Documenting Hypermedia REST APIs with Spring REST Docs

04:30, 23 Jul, 2018 ARCHITECTURE

Jeroen Reijn
Software Architect

WORKPLACE	Amsterdam
ARTICLES	6 Articles

More

Last year, at the end of summer, the project I was working on required a public REST API. During the requirements gathering phase we discussed the 'level' of our future REST API. In case you're unfamiliar with Leonard Richardson's REST maturity model I would highly recommend reading this article written by Martin Fowler about the model.

In my opinion a public API requires really good documentation. The documentation helps to explain how to use the API, what the resource represents (explain your domain model) and can help to increase adoption of the API. If I have to consume an API myself I'm always relieved if there is some well written API documentation available.

After the design phase we chose to build a Level 3 REST API. Documenting a level 3 REST api is not that easy. We looked at Swagger / OpenAPI, but in the 2.0 version of the spec, which was available at the time, it was not possible to design and or document link relations, which are part of the third level. After some research we learned there was a Spring project called Spring REST Docs, which allowed you to document any type of API. It works by writing tests for your API endpoints and acts as a proxy which captures the requests and responses and turns them into documentation. It does not only look at the request and response cycle, but actually inspects and validates if you've documented certain request or response fields. If you haven't specified and documented them, your actual test will fail. This is really neat feature! It makes sure that your documentation is always in sync with your API.

Using Spring REST Docs is pretty straight-forward. You can start by just adding a dependency to your Maven or Gradle based project.

HTML

```
<dependency>
  <groupId>org.springframework.restdocs</groupId>
  <artifactId>spring-restdocs-mockmvc</artifactId>
  <version>${spring.restdoc.version}</version>
  <scope>test</scope>
</dependency>
```

Now when you use for instance Spring MockMVC you can test an API resource by having the following code:

HTML

```
@Test
public void testGetAllPlanets() throws Exception {
    mockMvc.perform(get("/planets").accept(MediaType.APPLICATION_JSON))
    .andExpect(status().isOk())
    .andExpect(jsonPath("$.length()",is(2)));
}
```

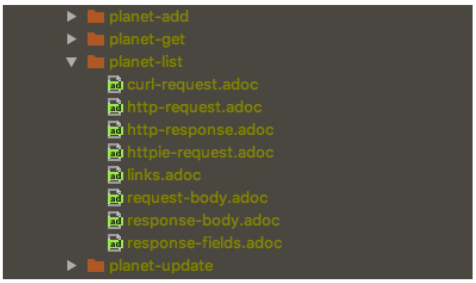
All the test does is performing a GET request on the /planets resource. Now to document this API resource all you

need to do is add the document() call with an identifier, which will result in documentation for the /planets resource.

```
HTML

@Test
public void testGetAllPlanets() throws Exception {
    mockMvc.perform(get("/planets").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.length()",is(2)))
        .andDo(document("planet-list"));
}
```

Now when you run this test, Spring REST Docs will generate several AsciiDoc snippets for this API resource.



Let's inspect one of these asciidoc snippets.

```
HTML

[source,bash]
----
$ curl 'https://api.mydomain.com/v1/planets' -i -X GET \
    -H 'Accept: application/hal+json'
----
```

Looks pretty neat right? It generates a nice example of how to perform a request against the API by using **curl**. It will show what headers are required or in case you want to send a payload how to pass the payload along with the request.

Documenting how to perform an API call is nice, but it gets even better when we start documenting fields. By documenting fields in the request or response we will immediately start validating the documentation for missing fields or parameters. For documenting fields in the JSON response body we can use the *responseFields* snippet instruction.

```
HTML

@Test
public void testGetPerson() throws Exception {
  mockMvc.perform(get("/people/{id}", personFixture.getId())
         .accept(MediaTypes.HAL_JSON_VALUE))
         .andExpect(status().isOk())
         .andDo(document("people-get-example",
                pathParameters(
                    parameterWithName("id").description("Person's id")
                ),
                links(halLinks(),
                      linkWithRel("self").ignored()
                ),
                responseFields(
                        fieldWithPath("id").description("Person's id"),
                        fieldWithPath("name").description("Person's name"),
                        subsectionWithPath("_links").ignored()
                 ))
          );
    }
```

In the above example we have documented 2 fields: **id** and **name**. We can add a description, but also a type, specify if they are optional or we can even ignore specific sections like I did in the above example. Ignoring a section is possible in case you want to document them once since they will be available across multiple resources. Now if you are very strict with writing JavaDoc you might also want to consider using Spring Auto REST Docs. Spring Auto REST Docs uses introspection of you Java classes and POJOs to generate the field descriptions for you. It's pretty neat, but I found some corner cases for when you use a hypermedia API. You can't really create specific documentation for Link objects

The documentation comes from the Spring Javadocs itself, so we chose to leave auto rest docs out.

Having a bunch of asciidoc snippets is nice, but it's better to have some human readable format like HTML. This is where the maven asciidoctor plugin comes in. It has the ability to process the asciidoc files and turn it into a publishable format like HTML or PDF. To get the HTML output (also known as backend) all you need to do is add the maven plugin with the correct configuration.

HTML

```
<build>
  <plugins>
    ....
    <plugin>
      <groupId>org.asciidoctor</groupId>
      <artifactId>asciidoctor-maven-plugin</artifactId>
      <version>1.5.3</version>
      <executions>
        <execution>
          <id>generate-docs</id>
          <phase>prepare-package</phase>
          <goals>
            <goal>process-asciidoc</goal>
          </goals>
          <configuration>
            <backend>html</backend>
            <doctype>book</doctype>
          </configuration>
        </execution>
      </executions>
      <dependencies>
        <dependency>
          <groupId>org.springframework.restdocs</groupId>
          <artifactId>spring-restdocs-asciidoctor</artifactId>
          <version>2.0.1.RELEASE</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
```

Now to turn all the different asciidoc snippets into once single documentation page you can create an index.adoc file that aggregates the generated AsciiDoc snippets into a single file. Let's take a look at an example:

HTML

```
= DevCon REST TDD Demo
Jeroen Reijn;
:doctype: book
:icons: font
:source-highlighter: highlightjs
:toc: left
:toclevels: 4
:sectlinks:
:operation-curl-request-title: Example request
:operation-http-response-title: Example response

[[resources-planets]]
== Planets

The Planets resources is used to create and list planets

[[resources-planets-list]]
=== Listing planets

A `GET` request will list all of the service's planets.

operation::planets-list-example[snippets='response-fields,curl-request,http-response']

[[resources-planets-create]]
=== Creating a planet

A `POST` request is used to create a planet.

operation::planets-create-example[snippets='request-fields,curl-request,http-response']
```

The above asciidoc snippet shows you how to write documentation in asciidoc and how to include certain operations and even how you can selectively pick certain snippets which you want to include. You can see the result in the Github pages version.

The advantage of splitting the generation from the actual HTML production has several benefits. One that I found appealing myself is that by documenting the API in two steps (code and documentation) you can have multiple people working on writing the documentation. At my previous company we had a dedicated technical writer that wrote the documentation for our product. An API is also a product so you can have engineers create the API, tests the API and

document the resources by generate the documentation snippets and the technical writer can then do their own tick when it comes to writing good readable/consumable content. Writing documentation is a trade by itself and I have always liked the mailchimp content style guide for some clear guidelines on writing technical documentation.

Now if we take a look at the overall process we will see it integrates nicely into our CI / CD pipeline. All documentation is version control managed and part of the same release cycle of the API itself.

If you want to take look at a working example you can check out my DevCon REST TDD demo repositoryon github or see me use Spring Rest Docs to live code and document an API during my talk at DevCon.

ARCHITECTURE

Related More from author

Recommend Tweet Share Sort by Newest

Start the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS

Name

Be the first to comment.

Subscribe Add Disqus to your site Disqus' Privacy Policy DISQUS

join our growing developer community

your email address

Subscribe

luminis
Conversing worlds

Stadsring 107
3811 HP Amersfoort
The Netherlands
+31 88 5864 600
info@luminis.eu

About

Our Academy

Luminis

Content

Blogs

Video

Interested

Want to learn more about what Luminis can do for you? Please don't hesitate to contact us.