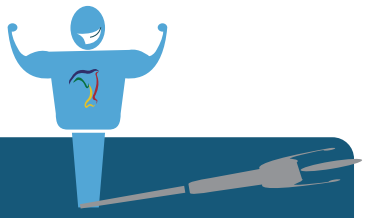
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

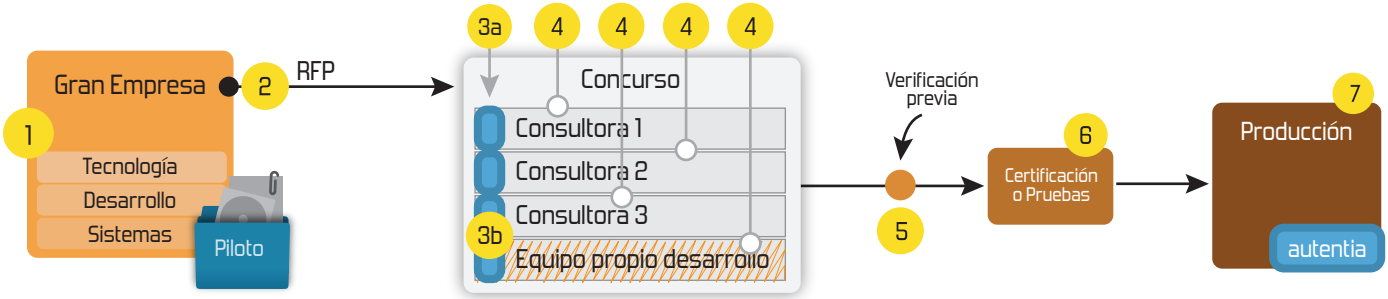
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

- Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery
- Gestor portales (Liferay)
- Gestor de contenidos (Alfresco)
- Aplicaciones híbridas
- Control de autenticación y acceso (Spring Security)
- UDDI
- Web Services
- Rest Services
- Social SSO
- SSO (Cas)
- JPA-Hibernate, MyBatis
- Motor de búsqueda empresarial (Solr)
- ETL (Talend)
- Dirección de Proyectos Informáticos.
- Metodologías ágiles
- Patrones de diseño
- TDD
- Tareas programadas (Quartz)
- Gestor documental (Alfresco)
- Inversión de control (Spring)
- BPM (jBPM o Bonita)
- Generación de informes (JasperReport)
- ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

autentia
Soporte a desarrollo informático
Hosting patrocinado por enredados

Entra en Adictos a través de
E-mail
Contraseña
Entrar
Deseo registrarme
Olvidé mi contraseña

Inicio Quiénes somos Formación Comparador de salarios Nuestros libros Más

» Estás en: Inicio Tutoriales REST, el principio HATEOAS y Spring HATEOAS

Miguel Arlandy Rodríguez

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/JEE

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2013-12-02

Tutorial visitado 1 veces Descargar en PDF

REST, el principio HATEOAS y Spring HATEOAS.

0. Índice de contenidos.

- 1. Introducción.
- 2. Entorno.
- 3. ¿Qué es HATEOAS?.
- 4. Añadiendo links a la representación de nuestros recursos con Spring HATEOAS.
- 5. ¿Vemos un ejemplo?.
  - 5.1. Respuestas sin links.
  - 5.2. Respuestas que siguen el principio HATEOAS
- 6. Referencias.
- 7. Conclusiones.

1. Introducción

Poco podemos decir de REST que no hayamos comentado ya. Su simplicidad hace que muchos desarrolladores opten por esta solución por encima de SOAP a la hora de exponer las diferentes funcionalidades (mejor dicho, recursos) de sus plataformas. La **simplicidad** es, sin lugar a dudas, su gran arma pero, **¿abusamos de ella?**, ¿diseñamos verdaderas APIs REST o simples interfaces sobre HTTP?, ¿conocemos realmente los principios de diseño de servicios RESTful?.

En este tutorial intentaremos explicar qué es el principio HATEOAS, de obligado cumplimiento para cualquier API REST que se enorgullezca de serlo y veremos, mediante un ejemplo, el soporte que nos proporciona Spring para conseguirlo gracias a Spring HATEOAS.

2. Entorno.

El tutorial está escrito usando el siguiente entorno:

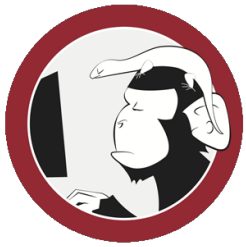
- Hardware: Portátil MacBook Pro 15' (2.2 Ghz Intel Core I7, 8GB DDR3).
- Sistema Operativo: Mac OS Mountain Lion 10.8
- Entorno de desarrollo: Intellij Idea 11.1 Ultimate.
- Apache Tomcat 7.0.47
- Maven 3.0.3
- Java 1.7.0_45
- Spring 3.2.4.RELEASE
- Spring HATEOAS 0.8.0.RELEASE
- H2 database 1.3.170

3. ¿Qué es HATEOAS?.

HATEOAS es un acrónimo de **Hypermedia As The Engine Of Application State** (hipermedia como motor del estado de la aplicación). Significa algo así como que, dado un punto de entrada genérico de nuestra API REST, podemos ser capaces de descubrir sus recursos basándonos únicamente en las respuestas del servidor. Dicho de otro modo, cuando el servidor nos devuelva la representación de un recurso (JSON, XML...) parte de la información devuelta serán identificadores únicos **en forma de hipervínculos** a otros recursos asociados.

Lo vamos a entender más fácilmente con este ejemplo. Imaginemos que tenemos un API de un concesionario de coches

Catálogo de servicios Autentia



Síguenos a través de:

Últimas Noticias

» IX Autentia Cycling Day (ACTUALIZADO)

» Spring 4.0 ¿qué hay de nuevo amigo?

» Torneo de pádel solidario AMEB

» Próxima charla: Gradle como alternativa a Maven para la construcción de proyectos en Java

» Conferencias Big Data Spain

Histórico de noticias

Últimos Tutoriales

» Ponle color a tu maven

» Primeros pasos con MongoDB

» Introducción Android

» Ejecutando MyBatis contra distintas bases de datos

» Minimizar código con anotaciones en Spring.

donde nuestros clientes, evidentemente, compran coches. Supongamos que queremos obtener los datos del cliente con **id** igual a 78. Haríamos una petición de este estilo:

```
Request URL: http://miservidor/concesionario/api/v1/clientes/78
Request Method: GET
Status Code: 200 OK
```

Y obtendríamos algo como:

```
{
    "id": 78,
    "nombre": "Juan",
    "apellido": "García",
    "coches": [
        {
            "id": 1033
        },
        {
            "id": 3889
        }
    ]
}
```

Con esto ya sabemos que nuestro cliente compró dos coches pero, **¿cómo accedemos a la representación de esos dos recursos?**. Sin consultar la documentación del API no tenemos forma de obtener la URL que identifique de forma única a cada uno de los coches. Además, aunque supiésemos conformar la URL de acceso a los recursos, cualquier cliente que quisiese consumir los recursos debería tener la **responsabilidad de construir dicha URL**. Por último, **¿qué ocurriría si la URL cambiase?**, habría que cambiar todos los clientes que consumen los recursos.

Siguiendo el principio HATEOAS la respuesta sería algo como:

```
{
    "id": 78,
    "nombre": "Juan",
    "apellido": "García",
    "coches": [
        {
            "coche": "http://miservidor/concesionario/api/v1/clientes/78/coches/1033"
        },
        {
            "coche": "http://miservidor/concesionario/api/v1/clientes/78/coches/3889"
        }
    ]
}
```

De esta forma, ya sabemos dónde debemos ir a buscar los recursos relacionados (coches) con nuestro recurso original (cliente) gracias a la respuesta del servidor (hypertext-driven). Sin seguir este principio, nuestra API nunca seguirá el verdadero estilo arquitectónico REST. Y no lo digo solo yo, lo dice su principal promotor Roy Fielding.

Spring HATEOAS es un pequeño módulo perteneciente al "ecosistema Spring" que nos ayudará a crear APIs REST que respeten el principio HATEOAS.

4. Añadiendo links a la representación de nuestros recursos con Spring HATEOAS.

Añadir links a nuestros recursos (mejor dicho, a sus representaciones) es muy sencillo con Spring HATEOAS. La única condición de que debemos cumplir es que el objeto u objetos que devolvamos en la respuesta extiendan de **org.springframework.hateoas.ResourceSupport**. Una vez que cumplan con esto ya podremos añadir a nuestro recurso los links que apunten a las URLs de otros recursos de nuestra API con los que queramos relacionarlo.

Supongamos que tenemos una clase PersonWrapper que representará un recurso "persona":

```
@XmlRootElement(name = "person")
public class PersonWrapper extends ResourceSupport {

    private String name;

    private int age;

    private PersonWrapper() {}

    public PersonWrapper(Person person) {
        this.name = person.getName();
        this.age = person.getAge();
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

En nuestro controlador **PersonController** tendremos un método que devolverá un recurso persona que coincida con el nombre solicitado. Algo como lo siguiente:

```
@Controller
public class PersonController {

    @RequestMapping(value = "/persons/{name}", method = RequestMethod.GET)
    public @ResponseBody PersonWrapper getPerson(@PathVariable String name) {
```


Últimos Tutoriales del Autor

» SOA y los tipos de servicios

» Introducción a Spring Batch

» Iconos increíbles para nuestra web con Font Awesome Icons

» WebSockets con Stomp y ActiveMQ: ¿chateamos?

» AngularJS: primeros pasos.

Últimas ofertas de empleo

2011-09-08
Comercial - Ventas - MADRID.

2011-09-03
Comercial - Ventas - VALENCIA.

2011-08-19
Comercial - Compras - ALICANTE.

2011-07-12
Otras Sin catalogar - MADRID.

2011-07-06
Otras Sin catalogar - LUGO.

```
        final Person person = getPersonByName(name);
        final PersonWrapper wrapper = new PersonWrapper(person);
        return wrapper;
    }

}
```

A través de la URI **/persons/pepe** obtendríamos la representación (XML, JSON o lo que sea) de la persona con nombre igual a pepe.

Ahora supongamos que queremos añadir un link a nuestro recurso que apunte a sí mismo. La forma de hacerlo es muy sencilla. La clase ResourceSupport de la que heredamos viene con un método **add** que recibe como parámetro un objeto de tipo **org.springframework.hateoas.Link** que representará un enlace a cualquier recurso. Dicho enlace sigue el estandar Atom para links.

Añadir el link a nuestro recurso que apunte a sí mismo es muy fácil. Tan fácil como invocar a este nuevo método antes de devolver el recurso:

```
private void addSelfLink(PersonWrapper resource){
    final PersonWrapper person = methodOn(PersonController.class).getPerson(resource.getNar
    final Link link = linkTo(person).withSelfRel();
    resource.add(link);
}
```

Creo que el código habla por sí solo, pero por si no queda claro vamos a explicarlo. Lo que hacemos es crear un link que apunta recurso que daría como resultado la invocación del método correspondiente en el controlador correspondiente. Nótese que tanto *methodOn* como *linkTo* son métodos estáticos de la clase **org.springframework.hateoas.mvc.ControllerLinkBuilder**. El resultado tras añadir el link nos daría una respuesta como la que sigue:

Respuesta en formato JSON:

```
{
    "links": [{
        "rel": "self",
        "href": "http://localhost:8080/myapi/v1/persons/pepe"
    }],
    "name": "pepe",
    "age": 25
}
```

Respuesta en formato XML:

```
<person xmlns:atom="http://www.w3.org/2005/Atom">
    <atom:link rel="self" href="http://localhost:8080/myapi/v1/persons/pepe"/>
    <age>25</age>
    <name>luis</name>
</person>
```

¿Existen otras formas de añadir links a nuestros recursos?

Pues la respuesta es sí y no. No porque todas pasan irremediablemente (al menos que yo sepa) por devolver un elemento del tipo ResourceSupport. Y sí porque Spring HATEOAS nos porporciona mecanismos para devolver estos elementos de tipo ResourceSupport de una manera **más elegante**. Al menos, existen estas dos alternativas:

- Hacer uso de la clase **org.springframework.hateoas.Resource** que ya extiende de ResourceSupport y que nos permite no tener que crear nosotros los envoltorios puesto que Resource ya es, en sí mismo, un envoltorio. Yo, personalmente, tuve una mala experiencia usando esta clase con representaciones XML. Aquí os dejo un enlace para que que quiera saber más.
- Hacer uso de la clase **org.springframework.hateoas.mvc.ResourceAssemblerSupport**, que básicamente es un conversor de POJOS que usemos en nuestro modelo de datos a objetos del tipo ResourceSupport. Probablemente esta **es la mejor solución** porque nos permite separar el proceso de conversión e inserción de links en una clase independiente (principio de responsabilidad única).

5. ¿Vemos un ejemplo?.

A continuación veremos un ejemplo que nos ayudará a asimilar mejor todos estos conceptos. Para ello crearemos un **API REST muy futbolera** :-). Estará compuesta de tres recursos distintos: **equipos, estadios y jugadores**, que estarán relacionados de la siguiente forma:

- Un equipo juega en un estadio.
- Un equipo está compuesto por un conjunto de jugadores.
- Cada jugador solo puede jugar en un equipo.
- En un estadio solo juega un equipo.

	Resource	Descripción
GET	/teams	Devuelve el listado de equipos
POST	/teams	Da de alta un equipo. Es necesario enviar en el cuerpo de la peticón los siguientes campos: *name*, *foundationYear* y *rankingPosition*.
GET	/teams/:id	Devuelve el equipo cuyo identificador coincida con *:id*
GET	/teams/:id/stadium	Devuelve el estadio del equipo cuyo identificador coincida con *:id*
POST	/teams/:id/stadium	Da de alta el estadio del equipo cuyo identificador coincida con *:id*. Es necesario enviar en el cuerpo de la petición los siguientes campos: *capacity*, *name* y *city*.
GET	/teams/:id/players	Devuelve los jugadores del equipo cuyo identificador coincida con *:id*
POST	/teams/:id/players	Da de alta un jugador perteneciente al equipo cuyo identificador coincida con *:id*. Es necesario enviar en el cuerpo de la petición los siguientes campos: *name*, *goals*, *country* y *age*.
GET	/teams/:id/players/:id_player	Devuelve los datos del jugador que juegue en un equipo cuyo identificador coincida con *:id* y con un identificador de jugador que coincida con *:id_player*.

Además, todas las peticiones y respuestas (body) soportarán **json y xml** (application/json y application/xml media types).

5.1. Respuestas sin links.

Si no seguimos el principio HATEOAS tendríamos el siguiente método en nuestro controlador que se encargaría de recibir las peticiones GET solicitando un equipo con un identificador dado.

```
@RequestMapping(value = "/teams/{id}", method = RequestMethod.GET)
public @ResponseBody Team getById(@PathVariable int id) {
    LOG.trace("Recibida solicitud para devolver el equipo con id {}", id);
    final Team team = teamDao.getById(id);
    if (team == null) {
        LOG.warn("El equipo con id {} no existe", id);
        throw new ResourceNotFoundException();
    }

    return team;
}
```

Tras realizar una la petición solicitando el equipo con id = 5000

```
Request URL: http://localhost:8080/soccer/api/teams/5000
Request Method: GET
Status Code: 200 OK
```

Obtendríamos un respuesta de este tipo:

```
{
    "teamId": 5000,
    "name": "Real Madrid C.F.",
    "foundationYear": 1902,
    "rankingPosition": 1
}
```

Como dijimos anteriormente, un equipo tiene asociado un **estadio** y un conjunto de **jugadores**, pero como cliente del API ¿cómo puedo saber cómo acceder a estos elementos asociados?. Sin seguir el principio HATEOAS, la única forma posible es **consultar la documentación** para saber que un equipo tiene un estadio y un conjunto de jugadores además de las URL´s donde residen los mismos. Además, esas URLs las deberíamos construir nosotros.

5.2. Respuestas que siguen el principio HATEOAS.

Vamos a modificar ligeramente nuestro código para que, haciendo uso de Spring-HATEOAS, podamos devolver los links que apuntan tanto al estadio como al listado de jugadores asociados a un equipo. Haremos lo siguiente:

- Nuestra clase team ahora extenderá de **ResourceSupport** y heredará el soporte para que podamos añadir links asociados al equipo.
- Añadimos los links del estadio y del listado de jugadores al equipo apuntando directamente a los métodos de sus correspondientes controladores, de esta forma, si cambiase la URL del recurso o su método HTTP no tendríamos que tocar nada en nuestro código.

OJO!!! probablemente ésta no sería la mejor forma en lo que a términos de diseño se refiere puesto que estamos mapeando directamente la salida de un DAO (Team) a la representación del recurso (se acabó el bajo acoplamiento). Lo ideal sería hacer uso de **ResourceAssemblerSupport** y devolver un wrapper. Sin embargo optamos por esta otra opción por simplicidad en el código.

Quedaría algo así (recordemos que ahora Team extiende de ResourceSupport):

```
@RequestMapping(value = "/teams/{id}", method = RequestMethod.GET)
public @ResponseBody Team getById(@PathVariable int id) {
    LOG.trace("Recibida solicitud para devolver el equipo con id {}", id);
    final Team team = teamDao.getById(id);
    if (team == null) {
        LOG.warn("El equipo con id {} no existe", id);
        throw new ResourceNotFoundException();
    }
    addTeamLinks(team);
    return team;
}

private void addTeamLinks(Team team) {
    addSelfLink(team);
    addStadiumLink(team);
    addPlayerLink(team);
}

private void addSelfLink(Team team) {
    team.add(linkTo(methodOn(TeamController.class).getById(team.getTeamId())).withSelfRel
}

private void addStadiumLink(Team team) {
    team.add(linkTo(methodOn(StadiumController.class).getByTeamId(team.getTeamId())).withI
}

private void addPlayerLink(Team team) {
    team.add(linkTo(methodOn(PlayerController.class).getTeamPlayers(team.getTeamId())).wi
}
```

Y, tras la misma petición que hicimos en el punto anterior, el resultado sería este:

```
{
    "links": [{
        "rel": "self",
        "href": "http://localhost:8080/soccer/api/teams/5000"
    }, {
```

```
            "rel": "stadium",
            "href": "http://localhost:8080/soccer/api/teams/5000/stadium"
        }, {
            "rel": "players",
            "href": "http://localhost:8080/soccer/api/teams/5000/players"
        }],
        "teamId": 5000,
        "name": "Real Madrid C.F.",
        "foundationYear": 1902,
        "rankingPosition": 1
    }
```

Como se puede apreciar, con una simple petición para obtener un equipo, podemos saber dónde residen sus recursos asociados. Si quisiésemos que la respuesta fuese XML, el resultado sería:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<team>
    <atom:link rel="self" href="http://localhost:8080/soccer/api/teams/5000" />
    <atom:link rel="stadium" href="http://localhost:8080/soccer/api/teams/5000/stadium" />
    <atom:link rel="players" href="http://localhost:8080/soccer/api/teams/5000/players" />
    <teamid>5000</teamid>
    <name>Real Madrid C.F.</name>
    <foundationyear>1902</foundationyear>
    <rankingposition>1</rankingposition>
</team>
```

Y, evidentemente, accediendo a alguno de los recursos marcados por los enlaces, obtendríamos respuestas válidas:

```
Request URL: http://localhost:8080/soccer/api/teams/5000/players
Request Method: GET
Status Code: 200 OK
```

```
{
    "players": [{
        "links": [{
            "rel": "self",
            "href": "http://localhost:8080/soccer/api/teams/5000/players/7000"
        }, {
            "rel": "team",
            "href": "http://localhost:8080/soccer/api/teams/5000"
        }],
        "playerId": 7000,
        "name": "Cristiano Ronaldo",
        "goals": 172,
        "age": 28,
        "country": "Portugal",
        "teamId": 5000
    }, {
        "links": [{
            "rel": "self",
            "href": "http://localhost:8080/soccer/api/teams/5000/players/7001"
        }, {
            "rel": "team",
            "href": "http://localhost:8080/soccer/api/teams/5000"
        }],
        "playerId": 7001,
        "name": "Xabi Alonso",
        "goals": 12,
        "age": 32,
        "country": "España",
        "teamId": 5000
    },
    // etc, etc, etc...
    ]
}
```

Podéis ver y descargar el CÓDIGO FUENTE COMPLETO DEL EJEMPLO AQUÍ.

6. Referencias.

- Roy Fielding: REST APIs must be hypertext-driven
- Spring HATEOAS: Quick start
- Código fuente del ejemplo
- HATEOAS: ¿qué y por qué? (excelente artículo en castellano)

7. Conclusiones.

En este tutorial hemos visto que, aunque REST destaca por su sencillez, es necesario que tengamos presentes ciertos principios de diseño a la hora de modelar nuestras APIs. HATEOAS es uno de los principios que debemos seguir.

Gracias al principio HATEOAS facilitamos el descubrimiento de los recursos que componen nuestra API delegando en el servidor la manera en la que enlazaremos con ellos, una gran ventaja a la hora de mantener diferentes versiones de nuestra API y evitando problemas a nuestros clientes a la hora de solicitar recursos :-)

Espero que este tutorial os haya sido de ayuda. Un saludo.

Miguel Arlandy

marlandy@autentia.com

Twitter: @m_arlandy

A continuación puedes evaluarlo:

Regístrate para evaluarlo

Por favor, vota +1 o compártelo si te pareció interesante

Share |

Anímate y coméntanos lo que pienses sobre este **TUTORIAL:**

» Registrate y accede a esta y otras ventajas **«**

SOME RIGHTS RESERVED Esta obra está licenciada bajo licencia Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5

PUSH THIS Page Pushers Community Help?

---
no clicks

0 people brought clicks to this page

+ + + + + + + +

powered by karmacracy

Copyright 2003-2013 © All Rights Reserved | Texto legal y condiciones de uso | Banners | Powered by Autentia | Contacto

W3C XHTML 1.0 W3C CSS XML RSS XML ATOM