Web on Reactive Stack

Version 5.0.3.RELEASE

Table of Contents

This part of the documentation covers support for reactive stack, web applications built on a Reactive Streams API to run on 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, please see Web on Servlet Stack.

Chapter 1. Spring WebFlux

1.1. Introduction

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

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

1.1.1. Why a new web framework?

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

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

1.1.2. Reactive: what and why?

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

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

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

Reactive Streams is a small spec, also adopted in Java 9, that defines the interaction between asynchronous components with back pressure. For example a data repository — acting as Publisher, can produce data that an HTTP server — acting as Subscriber, can then write to the

response. The main purpose of Reactive Streams is to allow the subscriber to control how fast or how slow the publisher will produce data.

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

1.1.3. Reactive API

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

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

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

1.1.4. Programming models

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

1.1.5. Choosing a web framework

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

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

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

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

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

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

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

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

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

1.1.6. Choosing a server

Spring WebFlux is supported on Netty, Undertow, Tomcat, Jetty, and Servlet 3.1+ containers. Each server is adapted to a common Reactive Streams API. The Spring WebFlux programming models are built on that common API.

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

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

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

1.1.7. Performance vs scale

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

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

1.2. Reactive Spring Web

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

Server support is organized in two layers:

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

1.2.1. HttpHandler

Every HTTP server has some API for HTTP request handling. HttpHandler is a simple contract with one method to handle a request and response. It is intentionally minimal. Its main purpose is to

provide a common, Reactive Streams based API for HTTP request handling over different servers.

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

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

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

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

Reactor Netty:

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

Undertow:

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

Tomcat:

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

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

Jetty:

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

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

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

To deploy as a WAR to a Servlet 3.1+ container, wrap `HttpHandler` with `ServletHttpHandlerAdapter` and register that as a `Servlet`. This can be automated through the use of AbstractReactiveWebInitializer.

1.2.2. WebHandler API

`HttpHandler` is the lowest level contract for running on different HTTP servers. On top of that foundation, the WebHandler API provides a slightly higher level, but still general purpose, set of components that form a chain of WebExceptionHandler's, WebFilter's, and a WebHandler.

All WebHandler API components take `ServerWebExchange` as input which goes beyond `ServerHttpRequest` and `ServerHttpResponse` to provide extra building blocks for use in web applications such as request attributes, session attributes, access to parsed form data, multipart data, and more.

`WebHttpHandlerBuilder` is used to assemble a request processing chain. You can use methods on the builder to add components manually, or more likely have them detected from a Spring `ApplicationContext`, with the resulting `HttpHandler` ready to run via a server adapter:

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

The table below lists the components that `WebHttpHandlerBuilder` detects:

Bean name	Bean type	Count	Description
<any>	`WebExceptionHandler`	0..N	Exception handlers to apply after all `WebFilter`'s and the target `WebHandler`.
<any>	`WebFilter`	0..N	Filters to invoke before and after the target `WebHandler`.
"webHandler"	`WebHandler`	1	The handler for the request.
"webSessionManager"	`WebSessionManager`	0..1	The manager for `WebSession`'s exposed through a method on `ServerWebExchange`. `DefaultWebSessionManager` by default.
"serverCodecConfigurer"	`ServerCodecConfigurer`	0..1	For access to `HttpMessageReader`'s for parsing form data and multipart data that's 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.

Form Reader

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

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

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 via the `ServerCodecConfigurer` bean (see Web Handler API).

Multipart Reader

Same in Spring MVC

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

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

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 3rd party library supported, and the only library we know for non-blocking parsing of multipart requests. It is enabled through the `ServerCodecConfigurer` bean (see Web Handler API).

To parse multipart data in streaming fashion, 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 `@RequestBody` can be used to decode the content to `Flux<Part>` without collecting to a `MultiValueMap`.

1.2.3. HTTP Message Codecs

Same in Spring MVC

The `spring-web` module defines the HttpMessageReader and HttpMessageWriter contracts for encoding and decoding the body of HTTP requests and responses via Rective Streams `Publisher`'s. These contacts are used on the client side, e.g. in the `WebClient`, and on the server side, e.g. in annotated controllers and functional endpoints.

The `spring-core` module defines the Encoder and Decoder contracts that are independent of HTTP and rely on the DataBuffer contract that abstracts different byte buffer representations such as the Netty `ByteBuf` and `java.nio.ByteBuffer` (see Data Buffers and Codecs). An `Encoder` can be wrapped with `EncoderHttpMessageWriter` to be used as an `HttpMessageWriter` while a `Decoder` can be wrapped with `DecoderHttpMessageReader` to be used as an `HttpMessageReader`.

The `spring-core` module contains basic `Encoder` and `Decoder` implementations for `byte[]`, `ByteBuffer`, `DataBuffer`, `Resource`, and `String`. The `spring-web` module adds `Encoder`'s and `Decoder`'s for Jackson JSON, Jackson Smile, and JAXB2. The `spring-web` module also contains some web-specific readers and writers for server-sent events, form data, and multipart requests.

To configure or customize the readers and writers to use applications will typically use `ClientCodecConfigurer` or `ServerCodecConfigurer`.

Jackson JSON

The decoder relies on Jackson's non-blocking, byte array parser to parse a stream of byte chunks into a `TokenBuffer` stream, which can then be turned into Objects with Jackson's `ObjectMapper`.

The encoder processes a `Publisher<?>` as follows:

- if the `Publisher` is a `Mono` (i.e. single value), the value is encoded to JSON.
- if media type is `application/stream+json`, each value produced by the `Publisher` is encoded individually to JSON followed by a new line.
- otherwise all items from the `Publisher` are gathered in with `Flux#collectToList()` and the resulting collection is encoded as a JSON array.

As a special case to the above rules the `ServerSentEventHttpMessageWriter` feeds items emitted from its input `Publisher` individually into the `Jackson2JsonEncoder` as a `Mono<?>`.

Note that both the Jackson JSON encoder and decoder explicitly back out of rendering elements of type `String`. Instead `String`'s are treated as low level content, (i.e. serialized JSON) and are rendered as-is by the `CharSequenceEncoder`. If you want a `Flux<String>` rendered as a JSON array, you'll have to use `Flux#collectToList()` and provide a `Mono<List<String>>` instead.

1.3. DispatcherHandler

Same in Spring MVC

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

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

Spring configuration in a WebFlux application typically contains:

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

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

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

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

1.3.1. Special bean types

Same in Spring MVC

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

Bean type	Explanation
HandlerMapping	Map a request to a handler. The mapping is based on some criteria the details of which vary by `HandlerMapping` implementation — annotated controllers, simple URL pattern mappings, etc. The main `HandlerMapping` implementations are `RequestMappingHandlerMapping` based on `@RequestMapping` annotated methods, `RouterFunctionMapping` based on functional endpoint routes, and `SimpleUrlHandlerMapping` based on explicit registrations of URI path patterns to handlers.
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. The built-in `HandlerResultHandler` implementations are `ResponseEntityResultHandler` supporting `ResponseEntity` return values, `ResponseBodyResultHandler` supporting `@ResponseBody` methods, `ServerResponseResultHandler` supporting the `ServerResponse` returned from functional endpoints, and `ViewResolutionResultHandler` supporting rendering with a view and a model.

1.3.2. Framework Config

Same in Spring MVC

The `DispatcherHandler` detects the special beans it needs in the `ApplicationContext`. Applications can declare the special beans they wish to have. However most applications will find a better starting point in the WebFlux Java config which provide a higher level configuration API that in turn make the necessary bean declarations. See WebFlux Config for more details.

1.3.3. Processing

Same in Spring MVC

The `DispatcherHandler` processes requests as follows:

- Each `HandlerMapping` is asked to find a matching handler and the first match is used.
- If a handler is found, it is executed through an appropriate `HandlerAdapter` which exposes the return value from the execution as `HandlerResult`.
- The `HandlerResult` is given to an appropriate `HandlerResultHandler` to complete processing by writing to the response directly or using a view to render.

1.4. Annotated Controllers

Same in Spring MVC

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

Here is a basic example:

```
@RestController
public class HelloController {

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

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

1.4.1. @Controller

Same in Spring MVC

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

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

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

    // ...
}
```

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

1.4.2. Request Mapping

Same in Spring MVC

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

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

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

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

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

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

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

URI Patterns

Same in Spring MVC

You can map requests using glob patterns and wildcards:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Pattern Comparison

Same in Spring MVC

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

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

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

Consumable Media Types

Same in Spring MVC

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

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

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

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

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

Producible Media Types

Same in Spring MVC

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

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

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

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

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

Parameters and Headers

Same in Spring MVC

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

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

You can also use the same with request header conditions:

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

HTTP HEAD, OPTIONS

Same in Spring MVC

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

By default HTTP OPTIONS is handled by setting the "Allow" response header to the list of HTTP

methods listed in all `@RequestMapping` methods with matching URL patterns.

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

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

1.4.3. Handler methods

Same in Spring MVC

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

Method arguments

Same in Spring MVC

The table below shows supported controller method arguments.

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

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

Controller method argument	Description
`ServerWebExchange`	Access to the full `ServerWebExchange` — container for the HTTP request and response, request and session attributes, `checkNotModified` methods, and others.
`ServerHttpRequest`, `ServerHttpResponse`	Access to the HTTP request or response.
`WebSession`	Access to the session; this does not force the start of a new session unless attributes are added. Supports reactive types.
`java.security.Principal`	Currently authenticated user; possibly a specific `Principal` implementation class if known. Supports reactive types.
`org.springframework.http.HttpMethod`	The HTTP method of the request.
`java.util.Locale`	The current request locale, determined by the most specific `LocaleResolver` available, in effect, the configured `LocaleResolver`/`LocaleContextResolver`.
Java 6+: `java.util.TimeZone` Java 8+: `java.time.ZoneId`	The time zone associated with the current request, as determined by a `LocaleContextResolver`.
`@PathVariable`	For access to URI template variables. See URI Patterns.

Controller method argument	Description
`@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, e.g. to set its attributes. See "Any other argument" further below 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. Cookies 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 using `HttpMessageReader`'s. Supports reactive types. @RequestBody.
`HttpEntity<B>`	For access to request headers and body. The body is converted with `HttpMessageReader`'s. Supports reactive types. See HttpEntity.
`@RequestPart`	For access to a part in a "multipart/form-data" request. Supports reactive types. See Multipart and Multipart Reader.
`java.util.Map`, `org.springframework.ui.Model`, `org.springframework.ui.ModelMap`	For access to the model that is used in HTML controllers and exposed to templates as part of view rendering.
`@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 Methods and Binder Methods. Note that use of `@ModelAttribute` is optional, e.g. to set its attributes. See "Any other argument" further below in this table.
`Errors`, `BindingResult`	For access to errors from validation and data binding for a command object (i.e. `@ModelAttribute` argument), or errors from the validation of an `@RequestBody` or `@RequestPart` arguments; 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, context path, and the literal part of the servlet mapping also taking into account `Forwarded` and `X-Forwarded-*` headers.
`@SessionAttribute`	For access to any session attribute; in contrast to model attributes stored in the session as a result of a class-level `@SessionAttributes` declaration. See @SessionAttribute for more details.
`@RequestAttribute`	For access to request attributes. See @RequestAttribute for more details.

Controller method argument	Description
Any other argument	If a method argument is not matched to any of the above, by default it is resolved as an `@RequestParam` if it is a simple type, as determined by BeanUtils#isSimpleProperty, or as an `@ModelAttribute` otherwise.

Return values

Same in Spring MVC

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

Controller method return value	Description
`@ResponseBody`	The return value is encoded through `HttpMessageWriter`s and written to the response. See @ResponseBody.
`HttpEntity<B>`, `ResponseEntity<B>`	The return value specifies the full response including HTTP headers and body be encoded through `HttpMessageWriter`s and written to the response. See ResponseEntity.
`HttpHeaders`	For returning a response with headers and no body.
`String`	A view name to be resolved with `ViewResolver`'s and used together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method may also programmatically enrich the model by declaring a `Model` argument (see above).
`View`	A `View` instance to use for rendering together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method may also programmatically enrich the model by declaring a `Model` argument (see above).
`java.util.Map`, `org.springframework.ui.Model`	Attributes to be added to the implicit model with the view name implicitly determined 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" further below in this table.
`Rendering`	An API for model and view rendering scenarios.

Controller method return value	Description
`void`	A method with a `void`, possibly async (e.g. `Mono<Void>`), return type (or a `null` return value) is considered to have fully handled the response if it also has a `ServerHttpResponse`, or a `ServerWebExchange` argument, or an `@ResponseStatus` annotation. The same is true also if the controller has made a positive ETag or lastModified timestamp check. If none of the above is true, a `void` return type may 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 `SeverSentEvent` wrapper can be omitted when only data needs to be written (however `text/event-stream` must be requested or declared in the mapping through the produces attribute).
Any other return value	If a return value is not matched to any of the above, by default it is 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 in Spring MVC

Some annotated controller method arguments that represent String-based request input — e.g. `@RequestParam`, `@RequestHeader`, `@PathVariable`, `@MatrixVariable`, and `@CookieValue`, may 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`, etc. 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 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, each variable separated by semicolon and multiple values separated by comma, e.g. `"/cars;color=red,green;year=2012"`. Multiple values can also be specified through repeated variable names, e.g. `"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're 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. Below is an example:

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

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

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

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

```
// 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
}
```

A matrix variable may be defined as optional and a default value specified:

```
// GET /pets/42

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

    // q == 1
}
```

To get all matrix variables, use a `MultiValueMap`:

```
// 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]
}
```

@RequestParam

Same in Spring MVC

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

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

    // ...

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

    // ...

}
```

Unlike the Servlet API "request paramater" concept that conflate query parameters, form data, and multiparts into one, in WebFlux each is accessed individually through the `ServerWebExchange`. While `@RequestParam` binds to query parameters only, you can use data binding to apply query paramerters, form data, and multiparts to a command object.

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

Type conversion is applied automatically if the target method parameter type is not `String`. See

[mvc-ann-typeconversion].

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

Note that use of `@RequestParam` is optional, e.g. 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 was annotated with `@RequestParam`.

@RequestHeader

Same in Spring MVC

Use the `@RequestHeader` annotation to bind a request header to a method argument in a controller.

Given 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 gets the value of the `Accept-Encoding` and `Keep-Alive` headers:

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

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

When an `@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/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 `String[]` or `List<String>`.

@CookieValue

Same in Spring MVC

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

Given request with the following cookie:

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

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

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

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

@ModelAttribute

Same in Spring MVC

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 overlaid with 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. For example:

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

The `Pet` instance above is resolved as follows:

- From the model if already added via Model Methods.
- From the HTTP session via @SessionAttributes.
- From the invocation of a default constructor.
- From the invocation of a "primary constructor" with arguments matching to query parameters or form fields; argument names are determined via JavaBeans `@ConstructorProperties` or via 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 Binder Methods.

Data binding may result in errors. By default a `WebExchangeBindException` is raised but to check for such errors in the controller method, add a `BindingResult` argument immediately next to the

`@ModelAttribute` as shown below:

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

Validation can be applied automatically after data binding by adding the `javax.validation.Valid` annotation or Spring's `@Validated` annotation (also see Bean validation and Spring validation). For example:

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

Spring WebFlux, unlike Spring MVC, supports reactive types in the model, e.g. `Mono<Account>` or `io.reactivex.Single<Account>`. An `@ModelAttribute` argument can be declared with or without a reactive type wrapper, and it will be resolved accordingly, to the actual value if necessary. Note however that in order 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:

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

Note that use of `@ModelAttribute` is optional, e.g. 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 was annotated with `@ModelAttribute`.

@SessionAttributes

Same 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 will typically list the names of model attributes or types of model attributes which should be transparently stored in the session for subsequent requests to access.

For example:

```
@Controller
<strong>@SessionAttributes("pet")</strong>
public class EditPetForm {
    // ...
}
```

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:

```
@Controller
<strong>@SessionAttributes("pet")</strong>
public class EditPetForm {

    // ...

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

@SessionAttribute

Same in Spring MVC

If you need access to pre-existing session attributes that are managed globally, i.e. outside the controller (e.g. by a filter), and may or may not be present use the `@SessionAttribute` annotation on a method parameter:

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

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 in Spring MVC

Similar to `@SessionAttribute` the `@RequestAttribute` annotation can be used to access pre-existing request attributes created earlier, e.g. by a `WebFilter`:

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

Multipart

Same in Spring MVC

As explained in Multipart Reader, `ServerWebExchange` provides access to multipart content. The best way to handle a file upload form (e.g. from a browser) in a controller is through data binding to a command object:

```
class MyForm {

    private String name;

    private MultipartFile file;

    // ...

}

@Controller
public class FileUploadController {

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

}
```

Multipart requests can also be submitted from non-browser clients in a RESTful service scenario. For example 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 the "meta-data" part with `@RequestPart` which would deserialize it from JSON (similar to `@RequestBody`) through one of the configured HTTP Message Codecs:

```
@PostMapping("/")
public String handle(<strong>@RequestPart("meta-data") MetaData metadata,
        @RequestPart("file-data") FilePart file</strong>) {
    // ...
}
```

To access multipart data sequentially, in streaming fashion, use `@RequestBody` with `Flux<Part>` instead. For example:

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

`@RequestPart` can be used 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 validation errors can be handled locally within the controller through an `Errors` or `BindingResult` argument:

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

@RequestBody

Same in Spring MVC

Use the `@RequestBody` annotation to have the request body read and deserialized into an Object through an HttpMessageReader. Below is an example with an `@RequestBody` argument:

```
@PostMapping("/accounts")
public void 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:

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

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

`@RequestBody` can be used 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 validation errors can be handled locally within the controller through an `Errors` or `BindingResult` argument:

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

HttpEntity

Same in Spring MVC

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

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

@ResponseBody

Same in Spring MVC

Use the `@ResponseBody` annotation on a method to have the return serialized to the response body through an HttpMessageWriter. For example:

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

`@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 on JSON rendering see Jackson JSON.

`@ResponseBody` methods can be combined with JSON serialization views. See [mvc-ann-jackson] for details.

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

ResponseEntity

Same in Spring MVC

`ResponseEntity` is more or less identical to using @ResponseBody but based on a container object that specifies request headers and body. Below is an example:

```
@PostMapping("/something")
public ResponseEntity<String> handle() {
    // ...
    URI location = ...
    return new ResponseEntity.created(location).build();
}
```

Jackson JSON

Jackson serialization views

Same 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, use Jackson's `@JsonView` annotation to activate a serialization view class:

```
@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;
    }
}
```

`@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.

1.4.4. Model Methods

Same in Spring MVC

The `@ModelAttribute` annotation can be used on `@RequestMapping` method arguments to create or access an Object from the model and bind it to the request. `@ModelAttribute` can also be used as a method-level annotation on controller methods whose purpose is not to handle requests but to add commonly needed model attributes prior to request handling.

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 via `@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 nor anything related to the request body.

An example `@ModelAttribute` method:

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

To add one attribute only:

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

When a name is not explicitly specified, a default name is chosen based on the Object 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, e.g. `Mono<Account>` or `io.reactivex.Single<Account>`. Such asynchronous model attributes may be transparently resolved (and the model updated) to their actual values at the time of `@RequestMapping` invocation, providing a `@ModelAttribute` argument is declared without a wrapper, for example:

```
@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) {
    // ...
}
```

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.

`@ModelAttribute` can also be used 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` which would otherwise be interpreted as a view name. `@ModelAttribute` can also help to customize the model attribute name:

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

1.4.5. Binder Methods

Same in Spring MVC

`@InitBinder` methods in an `@Controller` or `@ControllerAdvice` class can be used to customize type conversion for method arguments that represent String-based request values (e.g. request parameters, path variables, headers, cookies, and others). Type conversion also applies during data binding of request parameters onto `@ModelAttribute` arguments (i.e. command objects).

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

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

```
@Controller
public class FormController {

    <strong>@InitBinder</strong>
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new 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`'s:

```
@Controller
public class FormController {

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

    // ...
}
```

1.4.6. Controller Advice

Same in Spring MVC

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

`@ControllerAdvice` is marked with `@Component` which means such classes can be registered as Spring beans via component scanning. `@RestControllerAdvice` is also a meta-annotation marked with both `@ControllerAdvice` and `@ResponseBody` which essentially means `@ExceptionHandler` methods are rendered to the response body via message conversion (vs view resolution/template rendering).

On startup, the infrastructure classes for `@RequestMapping` and `@ExceptionHandler` methods detect Spring beans of type `@ControllerAdvice`, and then apply their methods at runtime. Global `@ExceptionHandler` methods (from an `@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, i.e. all controllers, but you can narrow that down to a subset of controllers via attributes on the annotation:

```
// 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 {}
```

Keep in mind the above selectors are evaluated at runtime and may negatively impact performance

if used extensively. See the @ControllerAdvice Javadoc for more details.

1.5. Functional Endpoints

Spring WebFlux includes 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 annotated-based programming model but otherwise running on the same Reactive Spring Web foundation

1.5.1. HandlerFunction

Incoming HTTP requests are handled by a `HandlerFunction`, which is essentially a function that takes a `ServerRequest` and returns a `Mono<ServerResponse>`. If you're familiar with the annotation-based programming model, a handler function is the equivalent of an `@RequestMapping` method.

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

`ServerRequest` gives access to various HTTP request elements: the method, URI, query parameters, and headers (via a separate `ServerRequest.Headers` interface. Access to the body is provided through the `body` methods. For instance, this is how to extract the request body into a `Mono<String>`:

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

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

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

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

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

Similarly, `ServerResponse` provides access to the HTTP response. Since it is immutable, you create a `ServerResponse` with a builder. The builder allows you to set the response status, add response headers, and provide a body. For instance, this is how to create a response with a 200 OK status, a JSON content-type, and a body:

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

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

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

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

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

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

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

public class PersonHandler {

    private final PersonRepository repository;

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

    public Mono<ServerResponse> listPeople(ServerRequest request) { ①
        Flux<Person> people = repository.allPeople();
        return ServerResponse.ok().contentType(APPLICATION_JSON).body(people, Person
.class);
    }

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

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

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

② `createPerson` is a handler function that stores a new `Person` contained in the request body. Note that `PersonRepository.savePerson(Person)` returns `Mono<Void>`: an empty Mono that emits a completion signal when the person has been read from the request and stored. So we use the `build(Publisher<Void>)` method to send a response when that completion signal is received, i.e. when the `Person` has been saved.

③ `getPerson` is a handler function that returns a single person, identified via the path variable `id`. We retrieve that `Person` via the repository, and create a JSON response if it is found. If it is not found, we use `switchIfEmpty(Mono<T>)` to return a 404 Not Found response.

1.5.2. RouterFunction

Incoming requests are routed to handler functions with a **`RouterFunction`**, which is a function that takes a `ServerRequest`, and returns a `Mono<HandlerFunction>`. If a request matches a particular route, a handler function is returned, or otherwise an empty `Mono` is returned. `RouterFunction` has a similar

purpose as the `@RequestMapping` annotation in the annotation-based programming model.

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

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

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

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

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

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

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

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

1.5.3. Running a server

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

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

The returned `HttpHandler` can then be used with a number of servers adapters by following HttpHandler for server-specific instructions.

A more advanced option is to run with a DispatcherHandler-based setup through the WebFlux Config which uses Spring configuration to declare the components quired to process requests. The WebFlux Java config declares the following infrastructure components to support functional endpoints:

- `RouterFunctionMapping` — detects one or more `RouterFunction<?>` beans in the Spring configuration, combines them via `RouterFunction.andOther`, and routes requests to the resulting composed `RouterFunction`.
- `HandlerFunctionAdapter` — simple adapter that allows the `DispatcherHandler` to 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 above components allow functional endpoints to 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 the Spring Boot WebFlux starter.

Below is example WebFlux Java config (see DispatcherHandler for how to run):

```
@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
    default void addCorsMappings(CorsRegistry registry) {
        // configure CORS...
    }

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

1.5.4. HandlerFilterFunction

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

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

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

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

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

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

1.6. CORS

Same in Spring MVC

1.6.1. Introduction

Same 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, e.g. withdrawing money from your account!

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

1.6.2. Processing

Same 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 refer to the specification for more details.

Spring WebFlux `HandlerMapping`'s provide built-in support for CORS. After successfully mapping a request to a handler, `HandlerMapping`'s check the CORS configuration for the given request and handler and take further actions. Preflight requests are handled directly while simple and actual

CORS requests are intercepted, validated, and have required CORS response headers set.

In order to enable cross-origin requests (i.e. 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 will use the WebFlux Java config to declare such mappings, which results in a single, global map passed to all `HadlerMappping`'s.

Global CORS configuration at the `HandlerMapping` level can be combined 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 — e.g. 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 make advanced customizations, check:

- `CorsConfiguration`
- `CorsProcessor`, `DefaultCorsProcessor`
- `AbstractHandlerMapping`

1.6.3. @CrossOrigin

Same in Spring MVC

The `@CrossOrigin` annotation enables cross-origin requests on annotated controller methods:

```
@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) {
        // ...
    }
}
```

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 only be used where appropriate.
- `maxAge` is set to 30 minutes.

`@CrossOrigin` is supported at the class level too and inherited by all methods:

```
@CrossOrigin(origins = "http://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) {
        // ...
    }
}
```

`CrossOrigin` can be used at both class and method-level:

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

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

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

1.6.4. Global Config

Same in Spring MVC

In addition to fine-grained, controller method level configuration you'll probably want to define some global CORS configuration too. You can set URL-based `CorsConfiguration` mappings individually on any `HandlerMapping`. Most applications however will use the WebFlux Java config 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 only be used where appropriate.
- `maxAge` is set to 30 minutes.

To enable CORS in the WebFlux Java config, use the `CorsRegistry` callback:

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

    @Override
    public void addCorsMappings(CorsRegistry registry) {

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

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

1.6.5. CORS WebFilter

Same in Spring MVC

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

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

```
@Bean
CorsWebFilter corsFilter() {

    CorsConfiguration config = new CorsConfiguration();

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

    config.setAllowCredentials(true);
    config.addAllowedOrigin("http://domain1.com");
    config.addAllowedHeader("<strong>");
    config.addAllowedMethod("</strong>");

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

    return new CorsWebFilter(source);
}
```

1.7. Web Security

Same in Spring MVC

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

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

1.8. WebFlux Config

Same in Spring MVC

The WebFlux Java config declares components 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 config but, if you want to, it's very easy to see them in `WebFluxConfigurationSupport` or read more what they are in Special bean types.

For more advanced customizations, not available in the configuration API, it is also possible to gain full control over the configuration through the Advanced config mode.

1.8.1. Enable WebFlux config

Same in Spring MVC

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

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

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

1.8.2. WebFlux config API

Same in Spring MVC

In your Java config implement the `WebFluxConfigurer` interface:

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

    // Implement configuration methods...

}
```

1.8.3. Conversion, formatting

Same in Spring MVC

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

To register custom formatters and converters:

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

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

}
```

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

1.8.4. Validation

Same in Spring MVC

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

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

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

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

}
```

Note that you can also register Validator's locally:

```
@Controller
public class MyController {

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

}
```

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

1.8.5. Content type resolvers

Same in Spring MVC

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

To customize the requested content type resolution:

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

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

1.8.6. HTTP message codecs

Same in Spring MVC

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

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

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

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

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

1. `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled.
2. `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled.

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

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

1.8.7. View resolvers

Same in Spring MVC

To configure view resolution:

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

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

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

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

    // ...

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

}
```

1.8.8. Static resources

Same in Spring MVC

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

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

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

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

}
```

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

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

For example in your Java config;

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

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

}
```

You can use `ResourceUrlProvider` to rewrite URLs and apply the full chain of resolvers and transformers — e.g. to insert versions. The WebFlux config provides a `ResourceUrlProvider` so it can be injected into others.

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

WebJars is also supported via `WebJarsResourceResolver` and automatically registered when

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

1.8.9. Path Matching

Same in Spring MVC

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

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

To customize those options:

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

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

}
```

1.8.10. Advanced config mode

Same in Spring MVC

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

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

```
@Configuration
public class WebConfig extends DelegatingWebFluxConfiguration {

    // ...

}
```

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

1.9. HTTP/2

Same 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 please check out 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.

Chapter 2. WebClient

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

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

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

2.1. Retrieve

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

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

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

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

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

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

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

2.2. Exchange

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

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

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

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

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

When using `exchange()` you must always use any of the body or toEntity methods of `ClientResponse` to ensure resources are released and to avoid potential issues with HTTP connection pooling. You can use `bodyToMono(Void.class)` if no response content is expected. However keep in mind that if the response does have content, the connection will be closed and will not be placed back in the pool.

2.3. Request body

The request body can be encoded from an Object:

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

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

You can also have a stream of objects encoded:

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

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

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

```
Person person = ... ;

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

2.3.1. Form data

To send form data, provide a `MultiValueMap<String, String>` as the body. Note that the content is automatically set to `"application/x-www-form-urlencoded"` by the `FormHttpMessageWriter`:

```
MultiValueMap<String, String> formData = ... ;

Mono<Void> result = client.post()
        .uri("/path", id)
        .syncBody(formData)
        .retrieve()
        .bodyToMono(Void.class);
```

You can also supply form data in-line via `BodyInserters`:

```
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);
```

2.3.2. Multipart data

To send multipart data, provide a `MultiValueMap<String, ?>` where values are either an Object representing the part body, or an `HttpEntity` representing the part body and headers. `MultipartBodyBuilder` can be used to build the parts:

```
MultipartBodyBuilder builder = new MultipartBodyBuilder();
builder.part("fieldPart", "fieldValue");
builder.part("filePart", new FileSystemResource("...logo.png"));
builder.part("jsonPart", new Person("Jason"));

MultiValueMap<String, HttpEntity<?>> parts = builder.build();

Mono<Void> result = client.post()
        .uri("/path", id)
        .syncBody(parts)
        .retrieve()
        .bodyToMono(Void.class);
```

Note that the content type for each part is automatically set based on the extension of the file being written or the type of Object. If you prefer you can also be more explicit and specify the content type for each part.

You can also supply multipart data in-line via `BodyInserters`:

```
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);
```

2.4. Builder options

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

To customize the underlying HTTP client:

```
SslContext sslContext = ...

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

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

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

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

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

The builder can be used to insert Filters.

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

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

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

2.5. Filters

`WebClient` supports interception style request filtering:

```
WebClient client = WebClient.builder()
        .filter((request, next) -> {
            ClientRequest filtered = ClientRequest.from(request)
                    .header("foo", "bar")
                    .build();
            return next.exchange(filtered);
        })
        .build();
```

`ExchangeFilterFunctions` provides a filter for basic authentication:

```
// static import of ExchangeFilterFunctions.basicAuthentication

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

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

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

Chapter 3. WebSockets

Same in Servlet stack

This part of the reference documentation covers support for Reactive stack, WebSocket messaging.

3.1. Introduction

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:

```
GET /spring-websocket-portfolio/portfolio HTTP/1.1
Host: localhost:8080
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: Uc9l9TMkWGbHFD2qnFHltg==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
```

Instead of the usual 200 status code, a server with WebSocket support returns:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1qVdfYHU9hPOl4JYYNXF623Gzn0=
Sec-WebSocket-Protocol: v10.stomp
```

After a successful handshake the TCP socket underlying the HTTP upgrade request remains open for both client and server to continue to send and receive messages.

A complete introduction of how WebSockets work is beyond the scope of this document. Please read RFC 6455, the WebSocket chapter of HTML5, or one of many introductions and tutorials on the Web.

Note that if a WebSocket server is running behind a web server (e.g. nginx) you will 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.

3.1.1. HTTP vs 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 just one URL for the initial connect and 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 there is no way to route or process a message unless client and server agree on message semantics.

WebSocket clients and servers can negotiate the use of a higher-level, messaging protocol (e.g. STOMP), via the `"Sec-WebSocket-Protocol"` header on the HTTP handshake request, or in the absence of that they need to come up with their own conventions.

3.1.2. When to use it?

WebSockets can make a web page dynamic and interactive. However in many cases a combination of Ajax and HTTP streaming and/or long polling could 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 (e.g. monitoring network failures) HTTP streaming or polling may provide an effective solution. It is the combination of low latency, high frequency and high volume that make the best case for the use WebSocket.

Keep in mind also that over the Internet, restrictive proxies outside 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 straight-forward decision than it is for public facing applications.

3.2. WebSocket API

Same in Servlet stack

The Spring Framework provides a WebSocket API that can be used to write client and server side applications that handle WebSocket messages.

3.2.1. WebSocketHandler

Same in Servlet stack

Creating a WebSocket server is as simple as implementing `WebSocketHandler`:

```
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) {
        // ...
    }
}
```

Spring WebFlux provides a `WebSocketHandlerAdapter` that can adapt WebSocket requests and use the above handler to handle the resulting WebSocket session. After the adapter is registered as a bean, you can map requests to your handler, for example using `SimpleUrlHandlerMapping`. This is shown below:

```
@Configuration
static class WebConfig {

    @Bean
    public HandlerMapping handlerMapping() {
        Map<String, WebSocketHandler> map = new HashMap<>();
        map.put("/path", new MyWebSocketHandler());

        SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
        mapping.setUrlMap(map);
        mapping.setOrder(-1); // before annotated controllers
        return mapping;
    }

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

3.2.2. WebSocket Handshake

Same in Servlet stack

`WebSocketHandlerAdapter` does not perform WebSocket handshakes itself. Instead it delegates to an instance of `WebSocketService`. The default `WebSocketService` implementation is

`HandshakeWebSocketService`.

The `HandshakeWebSocketService` performs basic checks on the WebSocket request and delegates to a server-specific `RequestUpgradeStrategy`. At present upgrade strategies exist for Reactor Netty, Tomcat, Jetty, and Undertow.

3.2.3. Server config

Same in Servlet stack

The `RequestUpgradeStrategy` for each server exposes the WebSocket-related configuration options available for the underlying WebSocket engine. Below is an example of setting WebSocket options when running on Tomcat:

```
@Configuration
static 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);
    }
}
```

Check the upgrade strategy for your server to see what options are available. Currently only Tomcat and Jetty expose such options.

3.2.4. CORS

Same in 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, etc. If for any reason you can't do that, you can also set the `corsConfigurations` property on the `SimpleUrlHandler` to specify CORS settings by URL pattern. If both are specified they're combined via the `combine` method on `CorsConfiguration`.

3.3. WebSocketClient

Spring WebFlux provides a `WebSocketClient` abstraction with implementations for Reactor Netty, Tomcat, Jetty, Undertow, and standard Java (i.e. JSR-356).

The Tomcat client is effectively an extension of the standard Java one with some extra functionality in the `WebSocketSession` handling taking advantage of Tomcat specific API to suspend receiving messages for back pressure.

To start a WebSocket session, create an instance of the client and use its `execute` methods:

```
WebSocketClient client = new ReactorNettyWebSocketClient();

URI url = new URI("ws://localhost:8080/path");
client.execute(url, session ->
        session.receive()
                .doOnNext(System.out::println)
                .then());
```

Some clients, e.g. Jetty, implement `Lifecycle` and need to be started in stopped before you can use them. All clients have constructor options related to configuration of the underlying WebSocket client.

Chapter 4. Testing

Same in Spring MVC

The `spring-test` module provides mock implementations of `ServerHttpRequest`, `ServerHttpResponse`, and `ServerWebExchange`. See Spring Web Reactive mock objects.

The WebTestClient builds on these mock request and response objects to provide support for testing WebFlux applications without and HTTP server. The `WebTestClient` can be used for end-to-end integration tests too.

Chapter 5. Reactive Libraries

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

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

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

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

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

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