Blog Zenika
IT Systems architecture, frameworks, and random thoughts

zenika
ARCHITECTURE INFORMATIQUE

16
juil.
2013

Documenting a REST API with Swagger and Spring MVC

Posté par Arnaud Cogoluègnes dans Développement
Tags : hateoas, rest, spring, springmvc, swagger

REST is now the most common way to expose web services. But how to say to clients how to use a REST API? There's no real standard or at least de facto standard to expose a REST contract. Many API resorts to a human-readable documentation, which is manually edited and thus hard to keep perfectly synchronized with the API. Another way is to create the documentation from the code itself, and this is what this article covers, by using Swagger on top of Spring MVC.

Swagger

Swagger (https://developers.helloreverb.com/swagger/) is a specification for documenting REST API. It specifies the format (URL, method, and representation) to describe REST web services. It provides also tools to generate/compute the documentation from application code.

What does this mean? As an application developer, you write web services using your favorite framework, Swagger scans your code and exposes the documentation on some URL. Any client can consume this URL (which comes as XML or JSON documents) and learn how to use your REST web services: which HTTP methods to call on which URL, which input documents to send, which status code to expect, etc.

We're going to see how to use Swagger on top of Spring MVC, but remember Swagger is a specification and supports a wide range of frameworks.

The controller

The controller contains the basic CRUD operations, it uses the Spring MVC API:

```
@Controller
@RequestMapping("/contacts")
public class ContactController {

 @Autowired ContactService contactService;

 @ResponseBody
 @RequestMapping(method=RequestMethod.GET)
 public Collection<Contact> select() {
  return contactService.select();
 }

 @ResponseBody
 @RequestMapping(value="/{id}",method=RequestMethod.GET)
 public Contact get(@PathVariable Long id) {
  return contactService.get(id);
 }

 @RequestMapping(method=RequestMethod.POST)
 @ResponseStatus(HttpStatus.CREATED)
```

```
 public void add(@RequestBody Contact contact,HttpServletResponse response) {
  contactService.add(contact);
  String location = ServletUriComponentsBuilder.fromCurrentRequest()
   .pathSegment("{id}").buildAndExpand(contact.getId())
   .toUriString();

  response.setHeader("Location",location);
 }

 @RequestMapping(value="/{id}",method=RequestMethod.PUT)
 @ResponseStatus(HttpStatus.NO_CONTENT)
 public void update(@PathVariable Long id,@RequestBody Contact contact) {
  contact.setId(id);
  contactService.update(contact);
 }

}
```

I made the controller as simple as possible, the point isn't to have a perfect, bullet-proof controller, but rather to illustrate the use of Swagger.

Swagger configuration

There's a nice and active project (https://github.com/martypitt/swagger-springmvc) on github that provides Swagger support for Spring MVC. The XML configuration is straightforward:

```
<bean class="com.mangofactory.swagger.configuration.DocumentationConfig" />

<context:property-placeholder location="classpath:/swagger.properties" />
```

Swagger Spring MVC needs a couple of properties from a property file:

```
documentation.services.version=1.0
documentation.services.basePath=http://localhost:8080/zencontact
```

We'll see shortly how Swagger Spring MVC uses this 2 properties.

What's happening under the hood? Swagger Spring MVC scans the Spring MVC controllers on start-up and registers a documentation controller that exposes the operations the controllers allows. This documentation follows the Swagger specification: any client that understands this specification can use the API. The good news is the documentation is based on the code itself: any change to the code is reflected on the documentation, no need to maintain an external document.

Swagger Spring MVC uses Spring MVC annotations to compute the documentation, but it also understands Swagger annotations. Let's add the `@Api` annotation on the controller:

```
@Api(value = "contacts", description = "contacts") // Swagger annotation
@Controller
@RequestMapping("/contacts")
public class ContactController { ... }
```

It's time now to discover the documentation.

The documentation

The documentation endpoint is on the `/api-docs` URL, if we hit this URL and ask for JSON content, we'll get the following:


```
{
  "apiVersion": "1.0",
  "swaggerVersion": "1.0",
  "basePath": "http://localhost:8080/zencontact",
  "apis": [
    {
      "path": "/api-docs/contacts",
      "description": "contacts"
    }
  ]
```

```
}
```

Note we stumble on the 2 properties we set up previously, the version of our API and the base path of the API. They'll appear on each page of our documentation.

If we know we want to work on contacts, we just have to follow the link to find out more about the exposed operations on this resource. So let's hit `/api-docs/contacts`, here is an excerpt of the result:

```
{
  "apiVersion": "1.0",
  "swaggerVersion": "1.0",
  "basePath": "http://localhost:8080/zencontact",
  "resourcePath": "/contacts",
  "apis": [
    {
      "path": "/contacts",
      "description": "contacts",
      "operations": [
        {
          "httpMethod": "GET",
          "summary": "select",
          "notes": "",
          "deprecated": false,
          "responseClass": "Collection[Contact]",
          "nickname": "select"
        }
      ]
    },
    (...) other APIs and operations
  ],
  "models": {
    "Collection[Contact]": {
      "properties": {
        "empty": {
          "type": "boolean"
        }
      },
      "type": "Collection[Contact]"
    },
    "Contact": {
      "properties": {
        "id": {
          "type": "long"
        },
        "lastname": {
          "type": "string"
        },
        "firstname": {
          "type": "string"
        }
      },
      "type": "Contact"
    }
  }
}
```

There are 2 parts in this documentation: the operations and the models. A client can send a GET on the `/contacts` URL to select the contacts. This is an example of an operation. We see this operation returns a collection of contacts. A client can learn more about this model in the `models` section. Note the `Contact` model, which is used by the PUT and POST operations (not shown above). All of this is scanned from the controller.

So far, so good: I write a controller and get its documentation for free. But this is only the beginning: let's see now how to consume this documentation, first from a programmatic client, and second from a neat user interface.

A client of the REST API

Let's see how a programmatic client manages to list the contacts by only knowing declaratively that it wants, that is the name of the resource, the description of the operation, and be sure it's a GET (read) operation. The unique entry point of the API is the documentation URL:

```
String documentationUrl = "http://localhost:8080/zencontact/api-docs"
String resourceType = "contacts";

String documentation = tpl.getForObject(documentationUrl, String.class);

// extracts http://localhost:8080/zencontact
String basePath = JsonPath.read(documentation, "basePath");
// extracts /api-docs/contacts to discover the available operations on the contacts resource
List<String> apiDocumentationPath = JsonPath.read(
  documentation,
  "apis[?].path",
  filter(where("description").is(resourceType))
);
```

Go back to the first JSON document if you're not sure about what the previous code snippet does. It actually extracts the path to the documentation of the contacts resource.

It's time now to discover the operation we want to use, so let's hit the resource documentation:

```
// let's go to http://localhost:8080/zencontact/api-docs/contacts
documentation = tpl.getForObject(basePath+apiDocumentationPath.get(0), String.class);
// selects the info about the "select" operation on "contacts"
// we know it's GET, but we need to know the URL (it's actually "/contacts")
List<String> apis = JsonPath.read(
  documentation,
  "$.apis[?].path",
  new OperationNicknameFilter("select","GET")
);
// contains "/contacts"
String resourcePath = apis.get(0);
String contacts = tpl.getForObject(basePath+resourcePath, String.class);
```

This time, refer back to the second JSON document to understand what the client is doing. It basically searches the path of an operation whose nickname is `select` and ensures this operation is exposed on a GET. Once it has the path, it sends the request and gets the following response, the contacts:


```
[
  {
    "id": 1,
    "firstname": "Erich",
    "lastname": "Gamma"
  },
  {
    "id": 2,
    "firstname": "Richard",
    "lastname": "Helm"
  },
  {
    "id": 3,
    "firstname": "Ralph",
    "lastname": "Johnson"
  },
  {
    "id": 4,
    "firstname": "John",
    "lastname": "Vlissides"
  }
]
```

Isn't that great? With only the documentation URL as an entry point and a small idea about that it wants to do, the client can find the appropriate request to make. If we modify the controller, the client shouldn't break, as long as it follows the documentation. Nice decoupling.

What about HATEOAS?

The Swagger documentation implements HATEOAS (http://en.wikipedia.org/wiki/HATEOAS) : once a client reaches the documentation "homepage", it can follow the link to learn more about the API of the various resources the system exposes.

Do we need HATEOAS in our resources when we have the Swagger documentation? Of course we do. The Swagger documentation helps us to know about the resources and even about the expected formats. But the resources should expose links to navigate across each other.

The sample controller doesn't use HATEOAS for simplicity sake, but we can add links to the resources quite easily.

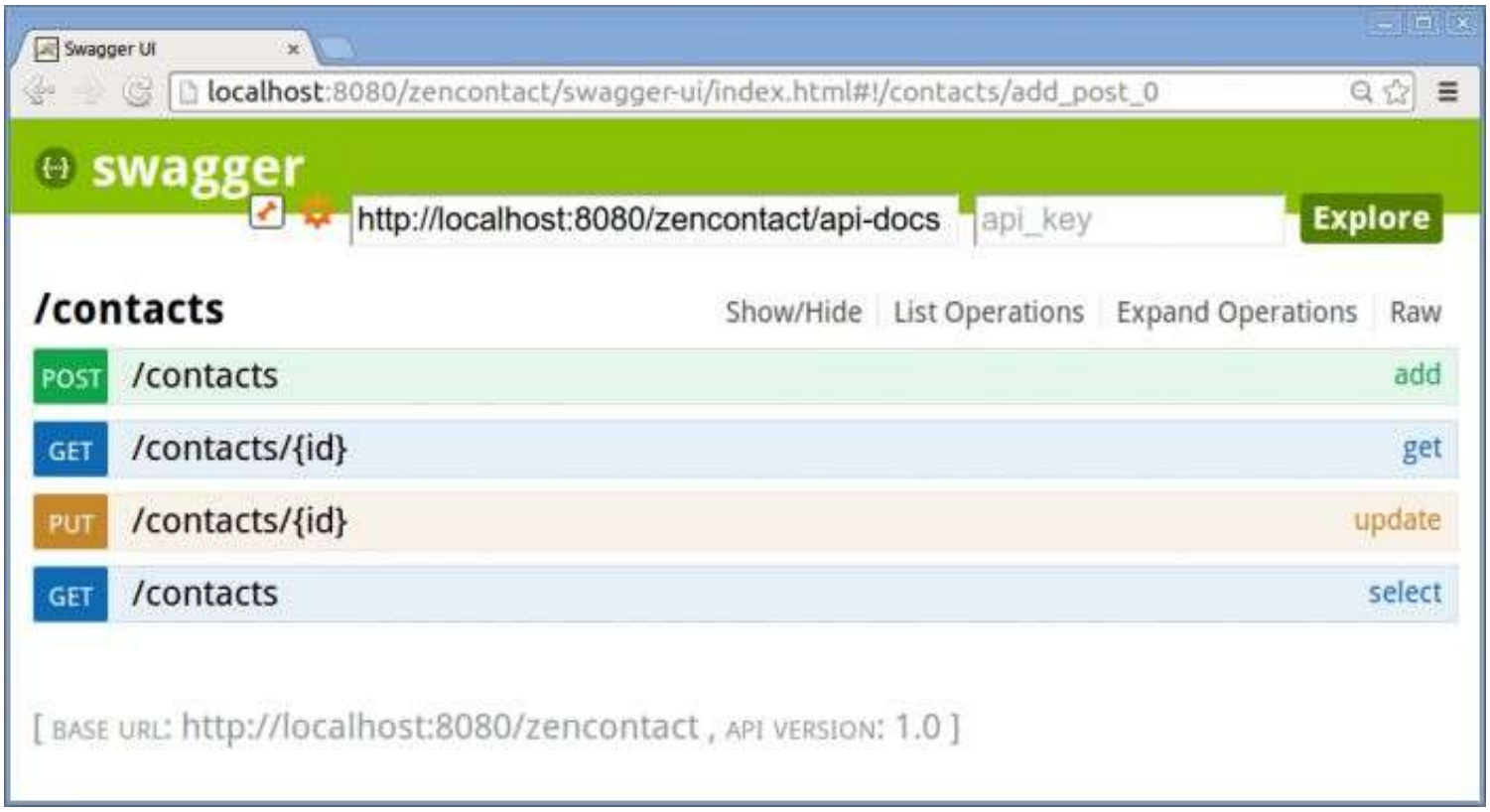
So see the Swagger documentation and HATEOAS in resources as complementary.

Let's see now another way to consume our documentation.

Swagger UI

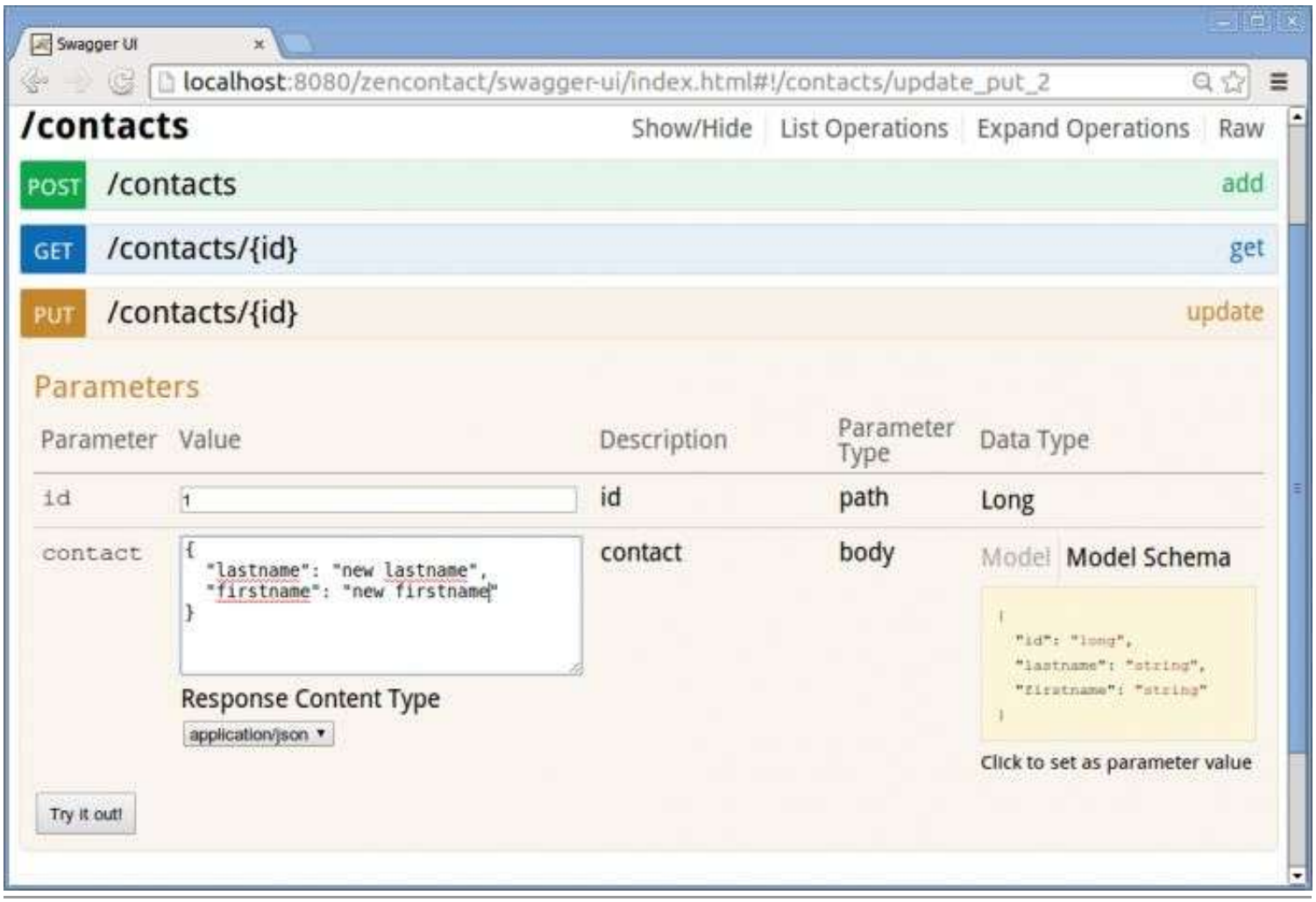
Have you ever used a written-by-hand web service documentation which isn't up-to-date? I guess we're all familiar with that. The API developers are busy writing the API, they don't have time or forget to update the documentation, and the API client developers are trying to figure out how to use the API, but the documentation is broken or obsolete. Nobody is happy.

Imagine now the developers that write client applications can consult a beautiful UI that tells them how to use the API. The UI even tells them what kind of documents the API expects. This UI exists and is called Swagger UI (https://github.com/wordnik/swagger-ui) :



Swagger UI just expects a URL that leads to a Swagger-compliant documentation. It then uses the documentation to display all the operations.

Swagger UI also inspects the models, so finding out about the structure of the JSON documents the API expects is straightforward. Here is an example for the PUT operation:



Note Swagger UI lets you try out operations and see the results.

Conclusion

Time to wrap up. I hope you're convinced tools like Swagger are the way to go to build real REST API and to get all the benefits this architecture style promises. Don't wait any longer and check how to include Swagger (https://developers.helloreverb.com/swagger/) into your project, as it has out-of-the-box support for various REST frameworks (JAX-RS/Spring MVC for Java, Django for Python, etc).

Source code (https://github.com/acogoluegnes/rest-documentation-with-swagger-spring-mvc)

Commentaires

1. Le dimanche 18 août 2013, 23:54 par Raphaël F.

   Doesn't WADL already describe the API of REST webservices? Many REST frameworks allow automatic generation of such descriptors:
   - https://wikis.oracle.com/display/Je... (https://wikis.oracle.com/display/Jersey/WADL)
   - http://cxf.apache.org/docs/jaxrs-se... (http://cxf.apache.org/docs/jaxrs-services-description.html#JAXRSServicesDescription-WADLAutoGeneration)

   The main benefit I see is Swagger UI which introduces an eye-candy and really usable web client to interact with webservices.

2. Le lundi 8 septembre 2014, 11:31 par Grahammkelly

   What about secured endpoints? How can one add the security parameters required? For example, REST? endpoints secured with Basic HTTP Authentication, etc.?

   Also, is there any way to restrict 'PUT' requests from being tried out?

Fil des commentaires de ce billet

Rechercher

ok

Recommander

— — — — —

A la une / A venir

- Introducing Spring MVC test framework
- Using Tomcat JDBC connection pool in a standalone environment
- AngularJS : la philosophie
- Intégrer ElasticSearch dans une application Java
- Creating a Varnish 4 module

Catégories

- Annonces Zenika (54)
- Architecture (17)
  - Cloud et Datagrids (20)
- Développement (90)
  - Web et Mobile (44)
  - Autres Langages (5)
  - Software Craftsmanship (8)
  - BigData (3)
- Tests et Performance (8)
- Usine logicielle (31)
- Reporting (8)
- Formations (10)
- Evénements (169)
- Méthode Agile (3)
  - Scrum (1)
- DevOps (9)

Tags

- java
- spring
- eclipse
- gradle
- birt
- html5
- google
- agile
- android
- gwt

Tous les mot-clés

Formations

- Java Specialist
- AngularJS
- Management 3.0
- Continuous Integration
- Découverte de l'Agilité
- BPEL
- Wicket

- Toutes nos formations...

Blogs

- The Coder's Breakfast
- Grégory Boissinot
- Nayima

Partenaires

- Actuate
- ej-technologies
- Nayima
- SpringSource
- Terracotta

Archives

- Accueil -
- Archives
