Spring Retrosocket 0.0.1-SNAPSHOT reference guide

Josh Long, Mario Gray

Table of Contents

Spring Retrosocket aims to provide a Feign-like or Retrofit-like experience for declarative RSocket-based clients. This guide introduces Retrosocket functionality and their uses.

Chapter 1. Build the Code

Build the code:

```
mvn clean install
```

It’s early days yet, so there may be some build breaks. Skip the tests if needed:

```
mvn -DskipTests=true clean install
```

Chapter 2. Configuring Your Build

The easiest way might be to go to the Spring Initializr and generate a new project. Make sure that you specify the `snapshot` or `milestone` dependencies and then add the following to your build.

```
<dependency>
    <groupId>org.springframework.retrosocket</groupId>
    <artifactId>spring-retrosocket</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

If you have an existing build, make sure that you have the `snapshot` or `milestone` Spring repositories.

```
<repositories>
   <repository>
     <id>spring-milestones</id>
     <name>Spring Milestones</name>
     <url>https://repo.spring.io/milestone</url>
   </repository>
   <repository>
     <id>spring-snapshots</id>
     <name>Spring Snapshots</name>
     <url>https://repo.spring.io/snapshot</url>
     <snapshots>
       <enabled>true</enabled>
     </snapshots>
   </repository>
 </repositories>
```

Chapter 3. Basic Usage

In your Java code, you need to enable the RSocket client support. Use the `@EnableRSocketClient` annotation. You'll also need to define an `RSocketRequester` bean.

```
@SpringBootApplication
@EnableRSocketClient
class RSocketClientApplication {

 @Bean
 RSocketRequester requester(RSocketRequester.Builder builder) {
  return builder.connectTcp("localhost", 8888).block();
 }
}
```

then, define an RSocket client interface, like this:

```
@RSocketClient
interface GreetingClient {

    @MessageMapping("supplier")
    Mono<GreetingResponse> greet();

    @MessageMapping("request-response")
    Mono<GreetingResponse> requestResponse(Mono<String> name);

    @MessageMapping("fire-and-forget")
    Mono<Void> fireAndForget(Mono<String> name);

    @MessageMapping("destination.variables.and.payload.annotations.{name}.{age}")
    Mono<String> greetMonoNameDestinationVariable(
            @DestinationVariable("name") String name,
        @DestinationVariable("age") int age,
            @Payload Mono<String> payload);
}
```

If you invoke methods on this interface, it'll, in turn, invoke endpoints using the configured `RSocketRequester` for you, turning destination variables into route variables and turning your payload into the data for the request.

Chapter 4. Mapping Headers (RSocket metadata) to the RSocket request

You can map `@Header` elements to parameters in the method invocation. The header parameters get sent as composite RSocket metadata. Normal invocations of RSocket metadata would require two parts - a mime type and a value that can be encoded. The encoding is a separate issue - Spring ships with a ton of encoders/decoders out of the box, but by default, Spring Framework's built-in support uses something called `CBOR`. There is still the question of how to communicate the mime-type. We expect the mime-type as the `value()` attribute for the `@Header` annotation. Thus:

```
import com.joshlong.rsocket.client.RSocketClient;
import org.springframework.messaging.handler.annotation.Header;
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.handler.annotation.Payload;
import reactor.core.publisher.Mono;

@RSocketClient
interface GreetingClient {

    @MessageMapping("greetings")
    Mono<String> greet(@Header( "messaging/x.bootiful.client-id") String clientId,
@Payload Mono<String> name);

}
```

This method definition should line up with the expectations for composite metadata on the responder side, of course.

Chapter 5. Pairing `RSocketRequesters` to `@RSocketClient` interfaces

You can annotate your interfaces with a `@Qualifier` annotation (or a meta-annotated qualifier of your own making) and then annotate an `RSocketRequester`. This module will use that `RSocketRequester` when servicing methods on a particular interface.

The following demonstrates the concept in action. RSocket connections are stateful. Once they've connected, they stay connected, and all subsequent interactions are assumed to be against the already established connection. Therefore, each `RSocketRequester` talks to a different logical (and physical) service, unlike, e.g., a `WebClient`, which may you may use to talk to any arbitrary host and port.

```
@RSocketClient
@Qualifier(Constants.QUALIFIER_2)
interface GreetingClient {

    @MessageMapping("greetings-with-name")
    Mono<Greeting> greet(Mono<String> name);

}

@RSocketClient
@PersonQualifier
interface PersonClient {

    @MessageMapping("people")
    Flux<Person> people();

}

@EnableRSocketClients
@SpringBootApplication
class RSocketClientConfiguration {

    @Bean
    @PersonQualifier // meta-annotation
    // @Qualifier(Constants.QUALIFIER_1)
    RSocketRequester one(@Value("${" + Constants.QUALIFIER_1 + ".port}") int port,
RSocketRequester.Builder builder) {
        return builder.connectTcp("localhost", port).block();
    }


    @Bean
    @Qualifier(Constants.QUALIFIER_2) // direct-annotation
    RSocketRequester two(@Value("${" + Constants.QUALIFIER_2 + ".port}") int port,
RSocketRequester.Builder builder) {
        return builder.connectTcp("localhost", port).block();
    }
}

@Target({ ElementType.FIELD, ElementType.METHOD, ElementType.TYPE, ElementType
.PARAMETER })
@Retention(RetentionPolicy.RUNTIME)
@Qualifier(Constants.QUALIFIER_1)
@interface PersonQualifier {
}
```

Chapter 6. Contact us

Did you not find what you sought? We're happy to help! We're always available on the Github Issues section for this repository.