① Handle the inbound message stream.

② Create the outbound message, producing a combined flow.

③ Return a `Mono<Void>` that does not complete while we continue to receive.

Inbound and outbound streams can be independent and be joined only for completion, as the following example shows:

Java

```
class ExampleHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {

        Mono<Void> input = session.receive()                                ①
                .doOnNext(message -> {
                    // ...
                })
                .concatMap(message -> {
                    // ...
                })
                .then();

        Flux<String> source = ... ;
        Mono<Void> output = session.send(source.map(session::textMessage)); ②

        return Mono.zip(input, output).then();                              ③
    }
}
```

① Handle inbound message stream.

② Send outgoing messages.

③ Join the streams and return a `Mono<Void>` that completes when either stream ends.

Kotlin

```
class ExampleHandler : WebSocketHandler {

    override fun handle(session: WebSocketSession): Mono<Void> {

        val input = session.receive()                                       ①
                .doOnNext {
                    // ...
                }
                .concatMap {
                    // ...
                }
                .then()

        val source: Flux<String> = ...
        val output = session.send(source.map(session::textMessage))         ②

        return Mono.zip(input, output).then()                               ③
    }
}
```

① Handle inbound message stream.

② Send outgoing messages.

③ Join the streams and return a `Mono<Void>` that completes when either stream ends.

`DataBuffer`

`DataBuffer` is the representation for a byte buffer in WebFlux. The Spring Core part of the reference has more on that in the section on Data Buffers and Codecs. The key point to understand is that on some servers like Netty, byte buffers are pooled and reference counted, and must be released when consumed to avoid memory leaks.

When running on Netty, applications must use `DataBufferUtils.retain(dataBuffer)` if they wish to hold on input data buffers in order to ensure they are not released, and subsequently use `DataBufferUtils.release(dataBuffer)` when the buffers are consumed.

Handshake

Same as in the Servlet stack

`WebSocketHandlerAdapter` delegates to a `WebSocketService`. By default, that is an instance of `HandshakeWebSocketService`, which performs basic checks on the WebSocket request and then uses `RequestUpgradeStrategy` for the server in use. Currently, there is built-in support for Reactor Netty, Tomcat, Jetty, and Undertow.

`HandshakeWebSocketService` exposes a `sessionAttributePredicate` property that allows setting a `Predicate<String>` to extract attributes from the `WebSession` and insert them into the attributes of the `WebSocketSession`.

Server Configation

Same as in the Servlet stack

The `RequestUpgradeStrategy` for each server exposes WebSocket-related configuration options available for the underlying WebSocket engine. The following example sets WebSocket options when running on Tomcat:

Java

```
@Configuration
class WebConfig {

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

    @Bean
    public WebSocketService webSocketService() {
        TomcatRequestUpgradeStrategy strategy = new TomcatRequestUpgradeStrategy();
        strategy.setMaxSessionIdleTimeout(0L);
        return new HandshakeWebSocketService(strategy);
    }
}
```

Kotlin

```
@Configuration
class WebConfig {

    @Bean
    fun handlerAdapter() =
            WebSocketHandlerAdapter(webSocketService())

    @Bean
    fun webSocketService(): WebSocketService {
        val strategy = TomcatRequestUpgradeStrategy().apply {
            setMaxSessionIdleTimeout(0L)
        }
        return HandshakeWebSocketService(strategy)
    }
}
```

Check the upgrade strategy for your server to see what options are available. Currently, only Tomcat and Jetty expose such options.

CORS

Same as in the Servlet stack

The easiest way to configure CORS and restrict access to a WebSocket endpoint is to have your `WebSocketHandler` implement `CorsConfigurationSource` and return a `CorsConfiguraiton` with allowed origins, headers, and other details. If you cannot do that, you can also set the `corsConfigurations` property on the `SimpleUrlHandler` to specify CORS settings by URL pattern. If both are specified, they are combined by using the `combine` method on `CorsConfiguration`.

Client

Spring WebFlux provides a `WebSocketClient` abstraction with implementations for Reactor Netty, Tomcat, Jetty, Undertow, and standard Java (that is, JSR-356).

The Tomcat client is effectively an extension of the standard Java one with some extra functionality in the `WebSocketSession` handling to take advantage of the Tomcat-specific API to suspend receiving messages for back pressure.

To start a WebSocket session, you can create an instance of the client and use its `execute` methods:

Java

```
WebSocketClient client = new ReactorNettyWebSocketClient();

URI url = new URI("ws://localhost:8080/path");
client.execute(url, session ->
        session.receive()
                .doOnNext(System.out::println)
                .then());
```

Kotlin

```
val client = ReactorNettyWebSocketClient()

        val url = URI("ws://localhost:8080/path")
        client.execute(url) { session ->
            session.receive()
                    .doOnNext(::println)
            .then()
        }
```

Some clients, such as Jetty, implement `Lifecycle` and need to be stopped and started before you can use them. All clients have constructor options related to configuration of the underlying WebSocket client.

Testing

Same in Spring MVC

The `spring-test` module provides mock implementations of `ServerHttpRequest`, `ServerHttpResponse`, and `ServerWebExchange`. See Spring Web Reactive for a discussion of mock objects.

`WebTestClient` builds on these mock request and response objects to provide support for testing WebFlux applications without an HTTP server. You can use the `WebTestClient` for end-to-end integration tests, too.

RSocket

This section describes Spring Framework's support for the RSocket protocol.

Overview

RSocket is an application protocol for multiplexed, duplex communication over TCP, WebSocket, and other byte stream transports, using one of the following interaction models:

- `Request-Response` — send one message and receive one back.
- `Request-Stream` — send one message and receive a stream of messages back.
- `Channel` — send streams of messages in both directions.
- `Fire-and-Forget` — send a one-way message.

Once the initial connection is made, the "client" vs "server" distinction is lost as both sides become symmetrical and each side can initiate one of the above interactions. This is why in the protocol calls the participating sides "requester" and "responder" while the above interactions are called "request streams" or simply "requests".

These are the key features and benefits of the RSocket protocol:

- Reactive Streams semantics across network boundary — for streaming requests such as `Request-Stream` and `Channel`, back pressure signals travel between requester and responder, allowing a requester to slow down a responder at the source, hence reducing reliance on network layer congestion control, and the need for buffering at the network level or at any level.
- Request throttling — this feature is named "Leasing" after the `LEASE` frame that can be sent from each end to limit the total number of requests allowed by other end for a given time. Leases are renewed periodically.
- Session resumption — this is designed for loss of connectivity and requires some state to be maintained. The state management is transparent for applications, and works well in combination with back pressure which can stop a producer when possible and reduce the amount of state required.
- Fragmentation and re-assembly of large messages.
- Keepalive (heartbeats).

RSocket has implementations in multiple languages. The Java library is built on Project Reactor, and Reactor Netty for the transport. That means signals from Reactive Streams Publishers in your application propagate transparently through RSocket across the network.

The Protocol

One of the benefits of RSocket is that it has well defined behavior on the wire and an easy to read specification along with some protocol extensions. Therefore it is a good idea to read the spec, independent of language implementations and higher level framework APIs. This section provides

a succinct overview to establish some context.

Connecting

Initially a client connects to a server via some low level streaming transport such as TCP or WebSocket and sends a `SETUP` frame to the server to set parameters for the connection.

The server may reject the `SETUP` frame, but generally after it is sent (for the client) and received (for the server), both sides can begin to make requests, unless `SETUP` indicates use of leasing semantics to limit the number of requests, in which case both sides must wait for a `LEASE` frame from the other end to permit making requests.

Making Requests

Once a connection is established, both sides may initiate a request through one of the frames `REQUEST_RESPONSE`, `REQUEST_STREAM`, `REQUEST_CHANNEL`, or `REQUEST_FNF`. Each of those frames carries one message from the requester to the responder.

The responder may then return `PAYLOAD` frames with response messages, and in the case of `REQUEST_CHANNEL` the requester may also send `PAYLOAD` frames with more request messages.

When a request involves a stream of messages such as as `Request-Stream` and `Channel`, the responder must respect demand signals from the requester. Demand is expressed as a number of messages. Initial demand is specified in `REQUEST_STREAM` and `REQUEST_CHANNEL` frames. Subsequent demand is signaled via `REQUEST_N` frames.

Each side may also send metadata notifications, via the `METADATA_PUSH` frame, that do not pertain to any individual request but rather to the connection as a whole.

Message Format

RSocket messages contain data and metadata. Metadata can be used to send a route, a security token, etc. Data and metadata can be formatted differently. Mime types for each are declared in the `SETUP` frame and apply to all requests on a given connection.

While all messages can have metadata, typically metadata such as a route are per-request and therefore only included in the first message on a request, i.e. with one of the frames `REQUEST_RESPONSE`, `REQUEST_STREAM`, `REQUEST_CHANNEL`, or `REQUEST_FNF`.

Protocol extensions define common metadata formats for use in applications:

- Composite Metadata-- multiple, independently formatted metadata entries.
- Routing — the route for a request.

Java Implementation

The Java implementation for RSocket is built on Project Reactor. The transports for TCP and WebSocket are built on Reactor Netty. As a Reactive Streams library, Reactor simplifies the job of implementing the protocol. For applications it is a natural fit to use `Flux` and `Mono` with declarative operators and transparent back pressure support.

The API in RSocket Java is intentionally minimal and basic. It focuses on protocol features and leaves the application programming model (e.g. RPC codegen vs other) as a higher level, independent concern.

The main contract io.rsocket.RSocket models the four request interaction types with `Mono` representing a promise for a single message, `Flux` a stream of messages, and `io.rsocket.Payload` the actual message with access to data and metadata as byte buffers. The `RSocket` contract is used symmetrically. For requesting, the application is given an `RSocket` to perform requests with. For responding, the application implements `RSocket` to handle requests.

This is not meant to be a thorough introduction. For the most part, Spring applications will not have to use its API directly. However it may be important to see or experiment with RSocket independent of Spring. The RSocket Java repository contains a number of sample apps that demonstrate its API and protocol features.

Spring Support

The `spring-messaging` module contains the following:

- RSocketRequester — fluent API to make requests through an `io.rsocket.RSocket` with data and metadata encoding/decoding.
- Annotated Responders — `@MessageMapping` annotated handler methods for responding.

The `spring-web` module contains `Encoder` and `Decoder` implementations such as Jackson CBOR/JSON, and Protobuf that RSocket applications will likely need. It also contains the `PathPatternParser` that can be plugged in for efficient route matching.

Spring Boot 2.2 supports standing up an RSocket server over TCP or WebSocket, including the option to expose RSocket over WebSocket in a WebFlux server. There is also client support and auto-configuration for an `RSocketRequester.Builder` and `RSocketStrategies`. See the RSocket section in the Spring Boot reference for more details.

Spring Security 5.2 provides RSocket support.

Spring Integration 5.2 provides inbound and outbound gateways to interact with RSocket clients and servers. See the Spring Integration Reference Manual for more details.

Spring Cloud Gateway supports RSocket connections.

RSocketRequester

`RSocketRequester` provides a fluent API to perform RSocket requests, accepting and returning objects for data and metadata instead of low level data buffers. It can be used symmetrically, to make requests from clients and to make requests from servers.

Client Requester

To obtain an `RSocketRequester` on the client side requires connecting to a server along with preparing and sending the initial RSocket `SETUP` frame. `RSocketRequester` provides a builder for that.

Internally uses RSocket Java's `RSocketFactory`.

This is the most basic way to connect with default settings:

Java

```
Mono<RSocketRequester> requesterMono = RSocketRequester.builder()
    .connectTcp("localhost", 7000);

Mono<RSocketRequester> requesterMono = RSocketRequester.builder()
    .connectWebSocket(URI.create("https://example.org:8080/rsocket"));
```

Kotlin

```
import org.springframework.messaging.rsocket.connectTcpAndAwait
import org.springframework.messaging.rsocket.connectWebSocketAndAwait

val requester = RSocketRequester.builder()
        .connectTcpAndAwait("localhost", 7000)

val requester = RSocketRequester.builder()
        .connectWebSocketAndAwait(URI.create("https://example.org:8080/rsocket"))
```

The above is deferred. To actually connect and use the requester:

Java

```
// Connect asynchronously
RSocketRequester.builder().connectTcp("localhost", 7000)
    .subscribe(requester -> {
        // ...
    });

// Or block
RSocketRequester requester = RSocketRequester.builder()
    .connectTcp("localhost", 7000)
    .block(Duration.ofSeconds(5));
```

Kotlin

```
// Connect asynchronously
import org.springframework.messaging.rsocket.connectTcpAndAwait

class MyService {

    private var requester: RSocketRequester? = null

    private suspend fun requester() = requester ?:
        RSocketRequester.builder().connectTcpAndAwait("localhost", 7000).also {
requester = it }

    suspend fun doSomething() = requester().route(...)
}

// Or block
import org.springframework.messaging.rsocket.connectTcpAndAwait

class MyService {

    private val requester = runBlocking {
        RSocketRequester.builder().connectTcpAndAwait("localhost", 7000)
    }

    suspend fun doSomething() = requester.route(...)
}
```

Connection Setup

`RSocketRequester.Builder` provides the following to customize the initial `SETUP` frame:

- `dataMimeType(MimeType)` — set the mime type for data on the connection.
- `metadataMimeType(MimeType)` — set the mime type for metadata on the connection.
- `setupData(Object)` — data to include in the `SETUP`.
- `setupRoute(String, Object…)` — route in the metadata to include in the `SETUP`.
- `setupMetadata(Object, MimeType)` — other metadata to include in the `SETUP`.

For data, the default mime type is derived from the first configured `Decoder`. For metadata, the default mime type is composite metadata which allows multiple metadata value and mime type pairs per request. Typically both don't need to be changed.

Data and metadata in the `SETUP` frame is optional. On the server side, `@ConnectMapping` methods can be used to handle the start of a connection and the content of the `SETUP` frame. Metadata may be used for connection level security.

Strategies

`RSocketRequester.Builder` accepts `RSocketStrategies` to configure the requester. You'll need to use this to provide encoders and decoders for (de)-serialization of data and metadata values. By default only the basic codecs from `spring-core` for `String`, `byte[]`, and `ByteBuffer` are registered. Adding `spring-web` provides access to more that can be registered as follows:

Java

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .encoders(encoders -> encoders.add(new Jackson2CborEncoder))
    .decoders(decoders -> decoders.add(new Jackson2CborDecoder))
    .build();

Mono<RSocketRequester> requesterMono = RSocketRequester.builder()
    .rsocketStrategies(strategies)
    .connectTcp("localhost", 7000);
```

Kotlin

```
import org.springframework.messaging.rsocket.connectTcpAndAwait

val strategies = RSocketStrategies.builder()
        .encoders { it.add(Jackson2CborEncoder()) }
        .decoders { it.add(Jackson2CborDecoder()) }
        .build()

val requester = RSocketRequester.builder()
        .rsocketStrategies(strategies)
        .connectTcpAndAwait("localhost", 7000)
```

`RSocketStrategies` is designed for re-use. In some scenarios, e.g. client and server in the same application, it may be preferable to declare it in Spring configuration.

Client Responders

`RSocketRequester.Builder` can be used to configure responders to requests from the server.

You can use annotated handlers for client-side responding based on the same infrastructure that's used on a server, but registered programmatically as follows:

Java

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .routeMatcher(new PathPatternRouteMatcher())  ①
    .build();

ClientHandler handler = new ClientHandler(); ②

Mono<RSocketRequester> requesterMono = RSocketRequester.builder()
    .rsocketFactory(RSocketMessageHandler.clientResponder(strategies, handler)) ③
    .connectTcp("localhost", 7000);
```

① Use `PathPatternRouteMatcher`, if `spring-web` is present, for efficient route matching.

② Create responder that contains `@MessageMaping` or `@ConnectMapping` methods.

③ Use static factory method in `RSocketMessageHandler` to register one or more responders.

Kotlin

```
import org.springframework.messaging.rsocket.connectTcpAndAwait

val strategies = RSocketStrategies.builder()
        .routeMatcher(PathPatternRouteMatcher())  ①
        .build()

val handler = ClientHandler() ②

val requester = RSocketRequester.builder()
        .rsocketFactory(RSocketMessageHandler.clientResponder(strategies, handler)) ③
        .connectTcpAndAwait("localhost", 7000)
```

① Use `PathPatternRouteMatcher`, if `spring-web` is present, for efficient route matching.

② Create responder that contains `@MessageMaping` or `@ConnectMapping` methods.

③ Use static factory method in `RSocketMessageHandler` to register one or more responders.

Note the above is only a shortcut designed for programmatic registration of client responders. For alternative scenarios, where client responders are in Spring configuration, you can still declare `RSocketMessageHandler` as a Spring bean and then apply as follows:

Java

```
ApplicationContext context = ... ;
RSocketMessageHandler handler = context.getBean(RSocketMessageHandler.class);

Mono<RSocketRequester> requesterMono = RSocketRequester.builder()
    .rsocketFactory(factory -> factory.acceptor(handler.responder()))
    .connectTcp("localhost", 7000);
```

Kotlin

```
import org.springframework.beans.factory.getBean
import org.springframework.messaging.rsocket.connectTcpAndAwait

val context: ApplicationContext = ...
val handler = context.getBean<RSocketMessageHandler>()

val requester = RSocketRequester.builder()
        .rsocketFactory { it.acceptor(handler.responder()) }
        .connectTcpAndAwait("localhost", 7000)
```

For the above you may also need to use `setHandlerPredicate` in `RSocketMessageHandler` to switch to a different strategy for detecting client responders, e.g. based on a custom annotation such as `@RSocketClientResponder` vs the default `@Controller`. This is necessary in scenarios with client and server, or multiple clients in the same application.

See also Annotated Responders, for more on the programming model.

Advanced

`RSocketRequesterBuilder` provides a callback to expose the underlying `ClientRSocketFactory` from RSocket Java for further configuration options for keepalive intervals, session resumption, interceptors, and more. You can configure options at that level as follows:

Java

```
Mono<RSocketRequester> requesterMono = RSocketRequester.builder()
    .rsocketFactory(factory -> {
        // ...
    })
    .connectTcp("localhost", 7000);
```

Kotlin

```
import org.springframework.messaging.rsocket.connectTcpAndAwait

val requester = RSocketRequester.builder()
        .rsocketFactory {
            //...
        }.connectTcpAndAwait("localhost", 7000)
```

Server Requester

To make requests from a server to connected clients is a matter of obtaining the requester for the connected client from the server.

In Annotated Responders, `@ConnectMapping` and `@MessageMapping` methods support an `RSocketRequester` argument. Use it to access the requester for the connection. Keep in mind that

`@ConnectMapping` methods are essentially handlers of the `SETUP` frame which must be handled before requests can begin. Therefore, requests at the very start must be decoupled from handling. For example:

Java

```
@ConnectMapping
Mono<Void> handle(RSocketRequester requester) {
    requester.route("status").data("5")
        .retrieveFlux(StatusReport.class)
        .subscribe(bar -> { ①
            // ...
        });
    return ... ②
}
```

① Start the request asynchronously, independent from handling.

② Perform handling and return completion `Mono<Void>`.

Kotlin

```
@ConnectMapping
suspend fun handle(requester: RSocketRequester) {
    GlobalScope.launch {
        requester.route("status").data("5").retrieveFlow<StatusReport>().collect { ①
            // ...
        }
    }
    /// ... ②
}
```

① Start the request asynchronously, independent from handling.

② Perform handling in the suspending function.

Requests

Once you have a client or server requester, you can make requests as follows:

Java

```
ViewBox box = ... ;

Flux<AirportLocation> locations = requester.route("locate.radars.within") ①
        .data(viewBox) ②
        .retrieveFlux(AirportLocation.class); ③
```

① Specify a route to include in the metadata of the request message.

② Provide data for the request message.

③ Declare the expected response.

Kotlin

```
val box: ViewBox = ...

val locations = requester.route("locate.radars.within") ①
        .data(viewBox) ②
        .retrieveFlow<AirportLocation>() ③
```

① Specify a route to include in the metadata of the request message.

② Provide data for the request message.

③ Declare the expected response.

The interaction type is determined implicitly from the cardinality of the input and output. The above example is a `Request-Stream` because one value is sent and a stream of values is received. For the most part you don't need to think about this as long as the choice of input and output matches an RSocket interaction type and the types of input and output expected by the responder. The only example of an invalid combination is many-to-one.

The `data(Object)` method also accepts any Reactive Streams `Publisher`, including `Flux` and `Mono`, as well as any other producer of value(s) that is registered in the `ReactiveAdapterRegistry`. For a multi-value `Publisher` such as `Flux` which produces the same types of values, consider using one of the overloaded `data` methods to avoid having type checks and `Encoder` lookup on every element:

```
data(Object producer, Class<?> elementClass);
data(Object producer, ParameterizedTypeReference<?> elementTypeRef);
```

The `data(Object)` step is optional. Skip it for requests that don't send data:

Java

```
Mono<AirportLocation> location = requester.route("find.radar.EWR"))
    .retrieveMono(AirportLocation.class);
```

Kotlin

```
import org.springframework.messaging.rsocket.retrieveAndAwait

val location = requester.route("find.radar.EWR")
    .retrieveAndAwait<AirportLocation>()
```

Extra metadata values can be added if using composite metadata (the default) and if the values are supported by a registered `Encoder`. For example:

Java

```
String securityToken = ... ;
ViewBox viewBox = ... ;
MimeType mimeType = MimeType.valueOf("message/x.rsocket.authentication.bearer.v0");

Flux<AirportLocation> locations = requester.route("locate.radars.within")
        .metadata(securityToken, mimeType)
        .data(viewBox)
        .retrieveFlux(AirportLocation.class);
```

Kotlin

```
import org.springframework.messaging.rsocket.retrieveFlow

val requester: RSocketRequester = ...

val securityToken: String = ...
val viewBox: ViewBox = ...
val mimeType = MimeType.valueOf("message/x.rsocket.authentication.bearer.v0")

val locations = requester.route("locate.radars.within")
        .metadata(securityToken, mimeType)
        .data(viewBox)
        .retrieveFlow<AirportLocation>()
```

For `Fire-and-Forget` use the `send()` method that returns `Mono<Void>`. Note that the `Mono` indicates only that the message was successfully sent, and not that it was handled.

Annotated Responders

RSocket responders can be implemented as `@MessageMapping` and `@ConnectMapping` methods. `@MessageMapping` methods handle individual requests, and `@ConnectMapping` methods handle connection-level events (setup and metadata push). Annotated responders are supported symmetrically, for responding from the server side and for responding from the client side.

Server Responders

To use annotated responders on the server side, add `RSocketMessageHandler` to your Spring configuration to detect `@Controller` beans with `@MessageMapping` and `@ConnectMapping` methods:

Java

```
@Configuration
static class ServerConfig {

    @Bean
    public RSocketMessageHandler rsocketMessageHandler() {
        RSocketMessageHandler handler = new RSocketMessageHandler();
        handler.routeMatcher(new PathPatternRouteMatcher());
        return handler;
    }
}
```

Kotlin

```
@Configuration
class ServerConfig {

    @Bean
    fun rsocketMessageHandler() = RSocketMessageHandler().apply {
        routeMatcher = PathPatternRouteMatcher()
    }
}
```

Then start an RSocket server through the Java RSocket API and plug the `RSocketMessageHandler` for the responder as follows:

Java

```
ApplicationContext context = ... ;
RSocketMessageHandler handler = context.getBean(RSocketMessageHandler.class);

CloseableChannel server =
    RSocketFactory.receive()
        .acceptor(handler.responder())
        .transport(TcpServerTransport.create("localhost", 7000))
        .start()
        .block();
```

Kotlin

```
import org.springframework.beans.factory.getBean

val context: ApplicationContext = ...
val handler = context.getBean<RSocketMessageHandler>()

val server = RSocketFactory.receive()
        .acceptor(handler.responder())
        .transport(TcpServerTransport.create("localhost", 7000))
        .start().awaitFirst()
```

`RSocketMessageHandler` supports composite and routing metadata by default. You can set its `MetadataExtractor` if you need to switch to a different mime type or register additional metadata mime types.

You'll need to set the `Encoder` and `Decoder` instances required for metadata and data formats to support. You'll likely need the `spring-web` module for codec implementations.

By default `SimpleRouteMatcher` is used for matching routes via `AntPathMatcher`. We recommend plugging in the `PathPatternRouteMatcher` from `spring-web` for efficient route matching. RSocket routes can be hierarchical but are not URL paths. Both route matchers are configured to use "." as separator by default and there is no URL decoding as with HTTP URLs.

`RSocketMessageHandler` can be configured via `RSocketStrategies` which may be useful if you need to share configuration between a client and a server in the same process:

Java

```
@Configuration
static class ServerConfig {

    @Bean
    public RSocketMessageHandler rsocketMessageHandler() {
        RSocketMessageHandler handler = new RSocketMessageHandler();
        handler.setRSocketStrategies(rsocketStrategies());
        return handler;
    }

    @Bean
    public RSocketStrategies rsocketStrategies() {
        return RSocketStrategies.builder()
            .encoders(encoders -> encoders.add(new Jackson2CborEncoder))
            .decoders(decoders -> decoders.add(new Jackson2CborDecoder))
            .routeMatcher(new PathPatternRouteMatcher())
            .build();
    }
}
```

Kotlin

```
@Configuration
class ServerConfig {

    @Bean
    fun rsocketMessageHandler() = RSocketMessageHandler().apply {
        rSocketStrategies = rsocketStrategies()
    }

    @Bean
    fun rsocketStrategies() = RSocketStrategies.builder()
            .encoders { it.add(Jackson2CborEncoder()) }
            .decoders { it.add(Jackson2CborDecoder()) }
            .routeMatcher(PathPatternRouteMatcher())
            .build()
}
```

Client Responders

Annotated responders on the client side need to be configured in the `RSocketRequester.Builder`. For details, see Client Responders.

@MessageMapping

Once server or client responder configuration is in place, `@MessageMapping` methods can be used as follows:

Java

```
@Controller
public class RadarsController {

    @MessageMapping("locate.radars.within")
    public Flux<AirportLocation> radars(MapRequest request) {
        // ...
    }
}
```

Kotlin

```
@Controller
class RadarsController {

    @MessageMapping("locate.radars.within")
    fun radars(request: MapRequest): Flow<AirportLocation> {
        // ...
    }
}
```

You don't need to explicit specify the RSocket interaction type. Simply declare the expected input and output, and a route pattern. The supporting infrastructure will adapt matching requests.

The following additional arguments are supported for `@MessageMapping` methods:

- `RSocketRequester` — the requester for the connection associated with the request, to make requests to the remote end.
- `@DestinationVariable` — the value for a variable from the pattern, e.g. `@MessageMapping("find.radar.{id}")`.
- `@Header` — access to a metadata value registered for extraction, as described in MetadataExtractor.
- `@Headers Map<String, Object>` — access to all metadata values registered for extraction, as described in MetadataExtractor.

@ConnectMapping

`@ConnectMapping` handles the `SETUP` frame at the start of an RSocket connection, and any subsequent metadata push notifications through the `METADATA_PUSH` frame, i.e. `metadataPush(Payload)` in `io.rsocket.RSocket`.

`@ConnectMapping` methods support the same arguments as @MessageMapping but based on metadata and data from the `SETUP` and `METADATA_PUSH` frames. `@ConnectMapping` can have a pattern to narrow handling to specific connections that have a route in the metadata, or if no patterns are declared then all connections match.

`@ConnectMapping` methods cannot return data and must be declared with `void` or `Mono<Void>` as the return value. If handling returns an error for a new connection then the connection is rejected. Handling must not be held up to make requests to the `RSocketRequester` for the connection. See Server Requester for details.

MetadataExtractor

Responders must interpret metadata. Composite metadata allows independently formatted metadata values (e.g. for routing, security, tracing) each with its own mime type. Applications need a way to configure metadata mime types to support, and a way to access extracted values.

`MetadataExtractor` is a contract to take serialized metadata and return decoded name-value pairs that can then be accessed like headers by name, for example via `@Header` in annotated handler methods.

`DefaultMetadataExtractor` can be given `Decoder` instances to decode metadata. Out of the box it has built-in support for "message/x.rsocket.routing.v0" which it decodes to `String` and saves under the "route" key. For any other mime type you'll need to provide a `Decoder` and register the mime type as follows:

Java

```
DefaultMetadataExtractor extractor = new DefaultMetadataExtractor(metadataDecoders);
extractor.metadataToExtract(fooMimeType, Foo.class, "foo");
```

Kotlin

```
import org.springframework.messaging.rsocket.metadataToExtract

val extractor = DefaultMetadataExtractor(metadataDecoders)
extractor.metadataToExtract<Foo>(fooMimeType, "foo")
```

Composite metadata works well to combine independent metadata values. However the requester might not support composite metadata, or may choose not to use it. For this, `DefaultMetadataExtractor` may needs custom logic to map the decoded value to the output map. Here is an example where JSON is used for metadata:

Java

```
DefaultMetadataExtractor extractor = new DefaultMetadataExtractor(metadataDecoders);
extractor.metadataToExtract(
    MimeType.valueOf("application/vnd.myapp.metadata+json"),
    new ParameterizedTypeReference<Map<String,String>>() {},
    (jsonMap, outputMap) -> {
        outputMap.putAll(jsonMap);
    });
```

Kotlin

```
import org.springframework.messaging.rsocket.metadataToExtract

val extractor = DefaultMetadataExtractor(metadataDecoders)
extractor.metadataToExtract<Map<String,
String>>(MimeType.valueOf("application/vnd.myapp.metadata+json")) { jsonMap, outputMap
->
    outputMap.putAll(jsonMap)
}
```

When configuring `MetadataExtractor` through `RSocketStrategies`, you can let `RSocketStrategies.Builder` create the extractor with the configured decoders, and simply use a callback to customize registrations as follows:

Java

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .metadataExtractorRegistry(registry -> {
        registry.metadataToExtract(fooMimeType, Foo.class, "foo");
        // ...
    })
    .build();
```

Kotlin

```
import org.springframework.messaging.rsocket.metadataToExtract

val strategies = RSocketStrategies.builder()
        .metadataExtractorRegistry { registry: MetadataExtractorRegistry ->
            registry.metadataToExtract<Foo>(fooMimeType, "foo")
            // ...
        }
        .build()
```

Reactive Libraries

`spring-webflux` depends on `reactor-core` and uses it internally to compose asynchronous logic and to provide Reactive Streams support. Generally, WebFlux APIs return `Flux` or `Mono` (since those are used internally) and leniently accept any Reactive Streams `Publisher` implementation as input. The use of `Flux` versus `Mono` is important, because it helps to express cardinality — for example, whether a single or multiple asynchronous values are expected, and that can be essential for making decisions (for example, when encoding or decoding HTTP messages).

For annotated controllers, WebFlux transparently adapts to the reactive library chosen by the application. This is done with the help of the `ReactiveAdapterRegistry`, which provides pluggable support for reactive library and other asynchronous types. The registry has built-in support for RxJava and `CompletableFuture`, but you can register others, too.

For functional APIs (such as Functional Endpoints, the `WebClient`, and others), the general rules for WebFlux APIs apply — `Flux` and `Mono` as return values and a Reactive Streams `Publisher` as input. When a `Publisher`, whether custom or from another reactive library, is provided, it can be treated only as a stream with unknown semantics (0..N). If, however, the semantics are known, you can wrap it with `Flux` or `Mono.from(Publisher)` instead of passing 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 (for example, `application/json+stream`), values are written and flushed individually. Otherwise, values are buffered into a list and rendered as a JSON array.