RSocket

Version 5.3.30

This section describes Spring Framework's support for the RSocket protocol.

Chapter 1. Overview

RSocket is an application protocol for multiplexed, duplex communication over TCP, WebSocket, and other byte stream transports, using one of the following interaction models:

- `Request-Response` — send one message and receive one back.
- `Request-Stream` — send one message and receive a stream of messages back.
- `Channel` — send streams of messages in both directions.
- `Fire-and-Forget` — send a one-way message.

Once the initial connection is made, the "client" vs "server" distinction is lost as both sides become symmetrical and each side can initiate one of the above interactions. This is why in the protocol calls the participating sides "requester" and "responder" while the above interactions are called "request streams" or simply "requests".

These are the key features and benefits of the RSocket protocol:

- Reactive Streams semantics across network boundary — for streaming requests such as `Request-Stream` and `Channel`, back pressure signals travel between requester and responder, allowing a requester to slow down a responder at the source, hence reducing reliance on network layer congestion control, and the need for buffering at the network level or at any level.
- Request throttling — this feature is named "Leasing" after the `LEASE` frame that can be sent from each end to limit the total number of requests allowed by other end for a given time. Leases are renewed periodically.
- Session resumption — this is designed for loss of connectivity and requires some state to be maintained. The state management is transparent for applications, and works well in combination with back pressure which can stop a producer when possible and reduce the amount of state required.
- Fragmentation and re-assembly of large messages.
- Keepalive (heartbeats).

RSocket has implementations in multiple languages. The Java library is built on Project Reactor, and Reactor Netty for the transport. That means signals from Reactive Streams Publishers in your application propagate transparently through RSocket across the network.

1.1. The Protocol

One of the benefits of RSocket is that it has well defined behavior on the wire and an easy to read specification along with some protocol extensions. Therefore it is a good idea to read the spec, independent of language implementations and higher level framework APIs. This section provides a succinct overview to establish some context.

Connecting

Initially a client connects to a server via some low level streaming transport such as TCP or

WebSocket and sends a SETUP frame to the server to set parameters for the connection.

The server may reject the SETUP frame, but generally after it is sent (for the client) and received (for the server), both sides can begin to make requests, unless SETUP indicates use of leasing semantics to limit the number of requests, in which case both sides must wait for a LEASE frame from the other end to permit making requests.

Making Requests

Once a connection is established, both sides may initiate a request through one of the frames REQUEST_RESPONSE, REQUEST_STREAM, REQUEST_CHANNEL, or REQUEST_FNF. Each of those frames carries one message from the requester to the responder.

The responder may then return PAYLOAD frames with response messages, and in the case of REQUEST_CHANNEL the requester may also send PAYLOAD frames with more request messages.

When a request involves a stream of messages such as Request-Stream and Channel, the responder must respect demand signals from the requester. Demand is expressed as a number of messages. Initial demand is specified in REQUEST_STREAM and REQUEST_CHANNEL frames. Subsequent demand is signaled via REQUEST_N frames.

Each side may also send metadata notifications, via the METADATA_PUSH frame, that do not pertain to any individual request but rather to the connection as a whole.

Message Format

RSocket messages contain data and metadata. Metadata can be used to send a route, a security token, etc. Data and metadata can be formatted differently. Mime types for each are declared in the SETUP frame and apply to all requests on a given connection.

While all messages can have metadata, typically metadata such as a route are per-request and therefore only included in the first message on a request, i.e. with one of the frames REQUEST_RESPONSE, REQUEST_STREAM, REQUEST_CHANNEL, or REQUEST_FNF.

Protocol extensions define common metadata formats for use in applications:

- Composite Metadata-- multiple, independently formatted metadata entries.
- Routing — the route for a request.

1.2. Java Implementation

The Java implementation for RSocket is built on Project Reactor. The transports for TCP and WebSocket are built on Reactor Netty. As a Reactive Streams library, Reactor simplifies the job of implementing the protocol. For applications it is a natural fit to use Flux and Mono with declarative operators and transparent back pressure support.

The API in RSocket Java is intentionally minimal and basic. It focuses on protocol features and leaves the application programming model (e.g. RPC codegen vs other) as a higher level, independent concern.

The main contract io.rsocket.RSocket models the four request interaction types with `Mono` representing a promise for a single message, `Flux` a stream of messages, and `io.rsocket.Payload` the actual message with access to data and metadata as byte buffers. The `RSocket` contract is used symmetrically. For requesting, the application is given an `RSocket` to perform requests with. For responding, the application implements `RSocket` to handle requests.

This is not meant to be a thorough introduction. For the most part, Spring applications will not have to use its API directly. However it may be important to see or experiment with RSocket independent of Spring. The RSocket Java repository contains a number of sample apps that demonstrate its API and protocol features.

1.3. Spring Support

The `spring-messaging` module contains the following:

- RSocketRequester — fluent API to make requests through an `io.rsocket.RSocket` with data and metadata encoding/decoding.
- Annotated Responders — `@MessageMapping` annotated handler methods for responding.

The `spring-web` module contains `Encoder` and `Decoder` implementations such as Jackson CBOR/JSON, and Protobuf that RSocket applications will likely need. It also contains the `PathPatternParser` that can be plugged in for efficient route matching.

Spring Boot 2.2 supports standing up an RSocket server over TCP or WebSocket, including the option to expose RSocket over WebSocket in a WebFlux server. There is also client support and auto-configuration for an `RSocketRequester.Builder` and `RSocketStrategies`. See the RSocket section in the Spring Boot reference for more details.

Spring Security 5.2 provides RSocket support.

Spring Integration 5.2 provides inbound and outbound gateways to interact with RSocket clients and servers. See the Spring Integration Reference Manual for more details.

Spring Cloud Gateway supports RSocket connections.

Chapter 2. RSocketRequester

`RSocketRequester` provides a fluent API to perform RSocket requests, accepting and returning objects for data and metadata instead of low level data buffers. It can be used symmetrically, to make requests from clients and to make requests from servers.

2.1. Client Requester

To obtain an `RSocketRequester` on the client side is to connect to a server which involves sending an RSocket `SETUP` frame with connection settings. `RSocketRequester` provides a builder that helps to prepare an `io.rsocket.core.RSocketConnector` including connection settings for the `SETUP` frame.

This is the most basic way to connect with default settings:

Java

```
RSocketRequester requester = RSocketRequester.builder().tcp("localhost", 7000);

URI url = URI.create("https://example.org:8080/rsocket");
RSocketRequester requester = RSocketRequester.builder().webSocket(url);
```

Kotlin

```
val requester = RSocketRequester.builder().tcp("localhost", 7000)

URI url = URI.create("https://example.org:8080/rsocket");
val requester = RSocketRequester.builder().webSocket(url)
```

The above does not connect immediately. When requests are made, a shared connection is established transparently and used.

2.1.1. Connection Setup

`RSocketRequester.Builder` provides the following to customize the initial `SETUP` frame:

- `dataMimeType(MimeType)` — set the mime type for data on the connection.
- `metadataMimeType(MimeType)` — set the mime type for metadata on the connection.
- `setupData(Object)` — data to include in the `SETUP`.
- `setupRoute(String, Object…)` — route in the metadata to include in the `SETUP`.
- `setupMetadata(Object, MimeType)` — other metadata to include in the `SETUP`.

For data, the default mime type is derived from the first configured `Decoder`. For metadata, the default mime type is composite metadata which allows multiple metadata value and mime type pairs per request. Typically both don't need to be changed.

Data and metadata in the `SETUP` frame is optional. On the server side, @ConnectMapping methods

can be used to handle the start of a connection and the content of the `SETUP` frame. Metadata may be used for connection level security.

2.1.2. Strategies

`RSocketRequester.Builder` accepts `RSocketStrategies` to configure the requester. You'll need to use this to provide encoders and decoders for (de)-serialization of data and metadata values. By default only the basic codecs from `spring-core` for `String`, `byte[]`, and `ByteBuffer` are registered. Adding `spring-web` provides access to more that can be registered as follows:

Java

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .encoders(encoders -> encoders.add(new Jackson2CborEncoder()))
    .decoders(decoders -> decoders.add(new Jackson2CborDecoder()))
    .build();

RSocketRequester requester = RSocketRequester.builder()
    .rsocketStrategies(strategies)
    .tcp("localhost", 7000);
```

Kotlin

```
val strategies = RSocketStrategies.builder()
        .encoders { it.add(Jackson2CborEncoder()) }
        .decoders { it.add(Jackson2CborDecoder()) }
        .build()

val requester = RSocketRequester.builder()
        .rsocketStrategies(strategies)
        .tcp("localhost", 7000)
```

`RSocketStrategies` is designed for re-use. In some scenarios, e.g. client and server in the same application, it may be preferable to declare it in Spring configuration.

2.1.3. Client Responders

`RSocketRequester.Builder` can be used to configure responders to requests from the server.

You can use annotated handlers for client-side responding based on the same infrastructure that's used on a server, but registered programmatically as follows:

Java

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .routeMatcher(new PathPatternRouteMatcher())  ①
    .build();

SocketAcceptor responder =
    RSocketMessageHandler.responder(strategies, new ClientHandler()); ②

RSocketRequester requester = RSocketRequester.builder()
    .rsocketConnector(connector -> connector.acceptor(responder)) ③
    .tcp("localhost", 7000);
```

① Use `PathPatternRouteMatcher`, if `spring-web` is present, for efficient route matching.

② Create a responder from a class with `@MessageMapping` and/or `@ConnectMapping` methods.

③ Register the responder.

Kotlin

```
val strategies = RSocketStrategies.builder()
        .routeMatcher(PathPatternRouteMatcher())  ①
        .build()

val responder =
    RSocketMessageHandler.responder(strategies, new ClientHandler()); ②

val requester = RSocketRequester.builder()
        .rsocketConnector { it.acceptor(responder) } ③
        .tcp("localhost", 7000)
```

① Use `PathPatternRouteMatcher`, if `spring-web` is present, for efficient route matching.

② Create a responder from a class with `@MessageMapping` and/or `@ConnectMapping` methods.

③ Register the responder.

Note the above is only a shortcut designed for programmatic registration of client responders. For alternative scenarios, where client responders are in Spring configuration, you can still declare `RSocketMessageHandler` as a Spring bean and then apply as follows:

Java

```
ApplicationContext context = ... ;
RSocketMessageHandler handler = context.getBean(RSocketMessageHandler.class);

RSocketRequester requester = RSocketRequester.builder()
    .rsocketConnector(connector -> connector.acceptor(handler.responder()))
    .tcp("localhost", 7000);
```

Kotlin

```
import org.springframework.beans.factory.getBean

val context: ApplicationContext = ...
val handler = context.getBean<RSocketMessageHandler>()

val requester = RSocketRequester.builder()
        .rsocketConnector { it.acceptor(handler.responder()) }
        .tcp("localhost", 7000)
```

For the above you may also need to use `setHandlerPredicate` in `RSocketMessageHandler` to switch to a different strategy for detecting client responders, e.g. based on a custom annotation such as `@RSocketClientResponder` vs the default `@Controller`. This is necessary in scenarios with client and server, or multiple clients in the same application.

See also Annotated Responders, for more on the programming model.

2.1.4. Advanced

`RSocketRequesterBuilder` provides a callback to expose the underlying `io.rsocket.core.RSocketConnector` for further configuration options for keepalive intervals, session resumption, interceptors, and more. You can configure options at that level as follows:

Java

```
RSocketRequester requester = RSocketRequester.builder()
    .rsocketConnector(connector -> {
        // ...
    })
    .tcp("localhost", 7000);
```

Kotlin

```
val requester = RSocketRequester.builder()
        .rsocketConnector {
            //...
        }
        .tcp("localhost", 7000)
```

2.2. Server Requester

To make requests from a server to connected clients is a matter of obtaining the requester for the connected client from the server.

In Annotated Responders, `@ConnectMapping` and `@MessageMapping` methods support an `RSocketRequester` argument. Use it to access the requester for the connection. Keep in mind that `@ConnectMapping` methods are essentially handlers of the `SETUP` frame which must be handled before

requests can begin. Therefore, requests at the very start must be decoupled from handling. For example:

Java

```
@ConnectMapping
Mono<Void> handle(RSocketRequester requester) {
    requester.route("status").data("5")
        .retrieveFlux(StatusReport.class)
        .subscribe(bar -> { ①
            // ...
        });
    return ... ②
}
```

① Start the request asynchronously, independent from handling.

② Perform handling and return completion `Mono<Void>`.

Kotlin

```
@ConnectMapping
suspend fun handle(requester: RSocketRequester) {
    GlobalScope.launch {
        requester.route("status").data("5").retrieveFlow<StatusReport>().collect { ①
            // ...
        }
    }
    /// ... ②
}
```

① Start the request asynchronously, independent from handling.

② Perform handling in the suspending function.

2.3. Requests

Once you have a client or server requester, you can make requests as follows:

Java

```
ViewBox viewBox = ... ;

Flux<AirportLocation> locations = requester.route("locate.radars.within") ①
        .data(viewBox) ②
        .retrieveFlux(AirportLocation.class); ③
```

① Specify a route to include in the metadata of the request message.

② Provide data for the request message.

③ Declare the expected response.

Kotlin

```
val viewBox: ViewBox = ...

val locations = requester.route("locate.radars.within") ①
        .data(viewBox) ②
        .retrieveFlow<AirportLocation>() ③
```

① Specify a route to include in the metadata of the request message.

② Provide data for the request message.

③ Declare the expected response.

The interaction type is determined implicitly from the cardinality of the input and output. The above example is a `Request-Stream` because one value is sent and a stream of values is received. For the most part you don't need to think about this as long as the choice of input and output matches an RSocket interaction type and the types of input and output expected by the responder. The only example of an invalid combination is many-to-one.

The `data(Object)` method also accepts any Reactive Streams `Publisher`, including `Flux` and `Mono`, as well as any other producer of value(s) that is registered in the `ReactiveAdapterRegistry`. For a multi-value `Publisher` such as `Flux` which produces the same types of values, consider using one of the overloaded `data` methods to avoid having type checks and `Encoder` lookup on every element:

```
data(Object producer, Class<?> elementClass);
data(Object producer, ParameterizedTypeReference<?> elementTypeRef);
```

The `data(Object)` step is optional. Skip it for requests that don't send data:

Java

```
Mono<AirportLocation> location = requester.route("find.radar.EWR"))
    .retrieveMono(AirportLocation.class);
```

Kotlin

```
import org.springframework.messaging.rsocket.retrieveAndAwait

val location = requester.route("find.radar.EWR")
    .retrieveAndAwait<AirportLocation>()
```

Extra metadata values can be added if using composite metadata (the default) and if the values are supported by a registered `Encoder`. For example:

Java

```
String securityToken = ... ;
ViewBox viewBox = ... ;
MimeType mimeType = MimeType.valueOf("message/x.rsocket.authentication.bearer.v0");

Flux<AirportLocation> locations = requester.route("locate.radars.within")
        .metadata(securityToken, mimeType)
        .data(viewBox)
        .retrieveFlux(AirportLocation.class);
```

Kotlin

```
import org.springframework.messaging.rsocket.retrieveFlow

val requester: RSocketRequester = ...

val securityToken: String = ...
val viewBox: ViewBox = ...
val mimeType = MimeType.valueOf("message/x.rsocket.authentication.bearer.v0")

val locations = requester.route("locate.radars.within")
        .metadata(securityToken, mimeType)
        .data(viewBox)
        .retrieveFlow<AirportLocation>()
```

For `Fire-and-Forget` use the `send()` method that returns `Mono<Void>`. Note that the `Mono` indicates only that the message was successfully sent, and not that it was handled.

For `Metadata-Push` use the `sendMetadata()` method with a `Mono<Void>` return value.

Chapter 3. Annotated Responders

RSocket responders can be implemented as `@MessageMapping` and `@ConnectMapping` methods. `@MessageMapping` methods handle individual requests while `@ConnectMapping` methods handle connection-level events (setup and metadata push). Annotated responders are supported symmetrically, for responding from the server side and for responding from the client side.

3.1. Server Responders

To use annotated responders on the server side, add `RSocketMessageHandler` to your Spring configuration to detect `@Controller` beans with `@MessageMapping` and `@ConnectMapping` methods:

Java

```
@Configuration
static class ServerConfig {

    @Bean
    public RSocketMessageHandler rsocketMessageHandler() {
        RSocketMessageHandler handler = new RSocketMessageHandler();
        handler.routeMatcher(new PathPatternRouteMatcher());
        return handler;
    }
}
```

Kotlin

```
@Configuration
class ServerConfig {

    @Bean
    fun rsocketMessageHandler() = RSocketMessageHandler().apply {
        routeMatcher = PathPatternRouteMatcher()
    }
}
```

Then start an RSocket server through the Java RSocket API and plug the `RSocketMessageHandler` for the responder as follows:

Java

```
ApplicationContext context = ... ;
RSocketMessageHandler handler = context.getBean(RSocketMessageHandler.class);

CloseableChannel server =
    RSocketServer.create(handler.responder())
        .bind(TcpServerTransport.create("localhost", 7000))
        .block();
```

Kotlin

```
import org.springframework.beans.factory.getBean

val context: ApplicationContext = ...
val handler = context.getBean<RSocketMessageHandler>()

val server = RSocketServer.create(handler.responder())
        .bind(TcpServerTransport.create("localhost", 7000))
        .awaitSingle()
```

`RSocketMessageHandler` supports composite and routing metadata by default. You can set its `MetadataExtractor` if you need to switch to a different mime type or register additional metadata mime types.

You'll need to set the `Encoder` and `Decoder` instances required for metadata and data formats to support. You'll likely need the `spring-web` module for codec implementations.

By default `SimpleRouteMatcher` is used for matching routes via `AntPathMatcher`. We recommend plugging in the `PathPatternRouteMatcher` from `spring-web` for efficient route matching. RSocket routes can be hierarchical but are not URL paths. Both route matchers are configured to use "." as separator by default and there is no URL decoding as with HTTP URLs.

`RSocketMessageHandler` can be configured via `RSocketStrategies` which may be useful if you need to share configuration between a client and a server in the same process:

Java

```
@Configuration
static class ServerConfig {

    @Bean
    public RSocketMessageHandler rsocketMessageHandler() {
        RSocketMessageHandler handler = new RSocketMessageHandler();
        handler.setRSocketStrategies(rsocketStrategies());
        return handler;
    }

    @Bean
    public RSocketStrategies rsocketStrategies() {
        return RSocketStrategies.builder()
            .encoders(encoders -> encoders.add(new Jackson2CborEncoder()))
            .decoders(decoders -> decoders.add(new Jackson2CborDecoder()))
            .routeMatcher(new PathPatternRouteMatcher())
            .build();
    }
}
```

Kotlin

```
@Configuration
class ServerConfig {

    @Bean
    fun rsocketMessageHandler() = RSocketMessageHandler().apply {
        rSocketStrategies = rsocketStrategies()
    }

    @Bean
    fun rsocketStrategies() = RSocketStrategies.builder()
            .encoders { it.add(Jackson2CborEncoder()) }
            .decoders { it.add(Jackson2CborDecoder()) }
            .routeMatcher(PathPatternRouteMatcher())
            .build()
}
```

3.2. Client Responders

Annotated responders on the client side need to be configured in the `RSocketRequester.Builder`. For details, see Client Responders.

3.3. @MessageMapping

Once server or client responder configuration is in place, `@MessageMapping` methods can be used as

follows:

Java

```
@Controller
public class RadarsController {

    @MessageMapping("locate.radars.within")
    public Flux<AirportLocation> radars(MapRequest request) {
        // ...
    }
}
```

Kotlin

```
@Controller
class RadarsController {

    @MessageMapping("locate.radars.within")
    fun radars(request: MapRequest): Flow<AirportLocation> {
        // ...
    }
}
```

The above `@MessageMapping` method responds to a Request-Stream interaction having the route "locate.radars.within". It supports a flexible method signature with the option to use the following method arguments:

Method Argument	Description
`@Payload`	The payload of the request. This can be a concrete value of asynchronous types like `Mono` or `Flux`. **Note:** Use of the annotation is optional. A method argument that is not a simple type and is not any of the other supported arguments, is assumed to be the expected payload.
`RSocketRequester`	Requester for making requests to the remote end.
`@DestinationVariable`	Value extracted from the route based on variables in the mapping pattern, e.g. `@MessageMapping("find.radar.{id}")`.
`@Header`	Metadata value registered for extraction as described in MetadataExtractor.
`@Headers Map<String, Object>`	All metadata values registered for extraction as described in MetadataExtractor.

The return value is expected to be one or more Objects to be serialized as response payloads. That can be asynchronous types like `Mono` or `Flux`, a concrete value, or either `void` or a no-value asynchronous type such as `Mono<Void>`.

The RSocket interaction type that an `@MessageMapping` method supports is determined from the cardinality of the input (i.e. `@Payload` argument) and of the output, where cardinality means the following:

Cardinality	Description
1	Either an explicit value, or a single-value asynchronous type such as `Mono<T>`.
Many	A multi-value asynchronous type such as `Flux<T>`.
0	For input this means the method does not have an `@Payload` argument. For output this is `void` or a no-value asynchronous type such as `Mono<Void>`.

The table below shows all input and output cardinality combinations and the corresponding interaction type(s):

Input Cardinality	Output Cardinality	Interaction Types
0, 1	0	Fire-and-Forget, Request-Response
0, 1	1	Request-Response
0, 1	Many	Request-Stream
Many	0, 1, Many	Request-Channel

3.4. @ConnectMapping

`@ConnectMapping` handles the `SETUP` frame at the start of an RSocket connection, and any subsequent metadata push notifications through the `METADATA_PUSH` frame, i.e. `metadataPush(Payload)` in `io.rsocket.RSocket`.

`@ConnectMapping` methods support the same arguments as @MessageMapping but based on metadata and data from the `SETUP` and `METADATA_PUSH` frames. `@ConnectMapping` can have a pattern to narrow handling to specific connections that have a route in the metadata, or if no patterns are declared then all connections match.

`@ConnectMapping` methods cannot return data and must be declared with `void` or `Mono<Void>` as the return value. If handling returns an error for a new connection then the connection is rejected. Handling must not be held up to make requests to the `RSocketRequester` for the connection. See Server Requester for details.

Chapter 4. MetadataExtractor

Responders must interpret metadata. Composite metadata allows independently formatted metadata values (e.g. for routing, security, tracing) each with its own mime type. Applications need a way to configure metadata mime types to support, and a way to access extracted values.

`MetadataExtractor` is a contract to take serialized metadata and return decoded name-value pairs that can then be accessed like headers by name, for example via `@Header` in annotated handler methods.

`DefaultMetadataExtractor` can be given `Decoder` instances to decode metadata. Out of the box it has built-in support for "message/x.rsocket.routing.v0" which it decodes to `String` and saves under the "route" key. For any other mime type you'll need to provide a `Decoder` and register the mime type as follows:

Java

```
DefaultMetadataExtractor extractor = new DefaultMetadataExtractor(metadataDecoders);
extractor.metadataToExtract(fooMimeType, Foo.class, "foo");
```

Kotlin

```
import org.springframework.messaging.rsocket.metadataToExtract

val extractor = DefaultMetadataExtractor(metadataDecoders)
extractor.metadataToExtract<Foo>(fooMimeType, "foo")
```

Composite metadata works well to combine independent metadata values. However the requester might not support composite metadata, or may choose not to use it. For this, `DefaultMetadataExtractor` may needs custom logic to map the decoded value to the output map. Here is an example where JSON is used for metadata:

Java

```
DefaultMetadataExtractor extractor = new DefaultMetadataExtractor(metadataDecoders);
extractor.metadataToExtract(
    MimeType.valueOf("application/vnd.myapp.metadata+json"),
    new ParameterizedTypeReference<Map<String,String>>() {},
    (jsonMap, outputMap) -> {
        outputMap.putAll(jsonMap);
    });
```

Kotlin

```
import org.springframework.messaging.rsocket.metadataToExtract

val extractor = DefaultMetadataExtractor(metadataDecoders)
extractor.metadataToExtract<Map<String,
String>>(MimeType.valueOf("application/vnd.myapp.metadata+json")) { jsonMap, outputMap
->
    outputMap.putAll(jsonMap)
}
```

When configuring `MetadataExtractor` through `RSocketStrategies`, you can let `RSocketStrategies.Builder` create the extractor with the configured decoders, and simply use a callback to customize registrations as follows:

Java

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .metadataExtractorRegistry(registry -> {
        registry.metadataToExtract(fooMimeType, Foo.class, "foo");
        // ...
    })
    .build();
```

Kotlin

```
import org.springframework.messaging.rsocket.metadataToExtract

val strategies = RSocketStrategies.builder()
        .metadataExtractorRegistry { registry: MetadataExtractorRegistry ->
            registry.metadataToExtract<Foo>(fooMimeType, "foo")
            // ...
        }
        .build()
```