Rest API

Victor Herrero Cazurro

{ REST }

Contenidos

1. Rest

Los servicios REST son definidos en 2000 por Roy Fielding, coautor de la especificación HTTP, son servicios basados en recursos, montados sobre HTTP, donde:

- Los recursos son representados por un formato (MediaType).
- Cada recurso es accesible con una URL unica.
- Los Method HTTP indican las operaciones que se pueden realizar sobre los recursos y no las URL.
- Las respuestas emplearán los codigos de Estado HTTP.
- Se hará uso de las cabeceras HTTP para intercambiar metainformación entre cliente y API.
- Cada peticion se trata de forma inpedendiente, no se mantiene el estado conversacional.

La palabra REST viene de

- **Representacion**: Permite representar los recursos en multiples formatos, aunque el mas habitual es JSON.
- **Estado**: Se centra en el estado del recurso y no en las operaciones que se pueden realizar con el.
- **Transferencia**: Transfiere los recursos al cliente.

Los significados que se dan a los Method HTTP son:

- **POST**: Permite crear un nuevo recurso.
- **GET**: Permite leer/obtener un recurso existente.
- **PUT**: Permiten actualizar un recurso existente totalmente, es una sustitucion.
- **PATCH**: Permiten actualizar un recurso existente de forma parcial, solo los campos enviados.
- **DELETE**: Permite borrar un recurso.

2. Hipermedia

Termino acuñado por **Ted Nelson**, que extiende el concepto de **hipertexto**, permitiendo que contenidos de distinto tipo, audio, video, texto e hipervínculos, se

entrelacen para formar un continuo de información interactiva.

En el caso de una API REST, hipermedia, introduce la capacidad del API para ofrecer a la interfaz de usuario los enlaces adecuados para navegar sobre los recursos que proporciona el API.

3. Hateoas

El principio HATEOAS, que significa **Hypermedia As The Engine Of Application State**, define que los APIs REST, deberán retornar hipervínculos asociados a los recursos con los que este relacionado el recurso accedido, permitiendo al cliente seguir navegando con esos hipervinculos por los recursos del API.

Para entenderlo, supongamos que se tienen dos tipos de recursos: **Cliente** y **Pedidos**, donde cada **cliente** tienen asociados unos **pedidos** particulares.

Se podrian obtener los clientes con URL tal que

```
api/v1/clientes/{id}
```

Y la respuesta podria ser

Ejemplo de respuesta a la consulta **api/v1/clientes/2**

```
{
    id:2,
    nombre:"Victor",
    apellido:"Herrero",
    pedidos: [
        {
            id:1,
            descripcion:"regalos Navidad"
        },
        {
            id:3,
            descripcion:"regalo San Valentin"
        }
    ]
}
```

Esta primera aproximación, retorna quizas mas información de la necesaria, haciendo una referencia a Hibernate, esta resolviendo la relacion de forma **eager**,

cuando seria mas interesante que lo hiciese **lazy**.

Si se quitase la relacion, se perderia la referencia a los recursos, habria que acudir a la documentacion para poder obtener la URL a componer, que podria ser

```
api/v1/clientes/{id}/pedidos
```

Pero si se retornase la URL como respueta, se mantendria sin necesidad de acudir a la documentacion, pero sin saturar la respuesta.

Ejemplo de respuesta a la consulta **api/v1/clientes/2**

```
{
    id:2,
    nombre:"Victor",
    apellido:"Herrero",
    pedidos:"api/v1/clientes/2/pedidos"
}
```

4. JAX-RS

Especificacion Java para la definicion de servicios REST.

4.1. @Path

- @Path : Permite definir la URL para acceder al recurso

```
@Path("/users")
```

4.2. @GET

- @GET : Permite definir que para acceder al codigo la peticion debe ser con el method HTTP GET

```
@GET
```

4.3. @POST

- @POST : Permite definir que para acceder al codigo la peticion debe ser con el method HTTP POST

```
@POST
```

4.4. @PUT

- @PUT : Permite definir que para acceder al codigo la peticion debe ser con el method HTTP PUT

```
@PUT
```

4.5. @PATCH

- @PATCH : Permite definir que para acceder al codigo la peticion debe ser con el method HTTP PATCH

```
@PATCH
```

4.6. @DELETE

- @DELETE : Permite definir que para acceder al codigo la peticion debe ser con el method HTTP DELETE

```
@DELETE
```

4.7. @Consumes

- @Consumes : Permite definir en que formato han de llegar los datos en el Body de la Request

```
@Consumes("application/json")
```

4.8. @Produces

- @Produces : Permite definir en que formato se retornarán los datos en el Body de la Response

```
@Produces("application/json")
```

NOTE	Para la definicion de los MimeTypes, se puede recurrir a la clase **javax.ws.rs.core.MediaType**

4.9. @QueryParam

- @QueryParam : Permite definir un parametro de la request. El parametro será opcional.
- @DefaultValue : Permite definir un valor por defecto para el parametro en caso de que no se reciba

```
Path("smooth")
@GET
public Response smooth(@DefaultValue("2") @QueryParam("step") int step
){}
```

4.10. @PathParam

- @PathParam :

```
@Path("{id}")
public Response getUserById(@PathParam("id") String id) {
    ...
}
```

4.11. @MatrixParam

- @MatrixParam : Permite recoger parametros que se concatenan con el path, no es lo mismo que los QueryParam ya que estos se separan del path por ;

*Ante la peticion */libros/aventura;autor=Cervantes;pais=spain*

```
@Path("/libros")
public class LibroService {

    @GET
    @Path("{genero}")
    public Response getBooks(@PathParam("genero") String genero,
            @MatrixParam("autor") String autor,
            @MatrixParam("pais") String pais) {}
```

4.12. @HeaderParam

- @HeaderParam : Permite recoger las cabeceras que envia el cliente

```
@GET
  @Produces(MediaType.TEXT_HTML)
  public Response showHeaders(@HeaderParam("Accept") String accept,
                @HeaderParam("Host") String host,
                @HeaderParam("Cache-Control") String cache,
                @HeaderParam("User-Agent") String useragent,
                @HeaderParam("Referer") String referer) {}
```

4.13. @CookieParam

- @CookieParam : Permite recoger Cookies que se intercambien con el cliente

```
@GET
public Response getCookie(@CookieParam("name") Cookie cookie){}
```

4.14. @FormParam

- @FormParam : Permite recoger parametros enviados por un formulario HTML por POST, con el MIME-Type **application/x-www-form-urlencoded**

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void post(@FormParam("name") String name) {
    // Store the message
}
```

4.15. @Context

- @Context : en general se puede emplear para obtener los objetos en la request/response, como son: ServletConfig, ServletContext, HttpServletRequest y HttpServletResponse, aunque tambien para obtener objetos como UriInfo.

```
@Context
private UriInfo info;

@Context
private HttpServletRequest servletRequest;

@Context
private ServletContext servletContext;

@GET
public String get(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters
();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}

@GET
public String get(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = hh.getRequestHeaders
();
    Map<String, Cookie> pathParams = hh.getCookies();
}
```

4.16. @BeanParam

- @BeanParam : Permite definir un Bean con todos los recursos a extraer de la request, para unificar su recepcion en el método del **endpoint**.

```
@POST
public void post(@BeanParam MyBeanParam beanParam, String entity) {
    final String pathParam = beanParam.getPathParam(); // contiene el
path parameter "p"
    ...
}

public class MyBeanParam {
    @PathParam("p")
    private String pathParam;

    @MatrixParam("m")
    @Encoded
    @DefaultValue("default")
    private String matrixParam;

    @HeaderParam("header")
    private String headerParam;

    private String queryParam;

    public MyBeanParam(@QueryParam("q") String queryParam) {
        this.queryParam = queryParam;
    }

    public String getPathParam() {
        return pathParam;
    }
    ...
}
```

4.17. URIs

JAX-RS permite la creación de URIs relacionadas con los contenidos gracias a una clase **UriBuilder**, la cual puede obtenerse de la instancia de **UriInfo** que nos puede proporcionar el contendor.

```
@Path("/users/")
public class UsersResource {

    @Context
    UriInfo uriInfo;

    @GET
    @Produces("application/json")
    public JSONArray getUsersAsJsonArray() {
        JSONArray uriArray = new JSONArray();
        for (UserEntity userEntity : getUsers()) {
            UriBuilder ub = uriInfo.getAbsolutePathBuilder();
            URI userUri = ub.
                    path(userEntity.getUserId()).
                    build();
            uriArray.put(userUri.toASCIIString());
        }
        return uriArray;
    }
}
```

Otra forma de emplear **UriBuilder** es sin partir del path de la propia aplicacion, es decir sin **UriInfo**

```
UriBuilder.fromUri("http://localhost/")
    .path("{a}")
    .queryParam("name", "{value}")
    .build("segment", "value");
```

O aplicando plantillas

```
URI uri = UriBuilder.fromUri("http://{host}/{path}?q={param}")
    .resolveTemplate("host", "localhost")
    .resolveTemplate("path", "myApp")
    .resolveTemplate("param", "value").build();

uri.toString(); // returns "http://localhost/myApp?q=value"
```

4.18. Respuesta

La respuesta de los servicios puede ser directamente una **entidad/recurso**, que se representa en el cuerpo de la respuesta, o si ademas se quiere establer el codigo de respuesta y cabeceras, se generan empleando el objeto **javax.ws.rs.core.Response**.

```
@POST
@Consumes("application/xml")
public Response post(String content) {
  URI createdUri = ...
  String createdContent = create(content);
  return Response
    .status(201)
    .contentLocation(new URI("/user-management/users/123")).
  .build();
}
```

Este objeto tienes diversos métodos para establecer las distintas posibles caracteristicas

- status()
- contentLocation()
- cacheControl()
- cookie()
- encoding()
- entity()
- expires()
- header()
- language()
- link()
- tag()
- type()
- variants()

5. Jersey

Para trabajar con Jersey como implementacion de JAX-RS, se ha de añadir la siguiente dependencia

```
<dependencies>
    <dependency>
        <groupId>org.glassfish.jersey.containers</groupId>
        <artifactId>jersey-container-servlet</artifactId>
        <version>2.25.1</version>
    </dependency>
</dependencies>
```

Y se ha de declarar el siguiente Servlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns=
"http://java.sun.com/xml/ns/javaee" xsi:schemaLocation=
"http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID"
version="3.0">
    <display-name>com.vogella.jersey.first</display-name>
    <servlet>
        <servlet-name>Jersey REST Service</servlet-name>
        <servlet-class>
org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <!-- Se indica en que paquete estan los resources y providers
-:
        <init-param>
            <param-name>jersey.config.server.provider.packages</param-
name>
            <param-value>com.ejemplo.jersey</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Jersey REST Service</servlet-name>
        <url-pattern>/api/v1/*</url-pattern>
    </servlet-mapping>
</web-app>
```

El escaneo de los paquetes, se puede configurar para que no sea recursivo.

```
<init-param>
    <param-name>
jersey.config.server.provider.scanning.recursive</param-name>
    <param-value>false</param-value>
</init-param>
```

Tambien se pueden declarar una a una las clases de los servicios

```
<init-param>
    <param-name>jersey.config.server.provider.classnames</param-name>
    <param-value>
        org.foo.myresources.MyDogResource,
        org.bar.otherresources.MyCatResource
    </param-value>
</init-param>
```

5.1. configuracion a partir de Servlet 3.0

Para el despliegue en contenedores a partir de la version 3.0 del API de Servlets, se puede prescindir del **web.xml** en favor de una clase que extienda de la jerarquia **javax.ws.rs.core.Application**.

```
@ApplicationPath("resources")
public class MyApplication extends ResourceConfig {
    public MyApplication() {
        packages("org.foo.rest;org.bar.rest");
    }
}
```

En esta clase se puede hacer uso de métodos como **packages** o **files** para registrar los paquetes a escanear en busca de las clases de **Servicios**/**Recursos**

5.2. Soporte para JSON

Para añadir soporte para la transformacion a/desde JSON, se necesita la siguiente dependencia

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.26</version>
</dependency>
```

Y añadir el siguiente parametro de configuracion al **ServletContainer**

```
<init-param>
    <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
    <param-value>true</param-value>
</init-param>
```

5.3. Soporte para HATEOAS

Se ha de añadir la siguiente dependencias de Maven

```
<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-declarative-linking</artifactId>
    <version>2.26</version>
</dependency>
<dependency>
    <groupId>javax.el</groupId>
    <artifactId>javax.el-api</artifactId>
    <version>2.2.4</version>
</dependency>
<dependency>
    <groupId>org.glassfish.web</groupId>
    <artifactId>javax.el</artifactId>
    <version>2.2.4</version>
</dependency>
```

Además se ha de añadir el siguiente parametro de configuracion al **ServletContainer**

```
<init-param>
    <param-name>jersey.config.server.provider.classnames</param-name>
    <param-value>
org.glassfish.jersey.linking.DeclarativeLinkingFeature</param-value>
</init-param>
```

Ua vez configurado, este API, permite definir campos de tipo **javax.ws.rs.core.Link** en los **recursos**, que representen relaciones entre recursos a traves de la **URL** que permite acceder al recurso en concreto.

Así se puede definir una clase que representa un recurso, con un campo de tipo **Link** anotado con **@InjectLink**

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Cliente implements Serializable{
    @InjectLink(
            resource = ClienteRestService.class,
            style = Style.ABSOLUTE,
            rel = "self",
            bindings = @Binding(name = "id", value = "${instance.id}"),
            method = "consultarPorId"
    )
    @XmlJavaTypeAdapter(Link.JaxbAdapter.class)
    @XmlElement(name = "self")
    Link self;
}
```

Donde en la anotacion **@InjectLink**, se definen:

- **resource** : clase que expone el servicio REST asociado con la entidad con la que se establece la relacion
- **style** : Formato de la URL
- **rel** : Valor de la caracteristica **rel**, define el recurso al que esta asociado, **self** indica que a si mismo.
- **bindings** : Sustitucion de variables en la URL a generar con valores presentes en el actual recurso, se puede emplear **instance** como referencia al actual recurso.
- **method** : Nombre del método en la clase representada por **resource** al que

apunta la URL generada.

- **Extensions** : Posibilidad de añadir mas metainforacion en formato claves/valores al enlace

La siguiente clase define el servicio al que la anterior configuracion hace referencia.

```
@Path("/Cliente")
public class ClienteRestService {

    @GET
    @Path("{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Cliente consultarPorId(@PathParam("id") long id) {
        return new Cliente(1, "Victor");
    }
}
```

Para establecer varios enlaces se emplea @**InjectLinks** esta causistica se suele emplear para agrupar enlaces que permiten realizar operaciones sobre un recurso particular (alta, baja, modificacion, ..)

```
@InjectLinks({
    @InjectLink(
            resource = ClienteRestService.class,
            style = Style.ABSOLUTE,
            rel = "self",
            bindings = @Binding(name = "id", value = "${instance.id}"),
            method = "consultarPorId"
    ),
    @InjectLink(
            resource = ClienteRestService.class,
            style = Style.ABSOLUTE,
            rel = "self",
            bindings = @Binding(name = "id", value = "${instance.id}"),
            method = "crear",
            extensions = {
                    @Extension(name = "method", value = "POST")
            }
    )
})
@XmlJavaTypeAdapter(Link.JaxbAdapter.class)
@XmlElement(name = "self")
List<Link> links;
```

Otra forma alternativa de definir los enlaces a otros recursos, es en el propio servicio con la anotacion **@ProvideLink**

```
@ProvideLink(
        value = Factura.class,
        rel = "otra",
        bindings = @Binding(name = "id", value = "${instance.id}"))
@GET
@Path("{id}")
@Produces(MediaType.APPLICATION_JSON)
public Cliente consultarPorId(
                @PathParam("id") long id,
                @DefaultValue("Victor") @QueryParam("nombre") String
nombre) {
    System.out.println("Nombre: " + nombre);
    return new Cliente(1, "Victor");
}
```

Donde las propiedades definidas significan lo mismo que en la anotacion

@**InjectLink**, salvo que no aparece **resource**, pero si aparece en cambio **value**.

- **value** : La entidad de la respuesta sobre la que se aplicará el enlace

NOTE	Disponible a partir de la version 2.26 de Jersey

5.4. WADL

Jersey soporta de forma nativa WADL, ofreciendo un endpoint con dicho fichero.

```
http://<host>:<port>/<app>/<servlet-mapping>/application.wadl

http://localhost:8080/JAX-RS-RAML/api/v1/application.wadl
```

5.5. Integracion de Jersey con Spring

Se ha de partir de un proyecto de Jersey normal, con la configuracion habitual, ya sea con la declaracion del **SErvletContainer** o con la clase **Application**.

A mayores se ha de definir una nueva dependencia de Maven

```
<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-spring4</artifactId>
    <version>2.26</version>
</dependency>
```

Este jar, incluye el registro del **ContextLoaderListener** de Spring en el contexto web, a traves de una clase que implementa **WebApplicationInitializer**, que busca la definicion del contexto de Spring en un fichero **applicationContext.xml** en el classpath.

Por lo tanto, lo unico que habrá que hacer será definir dicho fichero con la configuracion de beans que ha de gestionar Spring.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-
3.0.xsd">

    <context:component-scan base-package=
"com.ejemplo.jax_rs.spring.core.service" />

</beans>
```

E incluir la referencia a dichos Beans en los Servicios Rest de JAX-RS, para lo cual se puede emplear la anotacion de Spring @**Autowired** o la del estandar @**Inject**

```
@Path("/hello")
public class Hello {

    @Inject
    Servicio servicio;

    // This method is called if TEXT_PLAIN is request
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayPlainTextHello() {
        servicio.hacerCosas();
        return "Hello Jersey";
    }
}
```

5.6. Clientes

Jersey proporciona un API para consumir APIs Rest, para poder emplearlo se ha de añadir la dependencia de Maven

```
<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
    <version>2.26</version>
</dependency>
```

Si se necesita sporte para JSON, tambien la depedencia

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
    <version>2.26</version>
</dependency>
```

Lo primero será crear un objeto **Client**

```
Client client = ClientBuilder.newClient();
```

Una vez definido este objeto, se establece el path base del servicio a consumir

```
WebTarget webTarget = client.target(
"http://localhost:8080/EjemploJersey/api");
```

El cual se puede ir reutilizando definiendo subpaths

```
WebTarget employeeWebTarget = webTarget.path("clientes/" + id);
```

Una vez establecido el Path, se genera un objeto **Invocation.Builder**

```
Invocation.Builder invocationBuilder = employeeWebTarget.request
(MediaType.APPLICATION.JSON);
```

Y sobre este se realiza la peticion indicando el **METHOD HTTP**

```
//GET
Response response = invocationBuilder.get(Cliente.class);

//POST
Response response = invocationBuilder.post(Entity.entity(cliente,
MediaType.APPLICATION_JSON);
```

Es habitual la concatenacion de los métodos

```
//GET
client
    .target("http://localhost:8080/EjemploJersey/api")
    .path("clientes/" + id)
    .request(MediaType.APPLICATION_JSON)
    .get(Cliente.class);

//POST
client
      .target("http://localhost:8080/EjemploJersey/api")
      .request(MediaType.APPLICATION_JSON)
      .post(Entity.entity(cliente, MediaType.APPLICATION_JSON));
```

5.7. Validaciones

Se ha de partir de un proyecto de Jersey normal, con la configuracion habitual, ya sea con la declaracion del **ServletContainer** o con la clase **Application**.

A mayores se ha de definir una nueva dependencia de Maven

```
<dependency>
    <groupId>org.glassfish.jersey.ext</groupId>
    <artifactId>jersey-bean-validation</artifactId>
    <version>${jersey.version}</version>
</dependency>
```

Y activar una propiedad

```
property(ServerProperties.BV_SEND_ERROR_IN_RESPONSE, true);
```

6. RAML

RAML que significa **RESTful API Modeling Language**, es un formato de modelado de APIs REST, basado en YAML y JSON para las estructuras de datos (schemas).

NOTE	Al estar basado en YAML, no se han de incluir tabulaciones, sino espacios en blanco para la identacion

La idea es poder definir con un fichero **raml** la estructura del API, sin implementarlo y posteriormente generar el esquelo del API con herramientas que sean capaces de interpretarlo.

La configuracion básica para la definicion del contrato del servicio es

```
#%RAML 0.8
title: Mi API
baseUri: http://www.api.com/api/{version}
version: v1
```

El documento al estar basado en YAML es jerarquico, y las configuraciones que incluyamos en el primer nivel, se heredaran en el resto

La definicion del tipo de contenido en el primer nivel, hace que sea el por defecto.

```
mediaType: application/json
```

6.1. Recursos

Los recursos se definen comenzando por **/** y son anidables

```
/simpsons:
  /personajes:
    /{id}:
  /actores:
  /capitulos:
```

Dentro de los recursos, en el siguiente nivel, se establecen los METHOD HTTP soportados para el recurso

```
/simpsons:
  /personajes:
    get:
    put:
    post:
```

Los métodos tendran caracteristicas que los describan, como

- description : Texto descriptivo de lo que se consigue con la invocacion de ese endpoint
- queryParams : Parametros que es capaz de interpretar el endpoint y que permite configurar como se llevará a cabo la acción
- responses : Indica las posibles respuestas
- body : Para los METHOD POST y PUT, se puede enviar informacion el el cuerpo de la request.

6.1.1. Query Params

Se puede indicar una descripcion del parametro.

El tipo de los **queryParams** por defecto es **string**, pero hay mas posibilidade: date, boolean, number, integer y file, para indicarlo:

```
/simpsons:
  /personajes:
    get:
        description: Permite obtener todos los personajes de la serie
de los Simpsons
        queryParameters:
            apellido:
                description: apellido del personaje
                type: string
```

Otras caracteristicas que se pueden definir para cada **queryParam** son:

- example : Un ejemplo del valor que se le puede dar
- required : Indica si el parametro es obligatorio

```
/simpsons:
  /personajes:
    get:
        description: Permite obtener todos los personajes de la serie
de los Simpsons
        queryParameters:
            apellido:
                example:  *Simpson*
                required: false
```

6.1.2. Responses

Se pueden emplear JSON o XML para la descripcion del **Schema**.

Se puede establecer un ejemplo de la respuesta.

```
/simpsons:
  /personajes:
    get:
      description: Permite obtener todos los personajes de la serie de
los Simpsons
      queryParameters:
        apellido:
          description: apellido del personaje
      responses:
        200:
          body:
            schema: |
                {
                    "$schema": "http://json-schema.org/draft-
03/schema",
                    "type": "array",
                    "description": "personajes",
                    "items":
                    {
                        "type": "object",
                        "properties": {
                            "nombre": { "type": "string" },
                            "apellido": { "type": "string"}
                        }
                    }
                }
            example: |
```

```
                [
                    {
                        "nombre" : "Hommer",
                        "apellido": "Simpson"
                    },
                    {
                        "nombre" : "Marge",
                        "apellido": "Simpson"
                    },
                    {
                        "nombre" : "Ned",
                        "apellido": "Flanders"
                    }
                ]
    /{nombre}:
      get:
        responses:
          200:
            description: retorna un personaje por su nombre
          404:
            description: no se encontró un personaje con ese nombre
```

6.1.3. Body

Al igual que para las **responses**, para el **body, se puede definir un *schema**

```
/simpsons:
  /personajes:
    post:
      body:
        schema: |
            {
                "$schema": "http://json-schema.org/draft-03/schema",
                "type": "object",
                "description": "personaje",
                "properties":
                    {
                        "nombre": { "type": "string" },
                        "apellido": { "type": "string"}
                    }
            }

//-------------------------------------------------------------------
------------------
//-------------------------------------------------------------------
------------------
=== RAML + JAX-RS
//-------------------------------------------------------------------
------------------
//-------------------------------------------------------------------
------------------

//https://github.com/mulesoft-labs/raml-for-jax-rs/blob/master/jaxrs-
to-raml/README.md
//https://github.com/mulesoft-labs/raml-for-jax-rs/blob/master/raml-to-
jaxrs/README.md

//-------------------------------------------------------------------
------------------
//-------------------------------------------------------------------
------------------

Existen dos plugins de Maven, que permiten integrar *JAX-RS* con
*RAML*.

* raml-to-jaxrs-maven-plugin
* jaxrs-to-raml-maven-plugin

[source, maven]
.Configuracion de *raml-to-jaxrs-maven-plugin*
```

<build> <plugins> <plugin> <groupId>org.raml</groupId> <artifactId>raml-to-jaxrs-maven-plugin</artifactId> <version>2.0.0</version> <dependencies> <dependency> <groupId>org.raml</groupId> <artifactId>jaxrs-code-generator</artifactId> <version>2.0.0</version> </dependency> </dependencies> <configuration> <ramlFile> ${project.build.resources[0].directory}/types_user_defined.raml </ramlFile> <resourcePackage>example.resources</resourcePackage> <modelPackage>example.model</modelPackage> <supportPackage>example.support</supportPackage> <generateTypesWith> <value>jackson</value> </generateTypesWith> </configuration> </plugin> </plugins> </build>

```
Las opciones posibles para el parametro *generateTypesWith* son:
jackson, gson, jaxb, javadoc y jsr303.

La tarea para generar el codigo

[source, console]
```

mvn raml:generate ---

Configuracion de **jaxrs-to-raml-maven-plugin**

```
<build>
    <plugins>
        <plugin>
            <groupId>org.raml.jaxrs</groupId>
            <artifactId>jaxrs-to-raml-maven-plugin</artifactId>
            <version>2.1.1</version>
            <dependencies>
                <dependency>
                    <groupId>org.raml.jaxrs</groupId>
                    <artifactId>jaxrs-to-raml-methods</artifactId>
                    <version>2.1.1</version>
                </dependency>
            </dependencies>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>jaxrstoraml</goal>
                    </goals>
                    <configuration>
                        <input>${project.build.outputDirectory}</input>

<outputFileName>${project.artifactId}.raml</outputFileName>

<sourceDirectory>${project.build.sourceDirectory}</sourceDirectory>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

La tarea para generar el codigo es **jaxrstoraml**, que se puede integrar en el ciclo de vida de Maven, por ejemplo en la fase de **package**.

```
mvn package
---

//include::contenido/wadl.adoc[]
```