Holon Platform Project Reactor integration Module - Reference manual

Table of Contents

1. Introduction

The Reactor module provides integration between the Holon Platform core APIs, such as `Datastore` and `RestClient`, and the Project Reactor reactive programming model, using the `Flux` and `Mono` APIs.

1.1. Sources and contributions

The Holon Platform **Reactor** module source code is available from the GitHub repository https://github.com/holon-platform/holon-reactor.

See the repository `README` file for information about:

- The source code structure.
- How to build the module artifacts from sources.
- Where to find the code examples.
- How to contribute to the module development.

2. Obtaining the artifacts

The Holon Platform uses Maven for projects build and configuration. All the platform artifacts are published in the **Maven Central Repository**, so there is no need to explicitly declare additional repositories in your project `pom` file.

At the top of each *section* of this documentation you will find the Maven *coordinates* (group id, artifact id and version) to obtain the artifact(s) as a dependency for your project.

A **BOM (Bill Of Materials)** `pom` is provided to import the available dependencies for a specific version in your projects. The Maven coordinates for the core BOM are the following:

Maven coordinates:

```
<groupId>com.holon-platform.reactor</groupId>
<artifactId>holon-reactor-bom</artifactId>
<version>5.3.0</version>
```

The BOM can be imported in a Maven project in the following way:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.holon-platform.reactor</groupId>
      <artifactId>holon-reactor-bom</artifactId>
      <version>5.3.0</version>
      <strong><type>pom</type></strong>
      <strong><scope>import</scope></strong>
    </dependency>
  </dependencies>
</dependencyManagement>
```

2.1. Using the Platform BOM

The Holon Platform provides an **overall Maven BOM (Bill of Materials)** to easily obtain all the available platform artifacts.

See Obtain the platform artifacts for details.

3. Reactive Datastore

Maven coordinates:

```
<groupId>com.holon-platform.reactor</groupId>
<artifactId>holon-reactor-datastore</artifactId>
<version>5.3.0</version>
```

The ReactiveDatastore API is the reactive version of the Holon Platform core Datastore API using the Project Reactor programming model.

The `ReactiveDatastore` operations provides the same semantic and parameters as the core `Datastore` API operations, but use the Project Reactor `Mono` and `Flux` types as operation results, dependently on the result cardinality.

Additionaly, the `ReactiveBulkInsert`, `ReactiveBulkUpdate` and `ReactiveBulkUpdate` bulk operations handlers are provided to obtain a *bulk* operation result as a `Mono<OperationResult>` type.

The ReactiveQuery API can be used to configure and execute queries, and obtain the query results as `Mono` and `Flux` types, according to the query projection cardinality.

Finally, a ReactiveTransaction API is available for reactive Datastores which supports transactions, to reactively handle the transactions lifecycle.

See the available `ReactiveDatastore` implementations for more information about reactive Datastores.

The **MongoDB** `ReactiveDatastore` implementation documentation is available form here.

4. Reactive RestClient

Maven coordinates:

```
<groupId>com.holon-platform.reactor</groupId>
<artifactId>holon-reactor-http</artifactId>
<version>5.3.0</version>
```

The ReactiveRestClient API is the reactive version of the Holon Platform core RestClient API using the Project Reactor programming model.

The `ReactiveRestClient` operations provides the same semantic and parameters as the core `RestClient` API operations, but use the Project Reactor `Mono` and `Flux` types as operation results, dependently on the result cardinality.

The `ReactiveInvocation` API is used to configure and execute HTTP operations using the REST paradigm, providing the operation results through the `Mono` and `Flux` types.

4.1. Obtain a `ReactiveRestClient` instance

Concrete `ReactiveRestClient` implementations are obtained from a ReactiveRestClientFactory, registered using Java service extensions through a `com.holonplatform.reactor.http.ReactiveRestClientFactory` file under the `META-INF/services` folder.

A `ReactiveRestClient` instance can be obtained using one of the `create(…)` methods provided by the interface, either specifying the *fully qualified* class name of the `ReactiveRestClient` implementation to obtain or using the default implementation according to the available `ReactiveRestClientFactory` within the current `ClassLoader` (a specific `ClassLoader` can be used instead of the current one).

If more than one `RestClienReactiveRestClientFactorytFactory` is bound to the same `ReactiveRestClient` implementation type, or if more than one `ReactiveRestClientFactory` is available in the `ClassLoader` when the implementation class is not specified, the `ReactiveRestClientFactory` to use to build the `ReactiveRestClient` instance is selected according to the factory priority level, which can be specified using the `Priority` annotation, if available.

The `forTarget(…)` static methods of the `ReactiveRestClient` interface can be used as shorters to create a `ReactiveRestClient` using the default implementation and setting a default base **URI** to use for the client requests.

ReactiveRestClient creation examples

```
ReactiveRestClient client = ReactiveRestClient.create(); ①

client = ReactiveRestClient.create(
"com.holonplatform.jaxrs.client.reactor.JaxrsReactiveReactiveRestClient"); ②

client = ReactiveRestClient.forTarget("https://host/api"); ③
```

① Create a `ReactiveRestClient` API using the default available implementation for current ClassLoader

② Create a `ReactiveRestClient` API using a specific implementation class name

③ Create a `ReactiveRestClient` API using the default available implementation and setting the *default* base URI

4.1.1. Available implementations

The `ReactiveRestClient` implementations provided by the Holon Platform are are:

- A **JAX-RS** based implementation, using a standard JAX-RS `Client` to perform invocations, available from the holon-jaxrs.html#JaxrsReactiveRestClient[Holon platform JAX-RS module];
- A Spring based implementation, using the Spring `WebClient` API to perform invocations;

4.2. Configure defaults

The `ReactiveRestClient` API supports some **default** configuration attributes, which will be used for each request performed using a `ReactiveRestClient` instance:

- A **default target**, i.e. the default base URI which will be used for all the requests performed with the `ReactiveRestClient` API, unless overridden using the specific request configuration `target` method.
- A set of **default headers** to be included in all the requests performed with the `ReactiveRestClient` API.

```
ReactiveRestClient client = ReactiveRestClient.create();

client.defaultTarget(new URI("https://rest.api.example")); ①

client.withDefaultHeader(HttpHeaders.ACCEPT_LANGUAGE, "en-CA"); ②
client.withDefaultHeader(HttpHeaders.ACCEPT_CHARSET, "utf-8"); ③
```

① Set the default target request base URI, which will be used as target URI for every request configured using `request()`, if not overridden using `target(URI)`.

② Add a default request header which will be automatically added to every invocation request message

③ Add another default request header

4.3. Build and configure a request

To build a client request, the ReactiveRequestDefinition API is used, which represents both a *fluent* builder to configure the request message and a ReactiveInvocation API to perform the actual invocation and obtain a response.

The request can be configured using the `ReactiveInvocation` API methods as described below.

4.3.1. Request URI

The request URI can be composed using:

- A request **target**, i.e. the base URI of the request. If a *default* request target was configured for the `ReactiveRestClient` instance, it will be overriden by the specific request target.
- One ore more request *path*s, which will be appended to the base request target URI, adding *slash* characters to separate them from one another, if necessary.

```
ReactiveRestClient client = ReactiveRestClient.create();

ReactiveRequestDefinition request = client.request().target(URI.create(
"https://rest.api.example")); ①
request = request.path("apimethod"); ②
request = request.path("subpath"); ③
```

① Set the request *target*, i.e. the base request URI

② Set the request *path*, which will be appended to the base request URI

③ Append one more *path* to the request URI. The actual URI will be: `https://rest.api.example/apimethod/subpath`

4.3.2. URI *template* variable substitution values

The `ReactiveRestClient` API supports URI *template* variables substitution through the `resolve(…)` method.

IMPORTART: URI templates variables substitution is only supported for the request URI components specified as `path(…)` elements, not for the `target(…)` base URI part.

```
client.request().target("https://rest.api.example").path("/data/{name}/{id}").resolve(
"name", "test")
    .resolve("id", 123); ①

Map<String, Object> templates = new HashMap<>(1);
templates.put("id", "testValue");
request = client.request().target("https://rest.api.example").path("/test/{id}")
.resolve(templates); ②
```

① Subsitute two template variables values

② Subsitute template variables values using a name-value map

4.3.3. URI *query* parameters

The `ReactiveRestClient` API supports URI *query parameters* specification, with single or multiple values, through the `queryParameter(…)` methods.

```
client.request().queryParameter("parameter", "value") ①
    .queryParameter("multiValueParameter", 1, 2, 3); ②
```

① Set a single value query parameter

② Set a multiple values query parameter

4.3.4. Request headers

HTTP **headers** can be added to the request using the generic `header(String name, String… values)` method (supporting single or multiple header values) or a set of frequently used headers convenience setter methods, such as `accept`, `acceptLanguage` (supporting Java `Locale` types as arguments) and `cacheControl`.

The HttpHeaders interface can be used to refer to HTTP **header names** as constants.

The MediaType enumeration can be used for the `Accept` header values using the `accept(MediaType… mediaTypes)` builder method.

The CacheControl API provides a fluent builder to build and set a `Cache-Control` header value for the request, using the `cacheControl(CacheControl cacheControl)` builder method.

```
client.request().header("Accept", "text/plain"); ①
client.request().header(HttpHeaders.ACCEPT, "text/plain"); ②
client.request().accept("text/plain", "text/xml"); ③
client.request().accept(MediaType.APPLICATION_JSON); ④
client.request().acceptEncoding("gzip"); ⑤
client.request().acceptCharset("utf-8"); ⑥
client.request().acceptCharset(Charset.forName("utf-8")); ⑦
client.request().acceptLanguage("en-CA"); ⑧
client.request().acceptLanguage(Locale.US, Locale.GERMANY); ⑨
client.request().cacheControl(CacheControl.builder().noCache(true).noStore(true).build
()); ⑩
```

① Set a request header, providing its name and its value

② Set a request header, providing its name through the `HttpHeaders` enumeration and its value

③ Set the request `Accept` header values

④ Set the request `Accept` header value using the `MediaType` enumeration

⑤ Set the request `Accept-Encoding` header value

⑥ Set the request `Accept-Charset` header value

⑦ Set the request `Accept-Charset` header value using the Java `Charset` class

⑧ Set the request `Accept-Language` header value

⑨ Set the request `Accept-Language` header values using the Java `Locale` class

⑩ Build a `CacheControl` definition and set it as `Cache-Control` request header value

4.3.5. Authorization headers

The `ReactiveRestClient` API provides two convenience request builder methods to setup a request `Authorization` header using:

- The `Basic` authorization scheme, providing a *username* and a *password*, through the `authorizationBasic(String username, String password)` builder method.
- The `Bearer` authorization scheme, providing a *token*, through the `authorizationBearer(String bearerToken)` builder method.

```
client.request().authorizationBasic("username", "password"); ①
client.request().authorizationBearer("An389fz56xsr7"); ②
```

① Set the `Authorization` request header value using the `Basic` scheme and providing the credentials. Username and password will be encoded according to the HTTP specifications

② Set the `Authorization` request header value using the `Bearer` scheme and providing the bearer *token* value. See RFC6750

4.4. Invoke the request and obtain a response

The `ReactiveRequestDefinition` API can be used to perform the actual invocation and obtain a response.

The `ReactiveRequestDefinition` API provides a generic invocation method:

```
<T, R> Mono<ReactiveResponseEntity<T>> invoke(HttpMethod method, RequestEntity<R>
requestEntity, ResponseType<T> responseType)
```

This method requires the following parameters:

- The HTTP **method** to use to perform the request (`GET`, `POST`, and so on), specified using the `HttpMethod` enumeration.
- An optional **request entity**, i.e. the request message *payload* (body), represented through the `RequestEntity` API.
- The expected **response entity type** using the `ResponseType` class, to declare the Java type of the

response *payload* and apply a suitable converter, if available, to obtain the HTTP response body as the expected Java type.

The method returns a `Mono` of `ReactiveResponseEntity` type, which can be used to reactively handle the operation response.

The `ReactiveResponseEntity` API is a `ResponseEntity` extension which can be used to:

- Inspect the response message, for example to obtain the HTTP response **status** code, as a number or represented through the HttpStatus enumeration.
- Obtain the HTTP response raw *payload* or get it as a Java object, unmarshalled by a suitable converter which must be available from the concrete `ReactiveRestClient` API implementation.
- Obtain the response entity as a `Mono` or a `Flux` of the required type.

For non textual request or response payload types, any marshalling/unmarshalling strategy and implementation must be provided by the concrete `ReactiveRestClient` API. See the specific Available implementations documentation for additional information.

See the next sections for details about the invocation parameters and return types.

4.5. Request entity

The RequestEntity interface can be used to provide a *request entity* to the `ReactiveRestClient` API invocation methods, i.e. the request message *payload*.

The request *entity* is represented by a Java object and its serialization format is specified using a *media type* declaration (i.e. a **MIME** type definition) through the `Content-Type` request header value.

Depending on the `ReactiveRestClient` API implementation used, you must ensure the request media type is supported and a suitable request message body converter is available to deal with the Java object type and the media type of the request entity.

The `RequestEntity` interface provides a set of convenience static methods to build a request entity instance using the most common media types, such a `text/plain`, `application/json`, `application/xml` and `application/x-www-form-urlencoded` (the latter also providing a fluent *form* data builder method).

```
RequestEntity<String> request1 = RequestEntity.text("test"); ①

RequestEntity<TestData> request2 = RequestEntity.json(new TestData()); ②

RequestEntity request3 = RequestEntity
    .form(RequestEntity.formBuilder().set("value1", "one").set("value2", "a", "b")
.build()); ③
```

① Build a `text/plain` type request entity, using `test` as request entity value

② Build a `application/json` type request entity, using a `TestData` class instance as request entity value

③ Build a `application/x-www-form-urlencoded` type request entity, using the `formBuilder` method to build the *form* data map

The `RequestEntity.EMPTY` constant value can be used to provide an **empty** request entity.

```
RequestEntity<?> emptyRequest = RequestEntity.EMPTY; ①
```

① Build an empty request empty, to provide a request message without a payload

4.6. Response type

The ResponseType interface can be used to provide the expected response *entity* type to the `ReactiveRestClient` API invocation methods.

In addition to a simple Java class type, a *parametrized* type can be declared, allowing to use Java *generic* types as response types.

```
ResponseType<TestData> responseType1 = ResponseType.of(TestData.class); ①

ResponseType<List<TestData>> responseType2 = ResponseType.of(TestData.class, List
.class); ②
```

① Declares a response type as `TestData` type

② Declares a response type as a `List` of `TestData` types

4.7. Response entity

The ReactiveResponseEntity interface is used by `ReactiveRestClient` API to represent the invocation *response* and to deal with the response *entity* obtained as invocation result.

Since it is a `HttpResponse` instance, the `ReactiveRestClient` API can be used to inspect the response message, for example the HTTP message headers, including the HTTP status code.

```
Mono<ReactiveResponseEntity<TestData>> response = ReactiveRestClient
    .forTarget("https://rest.api.example/testget").request().accept(MediaType
.APPLICATION_JSON)
    .get(TestData.class); ①

response.doOnSuccess(r -> {
  HttpStatus status = r.getStatus(); ②
  int statusCode = r.getStatusCode(); ③
  long contentLength = r.getContentLength().orElse(-1L); ④
  Optional<String> value = r.getHeaderValue("HEADER_NAME"); ⑤
});
```

① Perform a `GET` request, setting the `Accept` header as `application/json` and declaring the `TestData` class as expected response entity Java type

② Get the response status as `HttpStatus` enumeration value

③ Get the response status code

④ Get the `Content-Length` header value

⑤ Get a generic header value

To obtain the response *entity* value as the expected Java type, the `asMono()` method can be used. The returned object generic type is provided according to the specified Response type, so the payload value will be and instance of the expected response Java type.

Furthermore, the `ReactiveResponseEntity` API makes available the `asMono(Class<E> entityType)` method, to obtain the response entity as a different type from the one specified with the Response type invocation parameter, if the media type is supported by the concrete `ReactiveRestClient` API implementation and a suitable converter is available.

In a similar way, the `asFlux(Class<E> entityType)` and `asInputStream()` methods provide the response entity content as a `Flux` and as a `InputStream` respectively.

```
Mono<ReactiveResponseEntity<TestData>> response = ReactiveRestClient
    .forTarget("https://rest.api.example/testget").request().accept(MediaType
.APPLICATION_JSON)
    .get(TestData.class); ①

response.doOnSuccess(r -> {
  Mono<TestData> entity = r.asMono(); ②
  Mono<String> asString = r.asMono(String.class); ③
});
```

① Perform a `GET` request, setting the `Accept` header as `application/json` and declaring the `TestData` class as expected response entity Java type

② Get the response entity `Mono` value as a `TestData` type, according to the declared response type

③ Get the response entity `Mono` value as a `String`

Depending on the concrete `ReactiveRestClient` API implementation, you must ensure the response media type is supported and a suitable message body converter is available to deal with the Java object type and the media type of the response entity.

4.8. Specific request invocation methods

In most cases, it is easier and faster to use HTTP *method*-specific invocation methods, made available by the `ReactiveRestClient` invocation API.

Each invocation method is relative to a specific HTTP request *method* and it is named accordingly. More than one method version is provided for each HTTP request method, providing the most suitable parameters and response types for for the most common situations.

For each HTTP request *method* (apart from the `HEAD` request method), the `ReactiveRestClient` API makes available a set of invocation methods organized as follows:

1. A set of methods to optionally provide a Request entity and to obtain a Response entity. If the response is expected to contain a payload which has to be deserialized into a Java object, the Response type can be specified, either as a simple or parametrized Java class.

```
final ReactiveRestClient client = ReactiveRestClient.forTarget(
"https://rest.api.example/test");

Mono<ReactiveResponseEntity<TestData>> response = client.request().get(TestData.class
); ①
response = client.request().get(ResponseType.of(TestData.class)); ②

response = client.request().put(RequestEntity.json(new TestData()), TestData.class);
③
```

① Perform an invocation using the `GET` method and obtain a `ResponseEntity` expecting the `TestData` class as response entity type

② The same invocation using the `ResponseType` API to specify the expected response entity type

③ Perform an invocation using the `PUT` method and providing an `application/json` type request entity, expecting a `TestData` response entity type

When a response entity is not expected, this category of invocation methods return a `Void` type `ReactiveResponseEntity`.

```
Mono<ReactiveResponseEntity<Void>> response2 = client.request().post(RequestEntity
.json(new TestData())); ①
response2.doOnSuccess(r -> {
  HttpStatus status = r.getStatus(); ②
});
```

① Perform an invocation using the `POST` method and providing an `application/json` type request

entity, but no response entity is expected

② Get the response HTTP status

2. A set of method to directly obtain the deserialized response *entity* value, named with the `ForEntity` suffix. This methods expects a *successful* response (i.e. a response with a `2xx` HTTP status code), otherwise an `UnsuccessfulResponseException` is thrown. The exception which can be inspected to obtain the response status code and the response itself.

```
Mono<TestData> value = client.request().getForEntity(TestData.class); ①
Mono<List<TestData>> values = client.request().getForEntity(ResponseType.of(TestData
.class, List.class)); ②
```

① Perform an invocation using the `GET` method and directly obtain the `TestData` type response entity value, if available

② Perform an invocation using the `GET` method and directly obtain a `List` of `TestData` type response entity values, if available

The UnsuccessfulResponseException type, which is thrown by the `xxxForEntity` invocation methods when the response status code do not belongs to the `2xx` family, provides some information about the invocation failure:

- The actual response **status** code.
- A reference to the actual response entity instance.

```
try {
  client.request().getForEntity(TestData.class);
} catch (UnsuccessfulResponseException e) {
  // got a response with a status code different from 2xx
  int httpStatusCode = e.getStatusCode(); ①
  Optional<HttpStatus> sts = e.getStatus(); ②
  ResponseEntity<?> theResponse = e.getResponse(); ③
}
```

① Get the actual response status code

② Get the response status code as a `HttpStatus`

③ Get the `ResponseEntity` instance

3. A set of convenience methods are provided for frequent needs and situations, for example:

- A `getForStream` method to perform a request using the HTTP `GET` method and obtain the response *entity* as an `InputStream`. This can be useful, for example, for API invocations which result is a stream of byte or characters.

```
Mono<InputStream> responseEntityStream = client.request().getForStream();
```

- A `getAsList` method, to perform a request using the HTTP `GET` method and obtain the response

entity contents as a `Flux` of deserialized Java objects in a specified expected response type.

```
Flux<TestData> collectionOfValues = client.request().getAsList(TestData.class);
```

- A `postForLocation` to perform a request using the HTTP `POST` and directly obtain the `Location` response header value as a Java `URI` instance, if available.

```
Mono<URI> locationHeaderURI = client.request().postForLocation(RequestEntity.json(new
TestData()));
```

4.9. `RestClient` API invocation methods reference

Below a reference list of the `ReactiveRestClient` invocation API, available from the reactive request definition API:

```
ReactiveRestClient reactiveRestClient = ReactiveRestClient.forTarget(
"http://api.example"); // Obtain a
                                                  // ReactiveRestClient
ReactiveRequestDefinition request = reactiveRestClient.request(); // Request
definition
```

Generic invocations:

Operation	Description	Parameters	Returns	Response status handling
`invoke`	Invoke the request and receive a response back.	1. HTTP method 2. Optional `RequestEntity` 3. Expected response entity type (`Void` for none)	A `Mono` of `ReactiveResponseEntity` type with the expected response entity payload type	*None*
`invokeForSuccess`	Invoke the request and receive a response back only if the response has a *success* (`2xx`) status code.	1. HTTP method 2. Optional `RequestEntity` 3. Expected response entity type (`Void` for none)	A `Mono` of `ReactiveResponseEntity` type with the provided response entity payload type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown

Operation	Description	Parameters	Returns	Response status handling
`invokeForEntity`	Invoke the request and receive back the response content entity, already deserialized in the expected response type.	1. HTTP method 2. Optional `RequestEntity` 3. Expected response entity type	A `Mono` with the response *entity* value, already deserialized in the expected response entity type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown
`invokeForFlux`	Invoke the request and receive back a response content entity, already deserialized in a `Flux` of the expected response type.	1. HTTP method 2. Optional `RequestEntity` 3. Expected response entity type	A `Flux` with the response *entity* values, already deserialized in the expected response entity type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown

By method invocations:

1. `GET`:

Operation	Parameters	Returns	Response status handling
`get`	Expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of `ReactiveResponseEntity<T>` type, with expected response entity payload type	*None*
`getForEntity`	Expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of the response *entity* value (`T`), already deserialized in the expected response entity type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown
`getForStream`	*None*	A `Mono` of the response payload `InputStream`	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown
`getAsList`	Expected response entity type (`Class<T>`)	A `Flux` of the deserialized response entities using the provided response entity type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown

2. `POST`:

Operation	First parameter	Second parameter	Returns	Response status handling
`post`	The request *entity* represented as `RequestEntity` instance	*Optional* expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of `ReactiveResponseEntity<T>` type, with expected response entity payload type. If the second parameter is not specified, a `Void` type `ReactiveResponseEntity` is returned	*None*
`postForEntity`	The request *entity* represented as `RequestEntity` instance	Expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of the response *entity* value (`T`), already deserialized in the expected response entity type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown
`postForLocation`	The request *entity* represented as `RequestEntity` instance	*None*	A `Mono` of the `Location` response header value	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown

3. `PUT`:

Operation	First parameter	Second parameter	Returns	Response status handling
`put`	The request *entity* represented as `RequestEntity` instance	*Optional* expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of `ReactiveResponseEntity<T>` type, with expected response entity payload type. If the second parameter is not specified, a `Void` type `ReactiveResponseEntity` is returned	*None*

Operation	First parameter	Second parameter	Returns	Response status handling
`putForEntity`	The request *entity* represented as `RequestEntity` instance	Expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of the response *entity* value (`T`), already deserialized in the expected response entity type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown

4. `PATCH`:

Operation	First parameter	Second parameter	Returns	Response status handling
`patch`	The request *entity* represented as `RequestEntity` instance	*Optional* expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of `ReactiveResponseEntity<T>` type, with expected response entity payload type. If the second parameter is not specified, a `Void` type `ReactiveResponseEntity` is returned	*None*
`patchForEntity`	The request *entity* represented as `RequestEntity` instance	Expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of the response *entity* value (`T`), already deserialized in the expected response entity type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown

5. `DELETE`:

Operation	Parameter	Returns	Response status handling
`delete`	*Optional* expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of `ReactiveResponseEntity<Void>` type	*None*
`deleteOrFail`	*None*	A `Mono` of `Void` type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown

Operation	Parameter	Returns	Response status handling
`deleteForEntity`	Expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of the response *entity* value (`T`), already deserialized in the expected response entity type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown

6. `OPTIONS`:

Operation	Parameter	Returns	Response status handling
`options`	*Optional* expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of `ReactiveResponseEntity<T>` type, with expected response entity payload type. If the second parameter is not specified, a `Void` type `ReactiveResponseEntity` is returned	*None*
`optionsForEntity`	Expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of the response *entity* value (`T`), already deserialized in the expected response entity type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown

7. `TRACE`:

Operation	Parameter	Returns	Response status handling
`trace`	*Optional* expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of `ReactiveResponseEntity<T>` type, with expected response entity payload type. If the second parameter is not specified, a `Void` type `ReactiveResponseEntity` is returned	*None*
`traceForEntity`	Expected response entity type, either using a `Class<T>` or a `ResponseType<T>` to handle generic types	A `Mono` of the response *entity* value (`T`), already deserialized in the expected response entity type	If the response status code is not `2xx`, an `UnsuccessfulResponseException` is thrown

8. `HEAD`:

Operation	Returns	Response status handling
`head`	A `Void` type `ResponseEntity`	A `Mono` of `ReactiveResponseEntity<Void>` type

4.9.1. `Property` and `PropertyBox` support

The `ReactiveRestClient` API fully supports the Holon Platform Property model when used along with the PropertyBox data type as a request/response *entity* in RESTful API calls.

Regarding the **JSON** media type, the `PropertyBox` type marshalling and unmarshalling support is provided by the Holon Platform JSON module. For the builtin `ReactiveRestClient` API implementations, the `PropertyBox` type JSON support is automatically setted up when the suitable Holon platform JSON module artifacts are available in classpath.

When a response entity value has to be deserialized into a `PropertyBox` object type, the **property set** to be used must be specified along with the reponse entity type, in order to instruct the JSON module unmarshallers about the property set with which to build the response `PropertyBox` instances.

For this purpose, the `ReactiveRestClient` invocation API `propertySet(…)` methods can be used to specify the **property set** with which to obtain a `PropertyBox` type response entity value.

```
final PathProperty<Integer> CODE = create("code", int.class);
final PathProperty<String> VALUE = create("value", String.class);
final PropertySet<?> PROPERTIES = PropertySet.of(CODE, VALUE);

ReactiveRestClient client = ReactiveRestClient.create();

Mono<PropertyBox> box = client.request().target("https://rest.api.example").path(
"/apimethod")
    .propertySet(PROPERTIES).getForEntity(PropertyBox.class); ①

Mono<PropertyBox> box2 = client.request().target("https://rest.api.example").path(
"/apimethod")
    .propertySet(CODE, VALUE).getForEntity(PropertyBox.class); ②

Flux<PropertyBox> boxes = client.request().target("https://rest.api.example").path(
"/apimethod")
    .propertySet(PROPERTIES).getAsList(PropertyBox.class); ③
```

① `GET` request for a `PropertyBox` type `Mono` response, using `PROPERTIES` as property set

② `GET` request for a `PropertyBox` type `Mono` response, using directly an array of properties

③ `GET` request for a `Flux` of `PropertyBox` type response, using `PROPERTIES` as property set

5. `ReactiveRestClient` implementation using the Spring `WebClient` API

Maven coordinates:

```
<groupId>com.holon-platform.reactor</groupId>
<artifactId>holon-reactor-spring</artifactId>
<version>5.3.0</version>
```

The `holon-reactor-spring` artifact provides a Reactive RestClient implementation using the Spring 5+ `WebClient` API.

The Spring `ReactiveRestClient` implementation is represented by the SpringReactiveRestClient interface, which provides a `create(WebClient webClient)` method to create a `ReactiveRestClient` instance using a provided Spring `WebClient` reference.

```
WebClient webClient = getWebClient(); ①

ReactiveRestClient client = SpringReactiveRestClient.create(webClient); ②
```

① Create or obtain a `WebClient` implementation

② Create a `ReactiveRestClient` using the `WebClient` implementation

When a `WebClient` instance is available as a Holon Platform [Context] resource, a `ReactiveRestClientFactory` is automatically registered to provide a `SpringReactiveRestClient` implementation using that `WebClient` implementation. This way, the default `ReactiveRestClient.create(…)` static methods can be used to obtain a `ReactiveRestClient` implementation.

If the Spring context scope is enabled with the default beans lookup strategy, it is sufficient that a `WebClient` bean type is registered in the Spring application context to obtain it as a *context resource*.

```
@Configuration
@EnableBeanContext ①
class Config {

  @Bean ②
  public WebClient webClient() {
    return WebClient.create();
  }

}

void restclient() {
  ReactiveRestClient client = ReactiveRestClient.create(); ③

  client = ReactiveRestClient.create(SpringReactiveRestClient.class.getName()); ④
}
```

① Use the `@EnableBeanContext` to enable Spring beans context

② Provide a `WebClient` bean definition

③ The `ReactiveRestClient.create()` method can be used to obtain a `ReactiveRestClient` implementation backed by the defined `WebClient` bean definition

④ If more than one `ReactiveRestClientFactory` is available, the `SpringReactiveRestClient` class name can be specified to ensure that a `SpringReactiveRestClient` type is obtained as a `ReactiveRestClient` implementation