Spring REST Docs

Andy Wilkinson
Version 2.0.2.RELEASE

Table of Contents

Document RESTful services by combining hand-written documentation with auto-generated snippets produced with Spring MVC Test.

Introduction

The aim of Spring REST Docs is to help you to produce documentation for your RESTful services that is accurate and readable.

Writing high-quality documentation is difficult. One way to ease that difficulty is to use tools that are well-suited to the job. To this end, Spring REST Docs uses Asciidoctor (http://asciidoctor.org) by default. Asciidoctor processes plain text and produces HTML, styled and layed out to suit your needs. If you prefer, Spring REST Docs can also be configured to use Markdown.

Spring REST Docs makes use of snippets produced by tests written with Spring MVC's test framework (https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/testing.html#spring-mvc-test-framework), Spring WebFlux's `WebTestClient` (https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/testing.html#webtestclient) or REST Assured 3 (http://www.rest-assured.io). This test-driven approach helps to guarantee the accuracy of your service's documentation. If a snippet is incorrect the test that produces it will fail.

Documenting a RESTful service is largely about describing its resources. Two key parts of each resource's description are the details of the HTTP requests that it consumes and the HTTP responses that it produces. Spring REST Docs allows you to work with these resources and the HTTP requests and responses, shielding your documentation from the inner-details of your service's implementation. This separation helps you to document your service's API rather than its implementation. It also frees you to evolve the implementation without having to rework the documentation.

Getting started

This section describes how to get started with Spring REST Docs.

Sample applications

If you want to jump straight in, a number of sample applications are available:

Table 1. MockMvc

Sample	Build system	Description
Spring Data REST (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/rest-notes-spring-data-rest)	Maven	Demonstrates the creation of a getting started guide and an API guide for a service implemented using Spring Data REST (https://projects.spring.io/spring-data-rest/) .
Spring HATEOAS (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/rest-notes-spring-hateoas)	Gradle	Demonstrates the creation of a getting started guide and an API guide for a service implemented using Spring HATEOAS (https://projects.spring.io/spring-hateoas/).

Table 2. WebTestClient

Sample	Build system	Description
WebTestClient (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/web-test-client)	Gradle	Demonstrates the use of Spring REST docs with Spring WebFlux's WebTestClient.

Table 3. REST Assured

Sample	Build system	Description

Sample	Build system	Description
Grails (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/rest-notes-grails)	Gradle	Demonstrates the use of Spring REST docs with Grails (https://grails.org) and Spock (https://github.com/spockframework/spock).
REST Assured (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/rest-assured)	Gradle	Demonstrates the use of Spring REST Docs with REST Assured (http://rest-assured.io).

Table 4. Advanced

Sample	Build system	Description
Slate (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/rest-notes-slate)	Gradle	Demonstrates the use of Spring REST Docs with Markdown and Slate (https://github.com/tripit/slate).
TestNG (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/testng)	Gradle	Demonstrates the use of Spring REST Docs with TestNG (http://testng.org).
JUnit 5 (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/junit5)	Gradle	Demonstrates the use of Spring REST Docs with JUnit 5 (http://junit.org/junit5/).

Requirements

Spring REST Docs has the following minimum requirements:

- Java 8
- Spring Framework 5 (5.0.2 or later)

Additionally, the `spring-restdocs-restassured` module has the following minimum requirements:

- REST Assured 3.0

Build configuration

The first step in using Spring REST Docs is to configure your project's build. The Spring HATEOAS (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/rest-notes-spring-hateoas) and Spring Data REST (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/rest-notes-spring-data-rest) samples contain a `build.gradle` and `pom.xml` respectively that you may wish to use as a reference. The key parts of the configuration are described below.

Maven | Gradle

```xml
<dependency> 1
    <groupId>org.springframework.restdocs</groupId>
    <artifactId>spring-restdocs-mockmvc</artifactId>
    <version>2.0.2.RELEASE</version>
    <scope>test</scope>
</dependency>

<build>
    <plugins>
        <plugin> 2
            <groupId>org.asciidoctor</groupId>
            <artifactId>asciidoctor-maven-plugin</artifactId>
            <version>1.5.3</version>
            <executions>
                <execution>
                    <id>generate-docs</id>
                    <phase>prepare-package</phase> 3
                    <goals>
                        <goal>process-asciidoc</goal>
                    </goals>
                    <configuration>
                        <backend>html</backend>
                        <doctype>book</doctype>
                    </configuration>
                </execution>
            </executions>
            <dependencies>
                <dependency> 4
                    <groupId>org.springframework.restdocs</groupId>
                    <artifactId>spring-restdocs-asciidoctor</artifactId>
                    <version>2.0.2.RELEASE</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
```

1. Add a dependency on `spring-restdocs-mockmvc` in the `test` scope. If you want to use `WebTestClient` or REST Assured rather than MockMvc, add a dependency on `spring-restdocs-webtestclient` or `spring-restdocs-restassured` respectively instead.
2. Add the Asciidoctor plugin.
3. Using `prepare-package` allows the documentation to be included in the package.
4. Add `spring-restdocs-asciidoctor` as a dependency of the Asciidoctor plugin. This will automatically configure the `snippets` attribute for use in your `.adoc` files to point to `target/generated-snippets`. It will also allow you to use the `operation` block macro.

Packaging the documentation

You may want to package the generated documentation in your project's jar file, for example to have it served as static content (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-spring-mvc-static-content) by Spring Boot. To do so, configure your project's build so that:

1. The documentation is generated before the jar is built
2. The generated documentation is included in the jar

Maven | Gradle

XML

```
<plugin>  1
    <groupId>org.asciidoctor</groupId>
    <artifactId>asciidoctor-maven-plugin</artifactId>
    <!-- … -->
</plugin>
<plugin>  2
    <artifactId>maven-resources-plugin</artifactId>
    <version>2.7</version>
    <executions>
        <execution>
            <id>copy-resources</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>copy-resources</goal>
            </goals>
            <configuration>  3
                <outputDirectory>
                    ${project.build.outputDirectory}/static/docs
                </outputDirectory>
                <resources>
                    <resource>
                        <directory>
                            ${project.build.directory}/generated-docs
                        </directory>
                    </resource>
                </resources>
            </configuration>
        </execution>
    </executions>
</plugin>
```

1 The existing declaration for the Asciidoctor plugin.

2 The resource plugin must be declared after the Asciidoctor plugin as they are bound to the same phase (`prepare-package`) and the resource plugin must run after the Asciidoctor plugin to ensure that the documentation is generated before it's copied.

3. Copy the generated documentation into the build output's `static/docs` directory, from where it will be included in the jar file.

Generating documentation snippets

Spring REST Docs uses Spring MVC's test framework (https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/testing.html#spring-mvc-test-framework), Spring WebFlux's `WebTestClient` (https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/testing.html#webtestclient) or REST Assured (http://www.rest-assured.io) to make requests to the service that you are documenting. It then produces documentation snippets for the request and the resulting response.

Setting up your tests

Exactly how you setup your tests depends on the test framework that you're using. Spring REST Docs provides first-class support for JUnit 4 and JUnit 5. Other frameworks, such as TestNG, are also supported although slightly more setup is required.

Setting up your JUnit 4 tests

When using JUnit 4, the first step in generating documentation snippets is to declare a `public JUnitRestDocumentation` field that's annotated as a JUnit `@Rule`.

JAVA

```
@Rule
public JUnitRestDocumentation restDocumentation = new JUnitRestDocumentation();
```

By default, the `JUnitRestDocumentation` rule is automatically configured with an output directory based on your project's build tool:

Build tool	Output directory
Maven	`target/generated-snippets`
Gradle	`build/generated-snippets`

The default can be overridden by providing an output directory when creating the `JUnitRestDocumentation` instance:

JAVA

```
@Rule
public JUnitRestDocumentation restDocumentation = new JUnitRestDocumentation("custom");
```

Next, provide an `@Before` method to configure MockMvc, WebTestClient or REST Assured:

MockMvc | WebTestClient | REST Assured

JAVA

```
private MockMvc mockMvc;

@Autowired
private WebApplicationContext context;

@Before
public void setUp() {
    this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context)
            .apply(documentationConfiguration(this.restDocumentation))  1
            .build();
}
```

1 The `MockMvc` instance is configured using a `MockMvcRestDocumentationConfigurer`. An instance of this class can be obtained from the static `documentationConfiguration()` method on `org.springframework.restdocs.mockmvc.MockMvcRestDocumentation`.

The configurer applies sensible defaults and also provides an API for customizing the configuration. Refer to the configuration section for more information.

Setting up your JUnit 5 tests

When using JUnit 5, the first step in generating documentation snippets is to apply the `RestDocumentationExtension` to your test class:

JAVA

```
@ExtendWith(RestDocumentationExtension.class)
public class JUnit5ExampleTests {
```

For testing a typical Spring application the `SpringExtension` should also be applied:

JAVA

```
@ExtendWith({RestDocumentationExtension.class, SpringExtension.class})
public class JUnit5ExampleTests {
```

The `RestDocumentationExtension` is automatically configured with an output directory based on your project's build tool:

Build tool	Output directory
Maven	`target/generated-snippets`
Gradle	`build/generated-snippets`

Next, provide a `@BeforeEach` method to configure MockMvc, WebTestClient, or REST Assured:

MockMvc | WebTestClient | REST Assured

```
private MockMvc mockMvc;

@BeforeEach
public void setUp(WebApplicationContext webApplicationContext,
        RestDocumentationContextProvider restDocumentation) {
    this.mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext)
            .apply(documentationConfiguration(restDocumentation))  1
            .build();
}
```

1 The `MockMvc` instance is configured using a `MockMvcRestDocumentationConfigurer`. An instance of this class can be obtained from the static `documentationConfiguration()` method on `org.springframework.restdocs.mockmvc.MockMvcRestDocumentation`.

The configurer applies sensible defaults and also provides an API for customizing the configuration. Refer to the configuration section for more information.

Setting up your tests without JUnit

The configuration when JUnit is not being used is largely similar to when it is being used. This section describes the key differences. The TestNG sample (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/testng) also illustrates the approach.

The first difference is that `ManualRestDocumentation` should be used in place of `JUnitRestDocumentation` and there's no need for the `@Rule` annotation:

```
private ManualRestDocumentation restDocumentation = new ManualRestDocumentation();
```

Secondly, `ManualRestDocumentation.beforeTest(Class, String)` must be called before each test. This can be done as part of the method that is configuring MockMvc, WebTestClient, or REST Assured:

MockMvc | WebTestClient | REST Assured

JAVA

```
private MockMvc mockMvc;

@Autowired
private WebApplicationContext context;

@BeforeMethod
public void setUp(Method method) {
    this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context)
            .apply(documentationConfiguration(this.restDocumentation))
            .build();
    this.restDocumentation.beforeTest(getClass(), method.getName());
}
```

Lastly, `ManualRestDocumentation.afterTest` must be called after each test. For example, with TestNG:

JAVA

```
@AfterMethod
public void tearDown() {
    this.restDocumentation.afterTest();
}
```

Invoking the RESTful service

Now that the testing framework has been configured, it can be used to invoke the RESTful service and document the request and response. For example:

MockMvc | WebTestClient | REST Assured

JAVA

```
this.mockMvc.perform(get("/").accept(MediaType.APPLICATION_JSON))  1
    .andExpect(status().isOk())  2
    .andDo(document("index"));  3
```

1. Invoke the root (`/`) of the service and indicate that an `application/json` response is required.
2. Assert that the service produced the expected response.
3. Document the call to the service, writing the snippets into a directory named `index` that will be located beneath the configured output directory. The snippets are written by a `RestDocumentationResultHandler`. An instance of this class can be obtained from the static `document` method on `org.springframework.restdocs.mockmvc.MockMvcRestDocumentation`.

By default, six snippets are written:

- `<output-directory>/index/curl-request.adoc`
- `<output-directory>/index/http-request.adoc`
- `<output-directory>/index/http-response.adoc`
- `<output-directory>/index/httpie-request.adoc`
- `<output-directory>/index/request-body.adoc`
- `<output-directory>/index/response-body.adoc`

Refer to Documenting your API for more information about these and other snippets that can be produced by Spring REST Docs.

Using the snippets

Before using the generated snippets, a `.adoc` source file must be created. You can name the file whatever you like as long as it has a `.adoc` suffix. The result HTML file will have the same name but with a `.html` suffix. The default location of the source files and the resulting HTML files depends on whether you are using Maven or Gradle:

Build tool	Source files	Generated files
Maven	`src/main/asciidoc/*.adoc`	`target/generated-docs/*.html`
Gradle	`src/docs/asciidoc/*.adoc`	`build/asciidoc/html5/*.html`

The generated snippets can then be included in the manually created Asciidoctor file from above using the include macro (http://asciidoctor.org/docs/asciidoc-syntax-quick-reference/#include-files). The `snippets` attribute that is automatically set by `spring-restdocs-asciidoctor` configured in the build configuration can be used to reference the snippets output directory. For example:

```
include::{snippets}/index/curl-request.adoc[]
```
ADOC

Documenting your API

This section provides more details about using Spring REST Docs to document your API.

Hypermedia

Spring REST Docs provides support for documenting the links in a Hypermedia-based (https://en.wikipedia.org/wiki/HATEOAS) API:

MockMvc | WebTestClient | REST Assured

JAVA

```
this.mockMvc.perform(get("/").accept(MediaType.APPLICATION_JSON))
    .andExpect(status().isOk())
    .andDo(document("index", links(  1
            linkWithRel("alpha").description("Link to the alpha resource"),  2
            linkWithRel("bravo").description("Link to the bravo resource"))));  3
```

1. Configure Spring REST docs to produce a snippet describing the response's links. Uses the static `links` method on `org.springframework.restdocs.hypermedia.HypermediaDocumentation`.
2. Expect a link whose rel is `alpha`. Uses the static `linkWithRel` method on `org.springframework.restdocs.hypermedia.HypermediaDocumentation`.
3. Expect a link whose rel is `bravo`.

The result is a snippet named `links.adoc` that contains a table describing the resource's links.

If a link in the response has a `title`, the description can be omitted from its descriptor and the `title` will be used. If you omit the description and the link does not have a `title` a failure will occur.

When documenting links, the test will fail if an undocumented link is found in the response. Similarly, the test will also fail if a documented link is not found in the response and the link has not been marked as optional.

If you do not want to document a link, you can mark it as ignored. This will prevent it from appearing in the generated snippet while avoiding the failure described above.

Links can also be documented in a relaxed mode where any undocumented links will not cause a test failure. To do so, use the `relaxedLinks` method on `org.springframework.restdocs.hypermedia.HypermediaDocumentation`. This can be useful when documenting a particular scenario where you only want to focus on a subset of the links.

Hypermedia link formats

Two link formats are understood by default:

- Atom – links are expected to be in an array named `links`. Used by default when the content type of the response is compatible with `application/json`.
- HAL – links are expected to be in a map named `_links`. Used by default when the content type of the response is compatible with `application/hal+json`.

If you are using Atom or HAL-format links but with a different content type you can provide one of the built-in `LinkExtractor` implementations to `links`. For example:

MockMvc | WebTestClient | REST Assured

JAVA

```
.andDo(document("index", links(halLinks(),  1
        linkWithRel("alpha").description("Link to the alpha resource"),
        linkWithRel("bravo").description("Link to the bravo resource"))));
```

1 Indicate that the links are in HAL format. Uses the static `halLinks` method on `org.springframework.restdocs.hypermedia.HypermediaDocumentation`.

If your API represents its links in a format other than Atom or HAL, you can provide your own implementation of the `LinkExtractor` interface to extract the links from the response.

Ignoring common links

Rather than documenting links that are common to every response, such as `self` and `curies` when using HAL, you may want to document them once in an overview section and then ignore them in the rest of your API's documentation. To do so, you can build on the support for reusing snippets to add link descriptors to a snippet that's preconfigured to ignore certain links. For example:

JAVA

```
public static LinksSnippet links(LinkDescriptor... descriptors) {
    return HypermediaDocumentation.links(linkWithRel("self").ignored().optional(),
            linkWithRel("curies").ignored()).and(descriptors);
}
```

Request and response payloads

In addition to the hypermedia-specific support described above, support for general documentation of request and response payloads is also provided.

By default, Spring REST Docs will automatically generate snippets for the body of the request and the body of the response. These snippets are named `request-body.adoc` and `response-body.adoc` respectively.

Request and response fields

To provide more detailed documentation of a request or response payload, support for documenting the payload's fields is provided.

Consider the following payload:

```
{
    "contact": {
        "name": "Jane Doe",
        "email": "jane.doe@example.com"
    }
}
```
JSON

Its fields can be documented like this:

MockMvc | WebTestClient | REST Assured

```
this.mockMvc.perform(get("/user/5").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andDo(document("index",
                responseFields(  1
                        fieldWithPath("contact.email")
                                .description("The user's email address"),  2
                fieldWithPath("contact.name").description("The user's name"))));  3
```
JAVA

1. Configure Spring REST docs to produce a snippet describing the fields in the response payload. To document a request `requestFields` can be used. Both are static methods on `org.springframework.restdocs.payload.PayloadDocumentation`.
2. Expect a field with the path `contact.email`. Uses the static `fieldWithPath` method on `org.springframework.restdocs.payload.PayloadDocumentation`.
3. Expect a field with the path `contact.name`.

The result is a snippet that contains a table describing the fields. For requests this snippet is named `request-fields.adoc`. For responses this snippet is named `response-fields.adoc`.

When documenting fields, the test will fail if an undocumented field is found in the payload. Similarly, the test will also fail if a documented field is not found in the payload and the field has not been marked as optional.

If you don't want to provide detailed documentation for all of the fields, an entire subsection of a payload can be documented. For example:

MockMvc | WebTestClient | REST Assured

JAVA

```
this.mockMvc.perform(get("/user/5").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andDo(document("index",
                responseFields(  1
                        subsectionWithPath("contact")
                                .description("The user's contact details"))));  1
```

1 Document the subsection with the path `contact`. `contact.email` and `contact.name` are now seen as having also been documented. Uses the static `subsectionWithPath` method on `org.springframework.restdocs.payload.PayloadDocumentation`.

`subsectionWithPath` can be useful for providing a high-level overview of a particular section of a payload. Separate, more detailed documentation for a subsection can then be produced.

If you do not want to document a field or subsection at all, you can mark it as ignored. This will prevent it from appearing in the generated snippet while avoiding the failure described above.

Fields can also be documented in a relaxed mode where any undocumented fields will not cause a test failure. To do so, use the `relaxedRequestFields` and `relaxedResponseFields` methods on `org.springframework.restdocs.payload.PayloadDocumentation`. This can be useful when documenting a particular scenario where you only want to focus on a subset of the payload.

By default, Spring REST Docs will assume that the payload you are documenting is JSON. If you want to document an XML payload the content type of the request or response must be compatible with `application/xml`.

Fields in JSON payloads

JSON field paths

JSON field paths use either dot notation or bracket notation. Dot notation uses '.' to separate each key in the path; `a.b`, for example. Bracket notation wraps each key in square brackets and single quotes; `['a']['b']`, for example. In either case, `[]` is used to identify an array. Dot notation is more concise, but using bracket notation enables the use of `.` within a key name; `['a.b']`, for example. The two different notations can be used in the same path; `a['b']`, for example.

With this JSON payload:

```JSON
{
    "a":{
        "b":[
            {
                "c":"one"
            },
            {
                "c":"two"
            },
            {
                "d":"three"
            }
        ],
        "e.dot" : "four"
    }
}
```

The following paths are all present:

Path	Value
`a`	An object containing `b`
`a.b`	An array containing three objects
`['a']['b']`	An array containing three objects
`a['b']`	An array containing three objects
`['a'].b`	An array containing three objects
`a.b[]`	An array containing three objects
`a.b[].c`	An array containing the strings `one` and `two`

Path	Value
`a.b[].d`	The string `three`
`a['e.dot']`	The string `four`
`['a']['e.dot']`	The string `four`

A payload that uses an array at its root can also be documented. The path `[]` will refer to the entire array. You can then use bracket or dot notation to identify fields within the array's entries. For example, `[].id` corresponds to the `id` field of every object found in the following array:

```JSON
[
    {
        "id":1
    },
    {
        "id":2
    }
]
```

You can use `*` as a wildcard to match fields with different names. For example, `users.*.role` could be used to document the role of every user in the following JSON:

```JSON
{
    "users":{
        "ab12cd34":{
            "role": "Administrator"
        },
        "12ab34cd":{
            "role": "Guest"
        }
    }
}
```

JSON field types

When a field is documented, Spring REST Docs will attempt to determine its type by examining the payload. Seven different types are supported:

Type	Description

Type	Description
array	The value of each occurrence of the field is an array
boolean	The value of each occurrence of the field is a boolean (`true` or `false`)
object	The value of each occurrence of the field is an object
number	The value of each occurrence of the field is a number
null	The value of each occurrence of the field is `null`
string	The value of each occurrence of the field is a string
varies	The field occurs multiple times in the payload with a variety of different types

The type can also be set explicitly using the `type(Object)` method on `FieldDescriptor` . The result of the supplied `Object` 's `toString` method will be used in the documentation. Typically, one of the values enumerated by `JsonFieldType` will be used:

MockMvc | WebTestClient | REST Assured

JAVA

```
.andDo(document("index",
        responseFields(
                fieldWithPath("contact.email").type(JsonFieldType.STRING)  1
                        .description("The user's email address"))));
```

1 Set the field's type to `String` .

XML payloads

XML field paths

XML field paths are described using XPath. `/` is used to descend into a child node.

XML field types

When documenting an XML payload, you must provide a type for the field using the `type(Object)` method on `FieldDescriptor` . The result of the supplied type's `toString` method will be used in the documentation.

Reusing field descriptors

In addition to the general support for reusing snippets, the request and response snippets allow additional descriptors to be configured with a path prefix. This allows the descriptors for a repeated portion of a request or response payload to be created once and then reused.

Consider an endpoint that returns a book:

JSON

```
{
    "title": "Pride and Prejudice",
    "author": "Jane Austen"
}
```

The paths for `title` and `author` are simply `title` and `author` respectively.

Now consider an endpoint that returns an array of books:

JSON

```
[{
    "title": "Pride and Prejudice",
    "author": "Jane Austen"
},
{
    "title": "To Kill a Mockingbird",
    "author": "Harper Lee"
}]
```

The paths for `title` and `author` are `[].title` and `[].author` respectively. The only difference between the single book and the array of books is that the fields' paths now have a `[].` prefix.

The descriptors that document a book can be created:

JAVA

```
FieldDescriptor[] book = new FieldDescriptor[] {
        fieldWithPath("title").description("Title of the book"),
        fieldWithPath("author").description("Author of the book") };
```

They can then be used to document a single book:

MockMvc | WebTestClient | REST Assured

JAVA

```
this.mockMvc.perform(get("/books/1").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk()).andDo(document("book", responseFields(book))); 1
```

1 Document `title` and `author` using existing descriptors

And an array of books:

MockMvc | WebTestClient | REST Assured

JAVA
```
this.mockMvc.perform(get("/books").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andDo(document("book",
                responseFields(
                        fieldWithPath("[]").description("An array of books"))  1
                                .andWithPrefix("[].", book)));  2
```

1 Document the array

2 Document `[].title` and `[].author` using the existing descriptors prefixed with `[].`

Documenting a subsection of a request or response payload

If a payload is large or structurally complex, it can be useful to document individual sections of the payload. REST Docs allows you to do so by extracting a subsection of the payload and then documenting it.

Documenting a subsection of a request or response body

Consider the following JSON response body:

JSON
```
{
    "weather": {
        "wind": {
            "speed": 15.3,
            "direction": 287.0
        },
        "temperature": {
            "high": 21.2,
            "low": 14.8
        }
    }
}
```

A snippet that documents the `temperature` object can be produces as follows:

MockMvc | WebTestClient | REST Assured

JAVA
```
this.mockMvc.perform(get("/locations/1").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk()).andDo(document("location",
                responseBody(beneathPath("weather.temperature"))));  1
```

1 Produce a snippet containing a subsection of the response body. Uses the static `responseBody` and `beneathPath` methods on `org.springframework.restdocs.payload.PayloadDocumentation`. To produce a snippet for the request body, `requestBody` can be used in place of `responseBody`.

The result is a snippet with the following contents:

```
{
    "temperature": {
        "high": 21.2,
        "low": 14.8
    }
}
```
JSON

To make the snippet's name distinct, an identifier for the subsection is included. By default, this identifier is `beneath-${path}`. For example, the code above will result in a snippet named `response-body-beneath-weather.temperature.adoc`. The identifier can be customized using the `withSubsectionId(String)` method:

```
responseBody(beneathPath("weather.temperature").withSubsectionId("temp"));
```
JAVA

This example will result in a snippet named `request-body-temp.adoc`.

Documenting the fields of a subsection of a request or response

As well as documenting a subsection of a request or response body, it's also possible to document the fields in a particular subsection. A snippet that documents the fields of the `temperature` object (`high` and `low`) can be produced as follows:

MockMvc | WebTestClient | REST Assured

```
this.mockMvc.perform(get("/locations/1").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andDo(document("location",
                responseFields(beneathPath("weather.temperature"),  1
                        fieldWithPath("high").description(
                                "The forecast high in degrees celcius"),  2
                fieldWithPath("low")
                        .description("The forecast low in degrees celcius"))));
```
JAVA

1. Produce a snippet describing the fields in the subsection of the response payload beneath the path `weather.temperature`. Uses the static `beneathPath` method on `org.springframework.restdocs.payload.PayloadDocumentation`.
2. Document the `high` and `low` fields.

The result is a snippet that contains a table describing the `high` and `low` fields of `weather.temperature`. To make the snippet's name distinct, an identifier for the subsection is included. By default, this identifier is `beneath-${path}`. For example, the code above will result in a snippet named `response-fields-beneath-weather.temperature.adoc`.

Request parameters

A request's parameters can be documented using `requestParameters`. Request parameters can be included in a `GET` request's query string. For example:

MockMvc | WebTestClient | REST Assured

```java
this.mockMvc.perform(get("/users?page=2&per_page=100")) 1
    .andExpect(status().isOk())
    .andDo(document("users", requestParameters( 2
            parameterWithName("page").description("The page to retrieve"), 3
            parameterWithName("per_page").description("Entries per page") 4
    )));
```

1. Perform a `GET` request with two parameters, `page` and `per_page` in the query string.
2. Configure Spring REST Docs to produce a snippet describing the request's parameters. Uses the static `requestParameters` method on `org.springframework.restdocs.request.RequestDocumentation`.
3. Document the `page` parameter. Uses the static `parameterWithName` method on `org.springframework.restdocs.request.RequestDocumentation`.
4. Document the `per_page` parameter.

Request parameters can also be included as form data in the body of a POST request:

MockMvc | WebTestClient | REST Assured

```java
this.mockMvc.perform(post("/users").param("username", "Tester")) 1
    .andExpect(status().isCreated())
    .andDo(document("create-user", requestParameters(
            parameterWithName("username").description("The user's username")
    )));
```

1 Perform a `POST` request with a single parameter, `username`.

In both cases, the result is a snippet named `request-parameters.adoc` that contains a table describing the parameters that are supported by the resource.

When documenting request parameters, the test will fail if an undocumented request parameter is used in the request. Similarly, the test will also fail if a documented request parameter is not found in the request and the request parameter has not been marked as optional.

If you do not want to document a request parameter, you can mark it as ignored. This will prevent it from appearing in the generated snippet while avoiding the failure described above.

Request parameters can also be documented in a relaxed mode where any undocumented parameters will not cause a test failure. To do so, use the `relaxedRequestParameters` method on `org.springframework.restdocs.request.RequestDocumentation`. This can be useful when documenting a particular scenario where you only want to focus on a subset of the request parameters.

Path parameters

A request's path parameters can be documented using `pathParameters`. For example:

MockMvc | WebTestClient | REST Assured

```java
this.mockMvc.perform(get("/locations/{latitude}/{longitude}", 51.5072, 0.1275)) 1
    .andExpect(status().isOk())
    .andDo(document("locations", pathParameters( 2
            parameterWithName("latitude").description("The location's latitude"), 3
            parameterWithName("longitude").description("The location's longitude") 4
    )));
```

1 Perform a `GET` request with two path parameters, `latitude` and `longitude`.

2 Configure Spring REST Docs to produce a snippet describing the request's path parameters. Uses the static `pathParameters` method on `org.springframework.restdocs.request.RequestDocumentation`.

3 Document the parameter named `latitude`. Uses the static `parameterWithName` method on `org.springframework.restdocs.request.RequestDocumentation`.

4 Document the parameter named `longitude`.

The result is a snippet named `path-parameters.adoc` that contains a table describing the path parameters that are supported by the resource.

If you are using MockMvc then, to make the path parameters available for documentation, the request must be built using one of the methods on `RestDocumentationRequestBuilders` rather than `MockMvcRequestBuilders`.

When documenting path parameters, the test will fail if an undocumented path parameter is used in the request. Similarly, the test will also fail if a documented path parameter is not found in the request and the path parameter has not been marked as optional.

Path parameters can also be documented in a relaxed mode where any undocumented parameters will not cause a test failure. To do so, use the `relaxedPathParameters` method on `org.springframework.restdocs.request.RequestDocumentation`. This can be useful when documenting a particular scenario where you only want to focus on a subset of the path parameters.

If you do not want to document a path parameter, you can mark it as ignored. This will prevent it from appearing in the generated snippet while avoiding the failure described above.

Request parts

The parts of a multipart request can be documenting using `requestParts`. For example:

MockMvc | WebTestClient | REST Assured

JAVA

```
this.mockMvc.perform(multipart("/upload").file("file", "example".getBytes())) 1
    .andExpect(status().isOk())
    .andDo(document("upload", requestParts( 2
            partWithName("file").description("The file to upload")) 3
));
```

1 Perform a `POST` request with a single part named `file`.

2 Configure Spring REST Docs to produce a snippet describing the request's parts. Uses the static `requestParts` method on `org.springframework.restdocs.request.RequestDocumentation`.

3 Document the part named `file`. Uses the static `partWithName` method on `org.springframework.restdocs.request.RequestDocumentation`.

The result is a snippet named `request-parts.adoc` that contains a table describing the request parts that are supported by the resource.

When documenting request parts, the test will fail if an undocumented part is used in the request. Similarly, the test will also fail if a documented part is not found in the request and the part has not been marked as optional.

Request parts can also be documented in a relaxed mode where any undocumented parts will not cause a test failure. To do so, use the `relaxedRequestParts` method on `org.springframework.restdocs.request.RequestDocumentation`. This can be useful when documenting a particular scenario where you only want to focus on a subset of the request parts.

If you do not want to document a request part, you can mark it as ignored. This will prevent it from appearing in the generated snippet while avoiding the failure described above.

Request part payloads

The payload of a request part can be documented in much the same way as the payload of a request with support for documenting a request part's body and its fields.

Documenting a request part's body

A snippet containing the body of a request part can be generated:

MockMvc | WebTestClient | REST Assured

JAVA

```
MockMultipartFile image = new MockMultipartFile("image", "image.png", "image/png",
        "<<png data>>".getBytes());
MockMultipartFile metadata = new MockMultipartFile("metadata", "",
        "application/json", "{ \"version\": \"1.0\"}".getBytes());

this.mockMvc.perform(fileUpload("/images").file(image).file(metadata)
            .accept(MediaType.APPLICATION_JSON))
    .andExpect(status().isOk())
    .andDo(document("image-upload", requestPartBody("metadata")));  1
```

1 Configure Spring REST docs to produce a snippet containing the body of the of the request part named `metadata`. Uses the static `requestPartBody` method on `PayloadDocumentation`. payload.

The result is a snippet `request-part-${part-name}-body.adoc` that contains the part's body. For example, documenting a part named `metadata` will produce a snippet named `request-part-metadata-body.adoc`.

Documenting a request part's fields

A request part's fields can be documented in much the same way as the fields of a request or response:

MockMvc | WebTestClient | REST Assured

JAVA

```
MockMultipartFile image = new MockMultipartFile("image", "image.png", "image/png",
        "<<png data>>".getBytes());
MockMultipartFile metadata = new MockMultipartFile("metadata", "",
        "application/json", "{ \"version\": \"1.0\"}".getBytes());

this.mockMvc.perform(fileUpload("/images").file(image).file(metadata)
            .accept(MediaType.APPLICATION_JSON))
    .andExpect(status().isOk())
    .andDo(document("image-upload", requestPartFields("metadata",  1
            fieldWithPath("version").description("The version of the image"))));  2
```

1 Configure Spring REST docs to produce a snippet describing the fields in the payload of the request part named `metadata`. Uses the static `requestPartFields` method on `PayloadDocumentation`. payload.

2 Expect a field with the path `version`. Uses the static `fieldWithPath` method on `org.springframework.restdocs.payload.PayloadDocumentation`.

The result is a snippet that contains a table describing the part's fields. This snippet is named `request-part-${part-name}-fields.adoc`. For example, documenting a part named `metadata` will produce a snippet named `request-part-metadata-fields.adoc`.

When documenting fields, the test will fail if an undocumented field is found in the payload of the part. Similarly, the test will also fail if a documented field is not found in the payload of the part and the field has not been marked as optional. For payloads with a hierarchical structure, documenting a field is sufficient for all of its descendants to also be treated as having been documented.

If you do not want to document a field, you can mark it as ignored. This will prevent it from appearing in the generated snippet while avoiding the failure described above.

Fields can also be documented in a relaxed mode where any undocumented fields will not cause a test failure. To do so, use the `relaxedRequestPartFields` method on `org.springframework.restdocs.payload.PayloadDocumentation`. This can be useful when documenting a particular scenario where you only want to focus on a subset of the payload of the part.

For further information on describing fields, documenting payloads that use XML, and more please refer to the section on documenting request and response payloads.

HTTP headers

The headers in a request or response can be documented using `requestHeaders` and `responseHeaders` respectively. For example:

MockMvc | WebTestClient | REST Assured

JAVA

```
this.mockMvc
    .perform(get("/people").header("Authorization", "Basic dXNlcjpzZWNyZXQ="))  1
    .andExpect(status().isOk())
    .andDo(document("headers",
            requestHeaders(  2
                    headerWithName("Authorization").description(
                            "Basic auth credentials")),  3
            responseHeaders(  4
                    headerWithName("X-RateLimit-Limit").description(
                            "The total number of requests permitted per period"),
                    headerWithName("X-RateLimit-Remaining").description(
                            "Remaining requests permitted in current period"),
                    headerWithName("X-RateLimit-Reset").description(
                            "Time at which the rate limit period will reset"))));
```

1. Perform a `GET` request with an `Authorization` header that uses basic authentication
2. Configure Spring REST Docs to produce a snippet describing the request's headers. Uses the static `requestHeaders` method on `org.springframework.restdocs.headers.HeaderDocumentation`.
3. Document the `Authorization` header. Uses the static `headerWithName` method on `org.springframework.restdocs.headers.HeaderDocumentation`.
4. Produce a snippet describing the response's headers. Uses the static `responseHeaders` method on `org.springframework.restdocs.headers.HeaderDocumentation`.

The result is a snippet named `request-headers.adoc` and a snippet named `response-headers.adoc`. Each contains a table describing the headers.

When documenting HTTP Headers, the test will fail if a documented header is not found in the request or response.

Reusing snippets

It's common for an API that's being documented to have some features that are common across several of its resources. To avoid repetition when documenting such resources a `Snippet` configured with the common elements can be reused.

First, create the `Snippet` that describes the common elements. For example:

JAVA
```
protected final LinksSnippet pagingLinks = links(
        linkWithRel("first").optional().description("The first page of results"),
        linkWithRel("last").optional().description("The last page of results"),
        linkWithRel("next").optional().description("The next page of results"),
        linkWithRel("prev").optional().description("The previous page of results"));
```

Second, use this snippet and add further descriptors that are resource-specific. For example:

MockMvc | WebTestClient | REST Assured

JAVA
```
this.mockMvc.perform(get("/").accept(MediaType.APPLICATION_JSON))
    .andExpect(status().isOk())
    .andDo(document("example", this.pagingLinks.and(  1
            linkWithRel("alpha").description("Link to the alpha resource"),
            linkWithRel("bravo").description("Link to the bravo resource"))));
```

1 Reuse the `pagingLinks` `Snippet` calling `and` to add descriptors that are specific to the resource that is being documented.

The result of the example is that links with the rels `first`, `last`, `next`, `previous`, `alpha`, and `bravo` are all documented.

Documenting constraints

Spring REST Docs provides a number of classes that can help you to document constraints. An instance of `ConstraintDescriptions` can be used to access descriptions of a class's constraints. For example:

JAVA

```
public void example() {
    ConstraintDescriptions userConstraints = new ConstraintDescriptions(UserInput.class);
 1
    List<String> descriptions = userConstraints.descriptionsForProperty("name");  2
}

static class UserInput {

    @NotNull
    @Size(min = 1)
    String name;

    @NotNull
    @Size(min = 8)
    String password;

}
```

1 Create an instance of `ConstraintDescriptions` for the `UserInput` class

2 Get the descriptions of the name property's constraints. This list will contain two descriptions; one for the `NotNull` constraint and one for the `Size` constraint.

The `ApiDocumentation` (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/rest-notes-spring-hateoas/src/test/java/com/example/notes/ApiDocumentation.java) class in the Spring HATEOAS sample shows this functionality in action.

Finding constraints

By default, constraints are found using a Bean Validation `Validator`. Currently, only property constraints are supported. You can customize the `Validator` that's used by creating `ConstraintDescriptions` with a custom `ValidatorConstraintResolver` instance. To take complete control of constraint resolution, your own implementation of `ConstraintResolver` can be used.

Describing constraints

Default descriptions are provided for all of Bean Validation 2.0's constraints:

- AssertFalse
- AssertTrue
- DecimalMax
- DecimalMin

- Digits
- Email
- Future
- FutureOrPresent
- Max
- Min
- Negative
- NegativeOrZero
- NotBlank
- NotEmpty
- NotNull
- Null
- Past
- PastOrPresent
- Pattern
- Positive
- PositiveOrZero
- Size

Default descriptions are also provided for the following constraints from Hibernate Validator:

- CodePointLength
- CreditCardNumber
- Currency
- EAN
- Email
- Length
- LuhnCheck
- Mod10Check

- Mod11Check
- NotBlank
- NotEmpty
- Currency
- Range
- SafeHtml
- URL

To override the default descriptions, or to provide a new description, create a resource bundle with the base name `org.springframework.restdocs.constraints.ConstraintDescriptions`. The Spring HATEOAS-based sample contains an example of such a resource bundle (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/rest-notes-spring-hateoas/src/test/resources/org/springframework/restdocs/constraints/ConstraintDescriptions.properties).

Each key in the resource bundle is the fully-qualified name of a constraint plus `.description`. For example, the key for the standard `@NotNull` constraint is `javax.validation.constraints.NotNull.description`.

Property placeholder's referring to a constraint's attributes can be used in its description. For example, the default description of the `@Min` constraint, `Must be at least ${value}`, refers to the constraint's `value` attribute.

To take more control of constraint description resolution, create `ConstraintDescriptions` with a custom `ResourceBundleConstraintDescriptionResolver`. To take complete control, create `ConstraintDescriptions` with a custom `ConstraintDescriptionResolver` implementation.

Using constraint descriptions in generated snippets

Once you have a constraint's descriptions, you're free to use them however you like in the generated snippets. For example, you may want to include the constraint descriptions as part of a field's description. Alternatively, you could include the constraints as extra information in the request fields snippet. The `ApiDocumentation` (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/samples/rest-notes-spring-hateoas/src/test/java/com/example/notes/ApiDocumentation.java) class in the Spring HATEOAS-based sample illustrates the latter approach.

Default snippets

A number of snippets are produced automatically when you document a request and response.

Snippet	Description
`curl-request.adoc`	Contains the `curl` (https://curl.haxx.se) command that is equivalent to the `MockMvc` call that is being documented
`httpie-request.adoc`	Contains the `HTTPie` (http://httpie.org) command that is equivalent to the `MockMvc` call that is being documented
`http-request.adoc`	Contains the HTTP request that is equivalent to the `MockMvc` call that is being documented
`http-response.adoc`	Contains the HTTP response that was returned
`request-body.adoc`	Contains the body of the request that was sent
`response-body.adoc`	Contains the body of the response that was returned

You can configure which snippets are produced by default. Please refer to the configuration section for more information.

Using parameterized output directories

When using MockMvc or REST Assured, the output directory used by `document` can be parameterized. The output directory cannot be parameterized when using WebTestClient.

The following parameters are supported:

Parameter	Description
{methodName}	The unmodified name of the test method
{method-name}	The name of the test method, formatted using kebab-case
{method_name}	The name of the test method, formatted using snake_case
{ClassName}	The unmodified simple name of the test class

Parameter	Description
{class-name}	The simple name of the test class, formatted using kebab-case
{class_name}	The simple name of the test class, formatted using snake_case
{step}	The count of calls made to the service in the current test

For example, `document("{class-name}/{method-name}")` in a test method named `creatingANote` on the test class `GettingStartedDocumentation`, will write snippets into a directory named `getting-started-documentation/creating-a-note`.

A parameterized output directory is particularly useful in combination with an `@Before` method. It allows documentation to be configured once in a setup method and then reused in every test in the class:

MockMvc | REST Assured

JAVA

```
@Before
public void setUp() {
    this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context)
            .apply(documentationConfiguration(this.restDocumentation))
            .alwaysDo(document("{method-name}/{step}/")).build();
}
```

With this configuration in place, every call to the service you are testing will produce the default snippets without any further configuration. Take a look at the `GettingStartedDocumentation` classes in each of the sample applications to see this functionality in action.

Customizing the output

Customizing the generated snippets

Spring REST Docs uses Mustache (https://mustache.github.io) templates to produce the generated snippets. Default templates (https://github.com/spring-projects/spring-restdocs/tree/v2.0.2.RELEASE/spring-restdocs-core/src/main/resources/org/springframework/restdocs/templates) are provided for each of the snippets that Spring REST Docs can produce. To customize a snippet's content, you can provide your own template.

Templates are loaded from the classpath from an `org.springframework.restdocs.templates` subpackage. The name of the subpackage is determined by the ID of the template format that is in use. The default template format, Asciidoctor, has the ID `asciidoctor` so snippets are loaded from `org.springframework.restdocs.templates.asciidoctor`. Each template is named after the snippet that it will produce. For example, to override the template for the `curl-request.adoc` snippet, create a template named `curl-request.snippet` in `src/test/resources/org/springframework/restdocs/templates/asciidoctor`.

Including extra information

There are two ways to provide extra information for inclusion in a generated snippet:

1. Use the `attributes` method on a descriptor to add one or more attributes to it.
2. Pass in some attributes when calling `curlRequest`, `httpRequest`, `httpResponse`, etc. Such attributes will be associated with the snippet as a whole.

Any additional attributes are made available during the template rendering process. Coupled with a custom snippet template, this makes it possible to include extra information in a generated snippet.

A concrete example of the above is the addition of a constraints column and a title when documenting request fields. The first step is to provide a `constraints` attribute for each field that you are documenting and to provide a `title` attribute:

MockMvc | WebTestClient | REST Assured

JAVA

```
.andDo(document("create-user", requestFields(
        attributes(key("title").value("Fields for user creation")),  1
        fieldWithPath("name").description("The user's name")
                .attributes(key("constraints")
                        .value("Must not be null. Must not be empty")),  2
        fieldWithPath("email").description("The user's email address")
                .attributes(key("constraints")
                        .value("Must be a valid email address"))));  3
```

1 Configure the `title` attribute for the request fields snippet

2 Set the `constraints` attribute for the `name` field

3 Set the `constraints` attribute for the `email` field

The second step is to provide a custom template named `request-fields.snippet` that includes the information about the fields' constraints in the generated snippet's table and adds a title:

```
.{{title}} 1
|===
|Path|Type|Description|Constraints 2

{{#fields}}
|{{path}}
|{{type}}
|{{description}}
|{{constraints}} 3

{{/fields}}
|===
```

1 Add a title to the table

2 Add a new column named "Constraints"

3 Include the descriptors' `constraints` attribute in each row of the table

Customizing requests and responses

There may be situations where you do not want to document a request exactly as it was sent or a response exactly as it was received. Spring REST Docs provides a number of preprocessors that can be used to modify a request or response before it's documented.

Preprocessing is configured by calling `document` with an `OperationRequestPreprocessor`, and/or an `OperationResponsePreprocessor`. Instances can be obtained using the static `preprocessRequest` and `preprocessResponse` methods on `Preprocessors`. For example:

MockMvc | WebTestClient | REST Assured

```
this.mockMvc.perform(get("/")).andExpect(status().isOk())
    .andDo(document("index", preprocessRequest(removeHeaders("Foo")),  1
            preprocessResponse(prettyPrint())));  2
```

1. Apply a request preprocessor that will remove the header named `Foo`.
2. Apply a response preprocessor that will pretty print its content.

Alternatively, you may want to apply the same preprocessors to every test. You can do so by configuring the preprocessors using the `RestDocumentationConfigurer` API in your `@Before` method. For example to remove the `Foo` header from all requests and pretty print all responses:

MockMvc | WebTestClient | REST Assured

```
private MockMvc mockMvc;

@Before
public void setup() {
    this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context)
        .apply(documentationConfiguration(this.restDocumentation).operationPreprocessors()
            .withRequestDefaults(removeHeaders("Foo"))  1
            .withResponseDefaults(prettyPrint()))  2
        .build();
}
```

1. Apply a request preprocessor that will remove the header named `Foo`.
2. Apply a response preprocessor that will pretty print its content.

Then, in each test, any configuration specific to that test can be performed. For example:

MockMvc | WebTestClient | REST Assured

JAVA

```
this.mockMvc.perform(get("/"))
        .andExpect(status().isOk())
        .andDo(document("index",
                links(linkWithRel("self").description("Canonical self link"))
        ));
```

Various built in preprocessors, including those illustrated above, are available via the static methods on `Preprocessors`. See below for further details.

Preprocessors

Pretty printing

`prettyPrint` on `Preprocessors` formats the content of the request or response to make it easier to read.

Masking links

If you're documenting a Hypermedia-based API, you may want to encourage clients to navigate the API using links rather than through the use of hard coded URIs. One way to do this is to limit the use of URIs in the documentation. `maskLinks` on `Preprocessors` replaces the `href` of any links in the response with `...`. A different replacement can also be specified if you wish.

Removing headers

`removeHeaders` on `Preprocessors` removes any headers from the request or response where the name is equal to any of the given header names.

`removeMatchingHeaders` on `Preprocessors` removes any headers from the request or response where the name matches any of the given regular expression patterns.

Replacing patterns

`replacePattern` on `Preprocessors` provides a general purpose mechanism for replacing content in a request or response. Any occurrences of a regular expression are replaced.

Modifying request parameters

`modifyParameters` on `Preprocessors` can be used to add, set, and remove request parameters.

Modifying URIs

If you are using MockMvc or a WebTestClient that is not bound to a server, URIs should be customized by changing the configuration.

`modifyUris` on `Preprocessors` can be used to modify any URIs in a request or a response. When using REST Assured or WebTestClient bound to a server, this allows you to customize the URIs that appear in the documentation while testing a local instance of the service.

Writing your own preprocessor

If one of the built-in preprocessors does not meet your needs, you can write your own by implementing the `OperationPreprocessor` interface. You can then use your custom preprocessor in exactly the same way as any of the built-in preprocessors.

If you only want to modify the content (body) of a request or response, consider implementing the `ContentModifier` interface and using it with the built-in `ContentModifyingOperationPreprocessor`.

Configuration

Documented URIs

MockMvc URI customization

When using MockMvc, the default configuration for URIs documented by Spring REST Docs is:

Setting	Default
Scheme	`http`
Host	`localhost`
Port	`8080`

This configuration is applied by `MockMvcRestDocumentationConfigurer`. You can use its API to change one or more of the defaults to suit your needs:

JAVA

```
this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context)
        .apply(documentationConfiguration(this.restDocumentation).uris()
                .withScheme("https")
                .withHost("example.com")
                .withPort(443))
        .build();
```

If the port is set to the default for the configured scheme (port 80 for HTTP or port 443 for HTTPS), it will be omitted from any URIs in the generated snippets.

To configure a request's context path, use the `contextPath` method on `MockHttpServletRequestBuilder`.

REST Assured URI customization

REST Assured tests a service by making actual HTTP requests. As a result, URIs must be customized once the operation on the service has been performed but before it is documented. A REST-Assured specific preprocessor is provided for this purpose.

WebTestClient URI customization

When using WebTestClient, the default base for URIs documented by Spring REST Docs is `http://localhost:8080`. This base can be customized using the `baseUrl(String)` method on `WebTestClient.Builder` (https://docs.spring.io/spring-framework/docs/5.0.x/javadoc-api/org/springframework/test/web/reactive/server/WebTestClient.Builder.html#baseUrl-java.lang.String-):

```
@Before
public void setUp() {
    this.webTestClient = WebTestClient.bindToApplicationContext(this.context)
        .configureClient()
        .baseUrl("https://api.example.com")  1
        .filter(documentationConfiguration(this.restDocumentation)).build();
}
```

1 Configure the base of documented URIs to be `https://api.example.com`.

Snippet encoding

The default snippet encoding is `UTF-8`. You can change the default snippet encoding using the `RestDocumentationConfigurer` API. For example, to use `ISO-8859-1`:

MockMvc | WebTestClient | REST Assured

```
this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context)
        .apply(documentationConfiguration(this.restDocumentation)
                .snippets().withEncoding("ISO-8859-1"))
        .build();
```

When Spring REST Docs converts a request or response's content to a String, the `charset` specified in the `Content-Type` header will be used if it is available. In its absence, the JVM's default `Charset` will be used. The JVM's default `Charset` can be configured using the `file.encoding` system property.

Snippet template format

The default snippet template format is Asciidoctor. Markdown is also supported out of the box. You can change the default format using the `RestDocumentationConfigurer` API:

MockMvc | WebTestClient | REST Assured

JAVA

```
this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context)
        .apply(documentationConfiguration(this.restDocumentation)
                .snippets().withTemplateFormat(TemplateFormats.markdown()))
        .build();
```

Default snippets

Six snippets are produced by default:

- `curl-request`
- `http-request`
- `http-response`
- `httpie-request`
- `request-body`
- `response-body`

You can change the default snippet configuration during setup using the `RestDocumentationConfigurer` API. For example, to only produce the `curl-request` snippet by default:

MockMvc | WebTestClient | REST Assured

JAVA

```
this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context)
        .apply(documentationConfiguration(this.restDocumentation).snippets()
                .withDefaults(curlRequest()))
        .build();
```

Default operation preprocessors

You can configure default request and response preprocessors during setup using the `RestDocumentationConfigurer` API. For example, to remove the `Foo` headers from all requests and pretty print all responses:

MockMvc | WebTestClient | REST Assured

JAVA

```
this.mockMvc = MockMvcBuilders.webAppContextSetup(this.context)
        .apply(documentationConfiguration(this.restDocumentation)
                .operationPreprocessors()
                .withRequestDefaults(removeHeaders("Foo"))  1
                .withResponseDefaults(prettyPrint()))  2
        .build();
```

1 Apply a request preprocessor that will remove the header named `Foo` .

2 Apply a response preprocessor that will pretty print its content.

Working with Asciidoctor

This section describes any aspects of working with Asciidoctor that are particularly relevant to Spring REST Docs.

Resources

- Syntax quick reference (http://asciidoctor.org/docs/asciidoc-syntax-quick-reference)
- User manual (http://asciidoctor.org/docs/user-manual)

Including snippets

Including multiple snippets for an operation

A macro named `operation` can be used to import all or some of the snippets that have been generated for a specific operation. It is made available by including `spring-restdocs-asciidoctor` in your project's build configuration.

If you are using Gradle and its daemon or support for continuous builds, do not use version 1.5.6 of the `org.asciidoctor.convert` plugin. It contains a regression (https://github.com/asciidoctor/asciidoctor-gradle-plugin/issues/222) that prevents extensions from working reliably.

The target of the macro is the name of the operation. In its simplest form, the macro can be used to include all of the snippets for an operation, as shown in the following example:

```
operation::index[]
```

The operation macro also supports a `snippets` attribute. The `snippets` attribute can be used to select the snippets that should be included. The attribute's value is a comma-separated list. Each entry in the list should be the name of a snippet file, minus the `.adoc` suffix, to include. For example, only the curl, HTTP request and HTTP response snippets can be included as shown in the following example:

```
operation::index[snippets='curl-request,http-request,http-response']
```

This is the equivalent of the following:

ADOC

```
[[example_curl_request]]
== Curl request

include::{snippets}/index/curl-request.adoc[]

[[example_http_request]]
== HTTP request

include::{snippets}/index/http-request.adoc[]

[[example_http_response]]
== HTTP response

include::{snippets}/index/http-response.adoc[]
```

Section titles

For each snippet that's including using `operation`, a section with a title will be created. Default titles are provided for the built-in snippets:

Snippet	Title
curl-request	Curl Request
http-request	HTTP request
http-response	HTTP response
httpie-request	HTTPie request
links	Links
request-body	Request body
request-fields	Request fields
response-body	Response body
response-fields	Response fields

For snippets not listed in the table above, a default title will be generated by replacing `-` characters with spaces and capitalising the first letter. For example, the title for a snippet named `custom-snippet` will be "Custom snippet".

The default titles can be customized using document attributes. The name of the attribute should be `operation-{snippet}-title`. For example, to customize the title of the `curl-request` snippet to be "Example request", use the following attribute:

```
:operation-curl-request-title: Example request
```

Including individual snippets

The include macro (http://asciidoctor.org/docs/asciidoc-syntax-quick-reference/#include-files) is used to include individual snippets in your documentation. The `snippets` attribute that is automatically set by `spring-restdocs-asciidoctor` configured in the build configuration can be used to reference the snippets output directory. For example:

```
include::{snippets}/index/curl-request.adoc[]
```

Customizing tables

Many of the snippets contain a table in its default configuration. The appearance of the table can be customized, either by providing some additional configuration when the snippet is included or by using a custom snippet template.

Formatting columns

Asciidoctor has rich support for formatting a table's columns (http://asciidoctor.org/docs/user-manual/#cols-format). For example, the widths of a table's columns can be specified using the `cols` attribute:

```
[cols="1,3"] 1
include::{snippets}/index/links.adoc[]
```

1 The table's width will be split across its two columns with the second column being three times as wide as the first.

Configuring the title

The title of a table can be specified using a line prefixed by a `.`:

```
.Links 1
include::{snippets}/index/links.adoc[]
```

1 The table's title will be `Links`.

Avoiding table formatting problems

Asciidoctor uses the `|` character to delimit cells in a table. This can cause problems if you want a `|` to appear in a cell's contents. The problem can be avoided by escaping the `|` with a backslash, i.e. by using `\|` rather than `|`.

All of the default Asciidoctor snippet templates perform this escaping automatically use a Mustache lamba named `tableCellContent`. If you write your own custom templates you may want to use this lamba. For example, to escape `|` characters in a cell that contains the value of a `description` attribute:

```
| {{#tableCellContent}}{{description}}{{/tableCellContent}}
```

Further reading

Refer to the Tables section of the Asciidoctor user manual (http://asciidoctor.org/docs/user-manual/#tables) for more information about customizing tables.

Working with Markdown

This section describes any aspects of working with Markdown that are particularly relevant to Spring REST Docs.

Limitations

Markdown was originally designed for people writing for the web and, as such, isn't as well-suited to writing documentation as Asciidoctor. Typically, these limitations are overcome by using another tool that builds on top of Markdown.

Markdown has no official support for tables. Spring REST Docs' default Markdown snippet templates use Markdown Extra's table format (https://michelf.ca/projects/php-markdown/extra/#table).

Including snippets

Markdown has no built-in support for including one Markdown file in another. To include the generated snippets of Markdown in your documentation, you should use an additional tool that supports this functionality. One example that's particularly well-suited to documenting APIs is Slate (https://github.com/tripit/slate).

Contributing

Spring REST Docs is intended to make it easy for you to produce high-quality documentation for your RESTful services. However, we can't achieve that goal without your contributions.

Questions

You can ask questions about Spring REST Docs on StackOverflow (https://stackoverflow.com) using the `spring-restdocs` tag. Similarly, we encourage you to help your fellow Spring REST Docs users by answering questions.

Bugs

If you believe you have found a bug, please take a moment to search the existing issues (https://github.com/spring-projects/spring-restdocs/issues?q=is%3Aissue). If no one else has reported the problem, please open a new issue (https://github.com/spring-projects/spring-restdocs/issues/new) that describes the problem in detail and, ideally, includes a test that reproduces it.

Enhancements

If you'd like an enhancement to be made to Spring REST Docs, pull requests are most welcome. The source code is on GitHub (https://github.com/spring-projects/spring-restdocs). You may want to search the existing issues (https://github.com/spring-projects/spring-restdocs/issues?q=is%3Aissue) and pull requests (https://github.com/spring-projects/spring-restdocs/pulls?q=is%3Apr) to see if the enhancement is already being worked on. You may also want to open a new issue (https://github.com/spring-projects/spring-restdocs/issues/new) to discuss a possible enhancement before work on it begins.

Version 2.0.2.RELEASE
Last updated 2018-07-19 13:44:32 UTC