Principios de Diseño de APIs REST

por Enrique Amodeo

Principios de diseño de APIs REST

(desmitificando REST)

Enrique Amodeo

This book is for sale at http://leanpub.com/introduccion_apis_rest

This version was published on 2013-03-06

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Enrique Amodeo by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Acabo de comprar ”Principios de Diseño de APIs REST” El libro de #REST en español #esrest

The suggested hashtag for this book is #esrest.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#esrest

Índice general

Sobre la cubierta

La foto de la cubierta es de un famoso trampantojo en la ciudad de Quebec, más concretamente en el *Quartier Petit Champlain*.

El autor eligió esta foto como cubierta porque ilustra claramente un conjunto de agentes interoperando entre sí.

Novedades en esta versión

Versión actual: 1.0

V1.0

- Corregido error de sintaxis en "Buenas prácticas y patrones de diseño básicos"
- Añadidos códigos de estado `415` y `422`
- Algunas aclaraciones sobre concurrencia optimista en "Buenas prácticas y patrones de diseño básicos"
- Algunos detalles más de seguridad en "Buenas prácticas y patrones de diseño básicos"
- Actualizado capítulo "HAL", cambiado algunos detalles, en especial la notación "Curie".
- Añadida sección sobre el patrón "*envelope*" en el capítulo "Hypermedia"
- Añadido capítulo "Collection+JSON"
- Añadido capítulo "HTML"
- Añadido capítulo "ATOM"
- Actualizadas las referencias

V0.5

- Actualizadas referencias
- Actualizada la sección de conclusiones del capítulo sobre "Hypermedia"
- Añadida aclaración en el ejemplo del carrito de la compra en el capítulo sobre "HAL" relativo al formulario de añadir una línea de pedido.
- Arreglado el segundo párrafo de la sección sobre versionado de APIs en el capítulo "Buenas prácticas y patrones de diseño básicos".
- Ampliada explicación sobre XML Schema extensible en la sección sobre versionado de APIs en el capítulo "Buenas prácticas y patrones de diseño básicos".
- Añadido capítulo "SIREN"

V0.4

- Erratas
- En el capítulo sobre "Hypermedia": Añadida una última sección
- En el capítulo sobre "Hypermedia": Los tipos de enlace `addtobasket` y `basket/items` ahora se llaman `form` e `items`
- Añadido capítulo sobre "Hypertext Application Language (HAL)"

V0.3

- “Hypermedia”
- Ajustado el formato a un tipo de letra más grande
- Arregladas algunas erratas e inexactitudes en “REST en la práctica: HTTP” sobre tipos mime y el código de estado `405`.
- Añadida nota sobre HMAC en “Buenas prácticas y patrones de diseño básicos”
- Añadida aclaracieon sobre “Consultas predefinidas VS. Consultas genéricas” en el capítulo “APIs orientadas a datos: CRUD”
- Actualizada bibliografía

V0.2

- “Novedades en esta versión”
- “La web programable”
- “Buenas prácticas y patrones de diseño básicos”
- Pequeña corrección de estilo en “Érase una vez”
- Aclaraciones en “REST en la práctica: HTTP”
- Clarificación de los riesgos de `PUT` y `DELETE` en “APIs orientadas a datos: CRUD”

V0.1

- Versión inicial
- Cubierta
- Índice
- “Agradecimientos”
- “Érase una vez”
- “¿Qué es REST?”
- “REST en la práctica: HTTP”
- “APIs orientadas a datos: CRUD”

Agradecimientos

Este libro empezó como un pequeño y corto capítulo en otro libro. Los que me conocen ya se temían que ese pequeño capítulo no iba a ser tan pequeño. Debido a ello, y a problemas de calendario decidí publicar el libro por separado y a mi ritmo. Gracias a ese equipo de personas por poner en marcha esto y por sus sugerencias, en especial a **@ydarias**, **@estebanm**, y **@carlosble**.

Gracias a todos aquellos que me han ayudado a mover el libro por el mundo del "social media".

Me gustaría agradecer a **@pasku1** y a **@juergas** por sus esfuerzos como revisores de este libro (os debo una caña… bueno dos). Sé que siempre puedo recurrir a ellos cuando me apetece calentarle la cabeza a alguien con mi última idea.

Y como no podría ser de otra forma, un agradecimiento especial a mi mujer, **@mcberros**, por no permitir nunca que dejara este proyecto y por su apoyo incondicional.

Érase una vez...

Tras muchos años intentando crear servicios web basados en tecnologías RPC, tales como CORBA o SOAP, la industria del desarrollo de software se encontraba en un punto muerto. Cierto, se había conseguido el gran logro de que un servicio implementado en .NET consiguiera comunicarse con uno escrito en Java, o incluso con otro hecho a base de COBOL, sin embargo todo esto sabía a poco. Es normal que supiera a poco, se había invertido cantidades ingentes de dinero en distintas tecnologías, frameworks y herramientas, y las recompensas eran escasas. Lo peor es que además las compañías se encontraban encalladas en varios problemas.

Por un lado la mantenibilidad de la base de código resultante era bastante baja. Se necesitaban complejos IDEs para generar las inescrutables toneladas de código necesarias para interoperar. Los desarrolladores tenían pesadillas con la posibilidad de que se descubriera algún bug en la herramienta de turno, o de que algún parche en éstas destruyera la interoperabilidad. Y si se necesitaba alguna versión o capacidad más avanzada de SOAP, probablemente el IDE no lo soportara o tuviera que ser actualizado.

Por otro lado, para depurar cualquier problema de interoperabilidad, había que bajar al nivel de HTTP: ¿estarían las cabeceras apropiadas? ¿La serialización del documento SOAP es conforme a "Basic Profile"[1]? ¿No se suponía que SOAP nos desacoplaba totalmente del protocolo de transporte?

Finalmente también había descontento. Se había soñado con un mundo de servicios web interoperables de manera transparente, organizados en directorios UDDI, con transacciones distribuidas a través de internet, etc. Al final esto no se consiguió, sólo interoperaban servicios entre distintos departamentos de una misma empresa, o de forma más rara algún servicio llamaba a otro servicio de otra empresa, todo con mucho cuidado y en condiciones bastante frágiles.

Cuando la situación se hizo insostenible, y algunos gigantes de la informática como Amazon, Google o Twitter necesitaron interoperabilidad a escala global y barata, alguien descubrió el camino al futuro mirando hacia el pasado, y descubrió REST...

[1] http://www.ws-i.org/profiles/BasicProfile-1.0-2004-04-16.html

1 La web programable: APIs y más APIs

¿Por qué debería preocuparme si mi sistema tiene o no una API web? Existen al menos dos razones que son importantes: costes y negocios.

La mayor parte de los costes en la industria del software provienen del mantenimiento, así que conviene preguntarnos: ¿qué es más mantenible, un sistema monolítico con millones de líneas de código? ¿O un sistema modular, donde cada subsistema colabore con los demás mediante una API bien definida? Obviamente lo segundo. Siguiendo este hilo de pensamiento, dichos subsistemas deberían tener entre si el menor acoplamiento posible; cada uno debería poder ser parado, arrancado y mantenido por separado, e implementados en la tecnología más adecuada a su funcionalidad. A partir de esta observación surgió el concepto de servicios web y aparecieron los *Enterprise Service Bus* y demás productos de integración empresarial.

Y desde el punto de vista de los negocios, ¿qué es más eficiente, una empresa que intenta solucionar todos los problemas derivados de su negocio? ¿O una empresa que se dedica sólo a lo esencial de su negocio, a mejorar continuamente su producto, y contrata los detalles no esenciales a sus proveedores? El autor no es un empresario de éxito (de momento), pero parece que la tendencia va en en esta última dirección.

Pero en la era de la información, el que dos empresas quieran colaborar de forma eficaz implica que sus sistemas de información deben colaborar de igual modo, por lo que ambas empresas deben contar con APIs que permitan dicha colaboración.

En este punto no es suficiente una API que esté desarrollada especificamente para que las empresas A y B colaboren. No, es mucho más eficaz que tu empresa tenga una API que permitiera a múltiples *partners* colaborar contigo. No es deseable tener que realizar un nuevo desarrollo para cada uno de ellos. Si unimos esta necesidad de colaboración entre negocios, con la necesidad de integración entre subsistemas, cobra sentido el que ambos problemas se resuelvan con la misma solución: una API web. En un principio se usó tecnología SOAP para implementar estas APIs web, pero los resultados no fueron del todo satisfactorios. Aun así, por un tiempo esta tecnología sobrevivió a base de una gran campaña de comercialización y de que al menos funcionaba (aunque fuera de una forma un tanto precaria).

Sin embargo este no es el fin de la historia. Con el advenimiento de la *web 2.0*, las aplicaciones móviles, webs con capacidades AJAX, y la explosión de startups en el mundo web, las APIs web han dejado de ser una mera solución de conveniencia para resolver los problemas de integración y de *Business to Busisness*, y han pasado a convertirse en una necesidad. De hecho hoy en día la API web puede ser en si misma el producto que queremos comercializar. Actualmente es común que se pague por el uso de una API en el caso de que el consumo sea muy intenso. Es normal tener miles de millones de llamadas en las APIs más populares (Google, ebay, twitter, facebook, netflix, amazon, accuweather, klout,salesforce, linkedin). Así que las APIs web pueden ser una fuente de ingresos bastante respetable.

En vista de estas circunstancias las APIs web deben estar diseñadas de tal manera que sean sencillas de consumir, no sólo desde otros servidores, sino también desde otros clientes con menores capacidades de cálculo, tales como dispositivos móviles y páginas web. Todo esto está haciendo que el enfoque SOAP a la construcción de APIs web sea cada vez menos adecuado, y que el enfoque REST haya triunfado. Cuanto más sencilla de usar sea nuestra API, más usuarios tendrá y nos ofrecerá mayores oportunidades de negocio.

En el momento de escribir este libro existen unas siete mil APIs públicas, esto es una señal distintiva de que todo esto no es una teoría sino una realidad, de que estamos entrando en la era de la **web programable**.

2 ¿Qué es REST?

2.1 Definición

REST no es una tecnología, ni siquiera una arquitectura, REST es un estilo arquitectónico. Es un conjunto de restricciones a respetar cuando diseñamos la arquitectura de nuestros servicios web. Las restricciones propuestas por REST son las siguientes:

- **REST no es RPC, sino orientado a recursos**. Los servicios web no representan acciones, sino entidades de negocio. En vez de publicar verbos como por ejemplo "comprar", se publican nombres como "carrito de la compra" o "pedido". En este sentido podemos pensar en RPC como intentar definir la API de un sistema en base a procedimientos, es decir, es un paradigma procedural. Sin embargo REST define la API como un conjunto de recursos que representan objetos de negocio; este enfoque está mucho más cercano a la OO que a otra cosa. En lo sucesivo usaré las palabras "recurso", "entidad" y "servicio" de forma intercambiable, aunque realmente la más correcta es "recurso".
- **Cada recurso posee un identificador único universal** (UUID o GUID) con el cual podemos hacer referencia a él. Estas referencias las puede usar un cliente de nuestra API para acceder al recurso, o bien puede utilizarse para crear una relación desde un recurso a otro. Crear estas relaciones es tan sencillo como incluir una referencia de un recurso a otro usando el UUID del último.
- La implementación, y la forma exacta en la que un recurso se representa internamente, debe ser **privada y no accesible al exterior**.
- Cada recurso tiene una **interfaz**, o conjunto de operaciones que admite. Basta saber el UUID del recurso para poder enviarle la operación que queremos realizar.
- La **interfaz es homogénea para todos los recursos**. Esto quiere decir que **todos los recursos deben escoger las operaciones que soportan de entre un conjunto cerrado ßde acciones**. Este conjunto de operaciones permitidas es una característica específica de cada arquitectura REST y no puede cambiarse. Como consecuencia no podemos inventar nuevas operaciones, sino que tenemos que modelar esas nuevas operaciones como recursos. Por ejemplo, en un sistema REST no podemos añadir una operación "reservarHotel", sino que tendríamos que crear un nuevo recurso llamado "ReservaDeHotel". El conjunto de mínimo de operaciones que debe tener un sistema REST es "leer", "actualizar" y "crear".
- Las operaciones se realizan mediante la **transferencia del estado del recurso entre cliente y servidor**. El cliente puede pedir que una **copia** del estado de un recurso sea transferido desde el servidor al cliente (leer), modificarlo, y mandar la copia modificada al servidor usando alguna de las operaciones de modificación permitidas (actualizar o crear).
- Las operaciones son **stateless**. Es decir, el resultado de una operación es independiente de la conversación que hayan mantenido el cliente y el servidor anteriormente. Como consecuencia de esto toda la información necesaria para llevar a cabo la operación debe mandarse como parámetros de ésta.

- Los recursos son **multimedia**, es decir, el estado de un recurso puede ser **representado mediante distintos formatos**. Por formato se entiende el formato concreto usado en la serialización del estado cuando se manda por red. Algunos ejemplos de formatos son XML, JSON, imagen JPEG, imagen GIF, etc. Como vimos antes, el estado de un recurso puede ser copiado desde un servidor al cliente o viceversa, pero por otro lado el formato usado para representar internamente el recurso es privado y desconocido por el cliente. Para que esto funcione, el cliente debe especificar al servidor que formatos entiende y viceversa, y ponerse de acuerdo en el conjunto de formatos a usar que sea más conveniente para ambas partes.

El acrónimo REST responde a "REpresentational State Transfer". El estilo arquitectónico REST fue descrito por primera vez por Roy Thomas Fielding[1], allá por el año 2000 (sí, ha llovido mucho desde entonces).

Veamos lo que significa todo esto con un pequeño ejemplo de como sería la reserva de una habitación de hotel usando REST:

1. El cliente pide (leer) el recurso "Habitaciones" realizando una operación leer contra el servidor. En dicha operación se pasan parámetros correspondientes al rango de fechas de disponibilidad y características de la habitación.
2. El servidor responde con un documento representando un resumen de la lista de habitaciones que son conformes a los criterios de búsqueda. El resumen de cada habitación contiene el UUID de ésta, y el UUID de un recurso asociado que representa las reservas de dicha habitación.
3. El cliente opcionalmente puede pedir (leer) el detalle completo de cada habitación usando el UUID contenido en el resumen.
4. El cliente envía una operación de "crear" al recurso que representa las reservas de la habitación elegida. Recordemos que el UUID de este recurso estaba tanto en el resumen como en el detalle de cada habitación. Al enviar la operación pasa como parámetro los detalles de la reserva.
5. El servidor puede responder de varias formas:

    - Rechazando la petición.
    - Devolviendo el UUID del recurso que representa la reserva que acabamos de crear e indicando que la reserva está completa.
    - Devolviendo el UUID del recurso que representa la reserva que estamos creando, y además un documento donde se pide información adicional, tal vez detalles de forma de pago o nombre de los huéspedes. El cliente debería modificar el documento e invocar la operación "actualizar" contra el UUID. De ahí se volvería al paso 5.

Como se ve toda la dinámica se basa en una conversación entre el cliente y el servidor, donde el cliente pide una copia del estado del recurso al servidor, el servidor se lo devuelve en un formato concreto (documento), el cliente lo modifica y se lo vuelve a mandar al servidor. Este a su vez

[1] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

procesa la petición, normalmente modificando el estado del recurso, o creando uno nuevo. A partir de ahí el servidor devuelve un nuevo documento con el resultado de la operación, y quizás también el UUID del nuevo recurso creado, y la conversación continúa. Ahora se entenderá mejor el significado de "REpresentational State Transfer", ya que todas las operaciones se basan en transferir una representación del estado de un recurso entre el cliente y el servidor y viceversa.

2.2 ¿Por qué usar REST?

¿Queremos diseñar un sistema de servicios web? Pues lo más sensato es ver como funciona la web, e investigar el único caso de éxito de interoperabilidad a escala global que existe en la actualidad. ¿Cuál es tal caso de éxito? ¿Acaso existe? Sí, y es ni más ni menos que la **World Wide Web**. Sí querido lector, las páginas web que navegamos a diario son un caso real de interoperabilidad a escala global, sólo es necesario usar un navegador decente que entienda los formatos disponibles para cada página (HTML, GIF, JPEG, códecs de video, etc) para poder consumirlas. No importa que el fabricante del navegador y el autor de la página no hayan interactuado en la vida, o que los autores de las páginas sean totalmente independiente entre si.

El enfoque RPC basado en SOAP ha fracasado, no conozco ningún sistema basado en este paradigma que alcance niveles de interoperabilidad tan altos, a costes tan bajos, como la World Wide Web. Ésta es por lo tanto un ejemplo a seguir, una guía a las decisiones de diseño necesarias para hacer servicios interoperables.

Es trivial darse cuenta de que si quieres hacer servicios web, tienes que respetar los principios de diseño de la web y diseñar tus servicios con ellos en mente. ¿Cuáles son los principios arquitectónicos de la web? Los principios REST.

3 REST en la práctica: HTTP

3.1 Introducción

Vayamos a lo práctico, ¿cómo implementar REST en el mundo real? Idealmente necesitamos un protocolo que cumpla los principios REST, y que esté ampliamente distribuido. Ya existe tal protocolo, que no es otro que Hyper Text Transfer Protocol[1] o **HTTP**

HTTP es una pieza fundamental en la que se basa la World Wide Web, y especifica como intercambiar entre cliente y servidor recursos web. Es un protocolo idóneo para implementar servicios web, ya que además de ser ubicuo, su diseño sigue los principios REST. Veamos ahora como funciona este protocolo y como encaja en el estilo arquitectónico REST.

HTTP es un protocolo que se sitúa al nivel de aplicación, incluyendo algo del nivel de presentación, dentro de la pila OSI[2]. En principio está diseñado para ser ejecutado sobre TCP o sobre transporte seguro TLS/SSL, aunque no sería descabellado mandar mensajes HTTP sobre otro protocolo de transporte. Se basa en un paradigma sencillo petición/respuesta, sin ningún tipo de memoria ni sesión de usuario, es decir, es un protocolo *stateless*.

El mensaje de petición en HTTP consta de una primera línea de texto indicando la versión del protocolo, el verbo HTTP y la URI destino. El verbo HTTP indica la operación a realizar sobre el recurso web localizado en la URI destino. A continuación, y de forma opcional, vienen las cabeceras. Cada cabecera va en una línea de texto distinta, y consiste en el nombre de la cabecera seguido de su valor, ambos separados por "`:`". Opcionalmente la petición puede llevar un cuerpo, separado por una línea en blanco del encabezamiento del mensaje, y que contiene un documento. Dicho documento puede estar en cualquier formato (incluso binario), aunque normalmente suele ser texto plano, HTML, XML o JSON. El formato del documento se define en la cabecera `Content-Type`.

Los mensajes de respuesta HTTP siguen el mismo formato que los de petición, excepto en la primera línea, donde se indica el código de respuesta junto a una explicación textual de dicha respuesta. El código de respuesta indica si la petición tuvo éxito o no y por qué razón.

Un ejemplo de petición HTTP sería:

[1] http://tools.ietf.org/html/rfc2616

[2] http://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.200-199407-I!!PDF-E&type=items

```
POST /server/payment HTTP/1.1
Host: www.myserver.com
Content-Type: application/x-www-form-urlencoded
Accept: application/json
Accept-Encoding:gzip,deflate,sdch
Accept-Language:en-US,en;q=0.8
Cache-Control:max-age=0
Connection:keep-alive

orderId=34fry423&payment-method=visa&card-number=2345123423487648&sn=345
```

Se observa como en la primera línea se indica que estamos usando una petición tipo `POST` contra la URI `/server/payment`. A continuación vienen las cabeceras con información adicional, como el tipo mime del cuerpo de la petición (`Content-Type`) o el tipo mime aceptable en la respuesta (`Accept`). Finalmente, tras una línea en blanco de separación, aparece el cuerpo, convenientemente formateado en el tipo mime especificado (`application/x-www-form-urlencoded`).

A continuación se detalla una posible respuesta:

```
HTTP/1.1 201 Created
Content-Type: application/json;charset=utf-8
Location: https://www.myserver.com/services/payment/3432
Cache-Control:max-age=21600
Connection:close
Date:Mon, 23 Jul 2012 14:20:19 GMT
ETag:"2cc8-3e3073913b100"
Expires:Mon, 23 Jul 2012 20:20:19 GMT

{
  "id":"https://www.myserver.com/services/payment/3432",
  "status": "pending"
}
```

En el caso de la respuesta la primera línea sólo indica la versión del protocolo HTTP usado (`HTTP/1.1`) y el código de respuesta (`201`). Después aparecen las cabeceras, y tras una linea en blanco, el cuerpo de la respuesta.

3.2 URIs

Los *Uniform Resource Identifiers* o **URIs**[3] son los identificadores globales de recursos en la web, y actúan de manera efectiva como UUIDs REST. Actualmente existen dos tipos de URIs, las URLs y

[3] http://www.ietf.org/rfc/rfc3986.txt

las URNs. Las primeras identifican un recurso de red mediante una IP o un DNS, las segundas son simples UUIDs lógicos con un espacio de nombres asociados.

Las URIs tiene el siguiente formato:

```
<esquema>:<parte específica esquema>/<ruta><querystring><fragmento>
```

El esquema, en el caso de una URL, indica qué protocolo se debe usar para acceder al recurso. En el caso que nos ocupa podría ser `http`, o `https` si usamos transporte seguro. Si es una URN el esquema es `urn`. En nuestro caso nos interesan las URLs, e ignoraremos las URNs en el resto de la discusión.

Tras el esquema, viene la parte específica del esquema. En las URLs sería la dirección del servidor donde se encuentra el recurso. Esta dirección se puede especificar mediante una dirección IP o mediante un nombre de dominio, y se separa del esquema mediante "`//`". Opcionalmente se especifica el puerto a usar en la conexión concatenándolo con "`:`". Esto no es necesario si se va a usar el puerto estándar del protocolo especificado. El puerto estándar para `http` es el `80` y para `https` es `443`.

A continuación viene la ruta del recurso dentro del servidor. Tiene la forma de un conjunto de segmentos separados por "`/`". Otros caracteres pueden aparecer como separadores, tales como "`;`" o "`@`". Esto raramente se usa, con la interesante excepción del "`;`" para indicar el identificador de sesión en servidor.

De forma opcional tenemos la *query string*, que indica parámetros adicionales de la URI a ser procesados en el servidor por la aplicación. Su uso más común es el de especificar criterios de búsqueda o filtrado o bien añadir parámetros o tokens de control. La *query string* se separa de la ruta mediante el carácter "`?`" y consiste en una serie de pares clave/valor separados por "`&`". Dentro de cada par la clave y el valor se separan mediate "`=`".

Finalmente y también de forma opcional tenemos el fragmento, que se separa del resto de la URI mediante "`#`". Al contrario que la *query string* el fragmento no se procesa en servidor, sino que está pensado para ser procesado por la aplicación cliente. Por lo tanto no tiene impacto en el diseño de servicios web.

Algunos ejemplos de URIs:

```
http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5.2
https://www.myserver.com/rest/books?author=vega
ftp://user@ftp.server.com/dir/somefile.txt
urn:oasis:names:specification:docbook:dtd:xml:4.1.2
urn:uuid:a74fc10b-58cc-4379-f5c7-1e02b2a3d479
```

3.3 Los verbos HTTP

Una característica completamente alineada con REST del protocolo HTTP es el hecho de que tenga una interfaz uniforme para todos los recursos web. HTTP define un conjunto predefinido y cerrado

de acciones o métodos HTTP. Es importante tener en cuenta que la propia especificación define los conceptos de *seguridad* e *idempotencia*, y clasifica los métodos conforme a estos dos criterios.

Un método se considera *seguro* si no produce efectos secundarios. Por efecto secundario se entiende cualquier modificación del estado del servidor, o interacción de éste con cualquier otro sistema, que produzca efectos perceptibles por el usuario. Normalmente sólo los métodos que representan lectura se consideran seguros.

Un método es idempotente si la ejecución repetida de éste, con exactamente los mismos parámetros, tiene el mismo efecto que si sólo se hubiera ejecutado una vez. Esta propiedad nos permite reintentar con seguridad una petición una y otra vez, y tener la seguridad de que la operación no se va a duplicar.

De forma muy común, debido a cortes y congestiones de red, el cliente no recibe la confirmación de si una operación se ha realizado o no. Tal vez la petición ha llegado al servidor y se ha ejecutado, con lo que no deberíamos duplicar la operación, o tal vez simplemente se ha quedado por el camino y debemos reintentarla. Los métodos idempotentes nos permiten recuperarnos de esta circunstancia simplemente repitiendo la petición, ya que nos aseguran que no se duplicará la operación si estuviéramos en el primer caso.

Los *métodos* (o también llamados *verbos*) HTTP usados com más frecuencia son los siguientes:

Método	Seguro	Idempotente	Semántica
`GET`	Sí	Sí	Leer el estado del recurso
`HEAD`	Sí	Sí	Leer, pero sólo las cabeceras
`PUT`	No	Sí	Actualizar o crear
`DELETE`	No	Sí	Eliminar un recurso
`POST`	No	No	Cualquier acción genérica no idempotente
`OPTIONS`	Sí	Sí	Averiguar las opciones de comunicación disponibles de un recurso

En general el uso de cada método es bastante explicativo, y más adelante veremos cómo usarlos, así como algunas buenas prácticas. Sin embargo conviene aclarar algunos aspectos.

El primero es la diferencia entre `HEAD` y `GET`. Ambos leen el recurso, pero el segundo devuelve tanto los datos del recurso web, como las cabeceras HTTP, mientras que el primero sólo las cabeceras.

Por otro lado el método `POST` es bastante misterioso y objeto de frecuentes malentendidos. En general se usa para crear un nuevo recurso, modificar uno existente o para ejecutar una acción genérica que no sea idempotente como realizar una transacción monetaria. Como veis la semántica de `PUT` parece que se solapa con la de `POST` de alguna manera, y esto es fuente de numerosos malentendidos. Más adelante aclararemos este tema.

El método `OPTIONS` se usa para determinar las opciones de comunicación de un recurso, tales como

qué métodos HTTP podemos usar contra esa URI. El resultado de `OPTIONS` no es *cacheable*, ya que el conjunto de métodos que podemos usar puede cambiar con el estado en el que se encuentre el recurso. Junto con `HEAD` nos permiten descubrir automáticamente cómo podemos comunicarnos con el servidor.

Existen otros métodos que no suelen ser muy usados en servicios REST, como `CONNECT` o `TRACE`.

3.4 Los tipos MIME

Otro aspecto muy apropiado para el desarrollo de servicios REST con el protocolo HTTP es su soporte para negociar distintos formatos (representaciones) a usar en la transferencia del estado entre servidor y cliente (y viceversa).

En HTTP cada uno de estos formatos constituye lo que se llama un tipo MIME. Existe un directorio con tipos MIME estandarizados[4]. Cada tipo MIME tiene el formato de `<tipo>/<subtipo>`. Algunos ejemplos de tipos MIME estándares: `application/json`, `application/xml`, `application/atom+xml`, `application/javascript`, `text/html` y `audio/vorbis`.

A veces en el subtipo suele aparecer el caracter "+". Esto indica que el tipo MIME es una variante de otro tipo principal. Por ejemplo, `application/atom+xml`, indica que el tipo MIME es una variante del más genérico `application/xml` pero siguiendo el XML schema de atom. Otra convención común es que si el subtipo empieza por `x-` entonces el tipo MIME no es estándar, sino propietario.

¿Cómo se negocia el tipo MIME entre el cliente y el servidor? Es sencillo, en el mensaje de petición se incluye una cabecera `Accept`, con una lista de tipos MIME que el cliente entiende, el servidor selecciona el tipo que más le interese de entre todos los especificados en la cabecera `Accept`, y devuelve la respuesta. Si el servidor no entiende ninguno de los tipos MIME propuestos devuelve un mensaje con código `406`, indicando que es incapaz de aceptar la petición. Si no se incluye la cabecera `Accept` se indica que se acepta cualquier tipo MIME. Por otro lado, ya sea en la petición como en la respuesta HTTP, la cabecera `Content-Type` debe contener el tipo MIME del documento que aparece en el cuerpo del mensaje. Si el mensaje no tiene cuerpo no debe aparecer dicha cabecera.

En la cabecera `Accept`, la lista de tipos MIME se especifica mediante lo que se llama un *media range*. Un *media range* es una lista separada por comas de tipos MIME, donde cada uno de los tipos MIME puede ir acompañados por uno o más parámetros. Para indicar un parámetro en un tipo MIME se usa el ";". Así `text/html;level=3` indica el tipo MIME `text/html` con un parámetro `level` con valor `3`. En general el significado del parámetro es específico de cada tipo MIME. Otro ejemplo de parámetro podría ser `charset`, indicando la codificación de caracteres dentro de un formato de tipo texto o de tipo aplicación no binario. Ej. `text/plain; charset=ISO-8859-1` y `application/json; charset=UTF-8`

Además en un *media range* pueden aparecer expresiones de rango. Hay dos:

1. `*/*` indica cualquier tipo MIME

[4] http://www.iana.org/assignments/media-types/index.html

2. `<tipo>/*` indica cualquier subtipo dentro del tipo. Por ejemplo `image/*` indica cualquier imagen en cualquier formato.

Un *media range* no sólo especifica una lista de tipos MIME aceptables, sino tambien un orden de preferencia, de tal manera que el servidor debe intentar elegir el tipo MIME que soporte pero que tenga mayor preferencia para el cliente. La preferencia se calcula usando las siguientes reglas:

1. Si el tipo MIME tiene un parámetro llamado "`q`", usar el valor de ese parámetro como preferencia. Este valor debe estar entre 0 y 1, y el parámetro "`q`" debe ser el primero.
2. Si no existe parámetro "`q`", asumir que tiene un valor de 1.
3. Si hay empate en el valor de "`q`", el orden de preferencia para desempatar, de mayor a menor es el que sigue:

    - El tipo MIME con mayor número de parámetros
    - Las expresiones `<tipo>/*`
    - La expresión `*/*`

Pongamos un ejemplo, la cabecer Accept en la siguiente petición:

```
1 GET /rest/booking/1 HTTP/1.1
2 Host: www.myserver.com
3 Accept: text/*;q=0.3, text/html;q=0.7, text/html;level=1
4
```

Especifica el siguiente orden de preferencias de tipos MIME:

1. `text/html;level=1`
2. `text/html`
3. Cualquier formato de texto (`text/*`)

3.5 Códigos de estado

En HTTP el mensaje de respuesta contiene en su primera línea lo que se llama el código de estado, que indica el resultado de la operación. Los códigos de respuesta más usados son:

- `200`. Indica éxito de la operación de forma genérica. Se usa cuando no hay otro código de éxito más específico. El mensaje de respuesta debe contener un cuerpo, en otro caso se usa `204`.
- `201`. Indica que se creó con éxito un nuevo recurso web. Suele devolverse cuando ejecutamos un método POST o PUT. En la respuesta se devuelve la URI del nuevo recurso creado dentro de la cabecera `Location`. La respuesta puede contener un cuerpo con los datos del nuevo recurso.

- `202`. Indica que la petición se ha aceptado pero que no está completa. Se puede devolver en la respuesta un cuerpo con información sobre cuanto queda para que se complete la operación, o cómo monitorizar el progreso de ésta. Una práctica común es usar la cabecera `Location` para indicar la URI de un recurso, sobre el que se puede hacer polling para saber el estado de progreso de la petición. Es muy útil cuando el servidor esta congestionado o para modelar operaciones de larga duración.
- `204`. Indica éxito de la operación de forma genérica. Se usa cuando no hay otro código de éxito más específico. El mensaje de respuesta debe estar vacio y no tener cuerpo, en otro caso usar `200`.
- `301`. Indica que el recurso se ha movido a otra URI de forma permanente. La nueva URI se indica en la cabecera `Location` de la respuesta. Este mecanismo de redirección se puede usar para versionar servicios como veremos más adelante.
- `304`. Indica que el recurso no se ha modificado. Normalmente es la respuesta de una *cache* a una operación. La respuesta no contiene cuerpo.
- `400`. El mensaje de petición está mal formado.
- `401`. La petición no está autorizada, acceso denegado. Las credenciales de la petición son incorrectas (tal vez el usuario no existía o el esquema de autenticación no es correcto) o simplemente la petición no tenía credenciales. Indica la necesidad de volver a hacer la petición con unas credenciales correctas para autenticarse.
- `403`. Acceso denegado: las credenciales son insuficientes para acceder al recurso, el usuario no tiene permiso.
- `404`. Recurso no encontrado.
- `405`. Método no soportado, el verbo HTTP especificado en la petición no está implementada. Esta respuesta debe incluir una cabecera `Allow` indicando los métodos HTTP soportados. Por ejemplo, si se produjera una petición `POST` a un recurso que sólo soporta `GET`, `HEAD` y `PUT`, el servidor respondería con:

```
1 HTTP/1.1 405 Method Not Allowed
2 Allow: GET, HEAD, PUT
3
```

- `406`. La petición no es aceptable. Tal vez el servidor no soporta los tipos MIME aceptados por el cliente.
- `409`. Conflicto de versiones. Usado en la implementación de concurrencia optimista (ver más adelante).
- `412`. Fallo en la precondición. Usado en la implementación de concurrencia optimista (ver más adelante).
- `415`. El tipo mime del cuerpo de la petición no está soportado por el servidor.
- `422`. El cuerpo de la petición está mal formado. Por ejemplo un JSON o un XML con una sintaxis incorrecta (no cierran las llaves o las etiquetas), o tal vez el tipo mime especificado en la cabecera `Content-Type` no concuerda con el formato del cuerpo.

- `500`. Se produjo un error inesperado en el servidor. Se puede usar para modelar excepciones en la capa de aplicación.

El uso de muchos de estos códigos de estado es bastante evidente. Más adelante veremos algunos casos de uso concretos.

3.6 QoS en HTTP

Existen cabeceras en HTTP que tienen que ver con mejorar la eficiencia de la comunicación.

Por ejemplo, las cabeceras `Content-Encoding` y `Accept-Encoding` nos permiten modificar la codificación del cuerpo de los mensajes. Un caso práctico de esto es comprimir el cuerpo de los mensajes con el algoritmo GZIP para ahorrar ancho de banda.

La cabecera `Keep-Alive` nos permite reutilizar una misma conexión a nivel de transporte (un socket TCP por ejemplo) para varios ciclos de petición/respuesta. Sin esta cabecera cada petición/respuesta iría en una conexión diferente, lo que consumiría mayor cantidad de recursos y aumentaría la latencia.

La cabecera `Upgrade`, nos permite cambiar a otro protocolo. Ahora está siendo usada principalmente para cambiar a protocolo WebSocket.

Una cabecera muy usada es `Transfer-Encoding`, que nos permite enviar mensajes de longitud variable, lo que es indispensable para hacer streaming.

Afortunadamente como desarrolladores es improbable que tengamos que preocuparnos de todo esto, ya que se gestiona de forma casi transparente a través de la infraestructura. Sin embargo sí que hay aspectos de HTTP relacionados con la calidad de servicio, como la gestión de caché, la seguridad o la concurrencia optimista, que impactan en el diseño de servicios. Veremos más sobre este tema en el próximo capítulo.

3.7 HTTP y REST

Como vemos el protocolo HTTP está diseñado para cumplir los principios REST. Tenemos recursos (documentos web), que poseen UUIDs (URIs) y que pueden ser accesibles mediante múltiples representaciones (tipos MIME). Además se tiene un conjunto de operaciones predefinidas, los verbos HTTP, que se pueden utilizar siguiendo un patrón de petición/respuesta *stateless*. Como se ve HTTP tiene todos los ingredientes de una arquitectura REST.

Sin embargo existen formas de usar el protocolo HTTP que no son REST, aunque se le parecen. A estos diseños los llamamos servicios RESTlike, en contraposición a los servicios REST puros o RESTful.

Un ejemplo típico de servicio RESTlike se produce cuando las distintas acciones tienen URI propia, como por ejemplo en `http://www.server.com/reserva/123/completar`. Esto rompe con

el principio REST más simple, que es que se deben exponer sólo recursos y no operaciones. También rompe el principio REST que indica que sólo los recursos deben tener UUID y ser referenciables.

> Otro ejemplo típico de este tipo de APIs RESTlike sería especificar la acción en un parámetro de la *query string*. Por ejemplo: `http://www.server.com/reserva/123?action=completar`

Otro tipo de servicios RESTlike son aquellos en los que se usa siempre el mismo verbo HTTP para todas las acciones, normalmente POST. A esto se le llama **HTTP tunneling**, ya que se trata a HTTP como un protocolo de nivel de transporte, ignorando todas sus capacidades de nivel de aplicación, un verdadero desperdicio. En estos diseños se suele usar o bien la URI (como vimos antes), o bien algún parámetro en el cuerpo de la petición, para indicar que acción queremos. En el último caso realmente estamos rompiendo HTTP. Como vimos cada verbo tiene un significado distinto y toda la infraestructura de la web está preparado para ello. Si hacemos que todas las operaciones sean transportadas mediante el mismo verbo, los nodos intermedios de la web tratarán todas las peticiones de la misma manera, ya que no tienen manera de saber cual es la semántica de la petición.

> Un ejemplo especialmente dañino de esto es usar `POST` para todo, incluso para hacer lecturas.
>
> Las consecuencias de esto se pudieron apreciar hace algunos años donde casi cualquier framework web usaba `POST` para navegar, lo que causó que el botón "atrás" de los navegadores y las *caches* no pudieran ser usadas de forma efectiva.
>
> Otro ejemplo lo encontramos en SOAP, que tuvo problema con algunos *firewall* HTTP debido a que todas las peticiones eran `POST`. Para solucinarlo se crearon y vendieron numerosos "*firewalls* de aplicación", que necesitaban entender el cuerpo SOAP de la petición HTTP. Estos *firewalls* eran más costosos de configurar y menos escalables que los *firewalls* HTTP más tradicionales. De esto se pasó a ESBs y *appliances*, aun más caros…

Otra práctica que no es REST es tener distintas URIs para pedir el recurso en distintos formatos. Ejemplo: `http://www.server.com/reserva/22.html` y `http://www.server.com/reserva/22.pdf`. El tipo de formato se debe indicar en las cabeceras HTTP `Content-Type` y `Accept`. Según REST sólo los recursos pueden tener URI, y una misma URI se puede servir usando distintas representaciones.

Esta práctica puede llegar a ser aceptable si queremos especificar exactamente el tipo mime que queremos dentro de un enlace HTML, que no soporta ningún atributo para ello. Un caso de uso sería cuando necesitamos tener un enlace HTML para abrir un recurso web como PDF en otra ventana. El navegador no va a enviar por defecto en la cabecera `Accept` de la petición el tipo mime correspondiente a PDF. De ahí que necesitemos poner la extensión.

Úsese sólo en este tipo de casos, cuando no hay otra solución disponible.

El buen diseño de servicios web REST se basa en saber como mapear los distintos conceptos de REST al protocolo HTTP. La idea es aprovechar bien todas las capacidades de dicho protocolo para implementar REST sin destruir la semántica estándar de HTTP. De esta forma conseguiremos aprovechar de forma efectiva toda la infraestructura que ya está disponible en la web, como caches, proxies, firewalls y CDNs.

4 APIs orientadas a datos: CRUD

4.1 Introducción

El caso de uso más sencillo al diseñar servicios REST con HTTP se produce cuando dichos servicios publican operaciones CRUD sobre nuestra capa de acceso a datos. El acrónimo CRUD responde a "Create Read Update Delete" y se usa para referirse a operaciones e mantenimiento de datos, normalmente sobre tablas de un gestor relacional de base de datos. En este estilo de diseño existen dos tipos de recursos: entidades y colecciones.

Las colecciones actuan como listas o contenedores de entidades, y en el caso puramente CRUD se suelen corresponder con tablas de base de datos. Normalmente su URI se deriva del nombre de la entidad que contienen. Por ejemplo, `http://www.server.com/rest/libro` sería una buena URI para la colección de todos los libros dentro de un sistema. Para cada colección se suele usar el siguiente mapeo de métodos HTTP a operaciones:

Método HTTP	Operación
`GET`	Leer todas las entidades dentro de la colección
`PUT`	Actualización mútiple y/o masiva
`DELETE`	Borrar la colección y todas sus entidades
`POST`	Crear una nueva entidad dentro de la colección

Las entidades son ocurrencias o instancias concretas, que viven dentro de una colección. La URI de una entidad se suele modelar concatenado a la URI de la colección correspondiente un identificador de entidad. Este identificador sólo necesita ser único dentro de dicha colección. Ej. `http://www.server.com/rest/libro/ASV2-4fw-3` sería el libro cuyo identificador es `ASV2-4fw-3`. Normalmente se suele usar la siguiente convención a la hora de mapear métodos HTTP a operaciones cuando se trabaja con entidades.

Método HTTP	Operación
`GET`	Leer los datos de una entidad en concreto
`PUT`	Actualizar una entidad existente o crearla si no existe
`DELETE`	Borrar una entidad en concreto
`POST`	Añadir información a una entidad ya existente

A continuación, en las siguientes secciones, veremos más en detalle algunas opciones de diseño para cada operación CRUD.

4.2 Leyendo

La operación que parece más sencilla de modelar es la de lectura, aunque como veremos, el demonio está en los detalles.

Todas las operaciones de lectura y consulta deben hacerse con el método `GET`, ya que según la especificación HTTP, indica la operación de recuperar información del servidor.

Lectura de entidades

El caso más sencillo es el de leer la información de una entidad, que se realiza haciendo un `GET` contra la URI de la entidad. Esto no tiene mucho más misterio, salvo en el caso de que el volumen de datos de la entidad sea muy alto. En estos casos es común que queramos recuperar los datos de la entidad pero sólo para consultar una parte de la información y no toda, con lo que estamos descargando mucha información que no nos es útil.

Una posible solución es dejar sólo en esa entidad los datos de uso más común, y el resto dividirlo en varios recursos hijos. De esta manera cuando el cliente lea la entidad, sólo recibirá los datos de uso más común y un conjunto de enlaces a los recursos hijos, que contienen los diferentes detalles asociados a ésta. Cada recurso hijo puede ser a su vez o una entidad o una colección.

En general se suele seguir la convención de concatenar el nombre del detalle a la URI de la entidad padre para conseguir la URI de la entidad hija. Por ejemplo, dada una entidad `/rest/libro/23424-dsdff`, si se le realiza un `GET`, recibiríamos un documento, con el título, los autores, un resumen, valoración global, una lista de enlaces a los distintos capítulos, otra para los comentarios y valoraciones, etc.

Una opción de diseño es hacer que todos los libros tengan una colección de capítulos como recurso hijo. Para acceder al capítulo `3`, podríamos modelar los capítulos como una colección y tener la siguiente URL: `/rest/libro/23424-dsdff/capitulo/3`. Con este diseño tenemos a nuestra disposición una colección en `/rest/libro/23424-dsdff/capitulo`, con la cual podemos operar de forma estándar, para insertar, actualizar, borrar o consultar capítulos. Este diseño es bastante flexible y potente.

Otro diseño, más simple, sería no tener esa colección intermedia y hacer que cada capítulo fuera un recurso que colgara directamente del libro, con lo que la URI del capítulo `3` sería: `/rest/libro/23424-dsdff/capitulo3`. Este diseño es más simple y directo y no nos ofrece la flexibilidad del anterior.

¿Cuál es la mejor opción? Depende del caso de uso que tengamos para nuestra API.

Si no tenemos claro que operación vamos a soportar para las entidades hijas, o si sabemos que necesitamos añadir, borrar y consultar por diversos criterios, es mejor usar una colección intermedia.

> Si no necesitamos todo esto, es mejor hacer enlaces directos, ya que es un diseño más sencillo.

Volviendo al problema de tener una entidad con un gran volumen de datos, existe otra solución en la que no es necesario descomponerla en varios recursos. Se trata simplemente de hacer un `GET` a la URI de la entidad pero añadiendo una *query string*. Por ejemplo, si queremos ir al capítulo número 3, podemos hacer `GET` sobre `/rest/libro/23424-dsdff?capitulo=3`. De esta forma hacemos una lectura parcial de la entidad, donde el servidor devuelve la entidad libro, pero con sólo el campo relativo al capítulo 3. A esta técnica la llamo *slicing*. El usar *slicing* nos lleva a olvidarnos de esta separación tan fuerte entre entidad y colección, ya que un recurso sobre el que podemos hacer *slicing* es, en cierta medida, una entidad y una colección al mismo tiempo.

Como se aprecia REST es bastante flexible y nos ofrece diferentes alternativas de diseño, el usar una u otra depende sólo de lo que pensemos que será más interoperable en cada caso. Un criterio sencillo para decidir si hacer *slicing* o descomponer la entidad en recursos de detalle, es cuantos niveles de anidamiento vamos a tener. En el caso del libro, ¿se accederá a cada capítulo como un todo o por el contrario el cliente va a necesitar acceder a las secciones de cada capítulo de forma individual? En el primer caso el *slicing* parece un buen diseño, en el segundo no lo parece tanto. Si hacemos *slicing*, para acceder a la sección `4` del capítulo `3`, tendríamos que hacer: `/rest/libro/23424-dsdff?capitulo=3&seccion=4`. Este esquema de URI es menos semántico, y además nos crea el problema de que puede confundir al cliente y pensar que puede hacer cosas como esta: `/rest/libro/23424-dsdff?seccion=4` ¿Qué devolvemos? ¿Una lista con todas las secciones `4` de todos los capítulos? ¿Un `404` no encontrado? Sin embargo en el diseño orientado a subrecursos es claro, un `GET` sobre `/rest/libro/23424-dsdff/capitulo/3/seccion/4` nos devuelve la sección `4` del capítulo `3`, y sobre `/rest/libro/23424-dsdff/seccion/4` nos debería devolver `404` no encontrado, ya que un libro no tiene secciones por dentro, sino capítulos. Otra desventaja del *slicing* es que la URI no es limpia, y el posicionamiento en buscadores de nuestro recurso puede ser afectado negativamente por esto (sí, un recurso REST puede tener SEO, ya lo veremos más adelante).

A veces no tenemos claro cual va a ser el uso de nuestra API REST. En estos casos es mejor optar por el modelo más flexible de URIs, de forma que podamos evolucionar el sistema sin tener que romper el esquema de URIs, cosa que rompería a todos los clientes. En este caso el sistema más flexible es descomponer la entidad en recursos de detalle, usando colecciones intermedias si es necesario.

Recordad que se tome la decisión que se tome, esta no debe afectar al diseño interno del sistema. Por ejemplo, si decidimos no descomponer la entidad en recursos hijos, eso no significa que no pueda internamente descomponer una supuesta tabla de libros, en varias tablas siguiendo un esquema maestro detalle. Y viceversa, si decido descomper la entidad en varios subrecursos, podría decidir desnormalizar y tenerlo todo en una tabla, o quizás no usar tablas sino una base de datos documental. Estas decisiones de implementación interna, guiadas por el rendimiento y la mantenibilidad del sistema, deben ser invisibles al consumidor del servicio REST.

Ningún cambio motivado por razones técnicas, que no altere la funcionalidad ofrecida por nuestra API, debe provocar un cambio en nuestra API REST

Un ejemplo de esto sería un cambio en la base de datos debido a que vamos a normalir o denormalizar el esquema de base de datos.

Consultas sobre colecciones

La operación más común sobre una colección es la consulta. Si queremos obtener todos los miembros de una colección, simplemente hay que realizar un `GET` sobre la URI de la colección. En el ejemplo de los libros sería: `http://www.server.com/rest/libro`. Yo he puesto `libro` en singular, pero realmente es una colección. ¿Qué nos devolvería esta llamada? Realmente hay dos opciones: una lista con enlaces a todos los libros o una lista de libros, con todos sus datos.

La petición podría ser algo así:

```
GET /rest/libro HTTP/1.1
Host: www.server.com
Accept: application/json

```

Una respuesta, donde se devuelvan sólo enlaces:

```
HTTP/1.1 200 Ok
Content-Type: application/json;charset=utf-8

["http://www.server.com/rest/libro/45",
 "http://www.server.com/rest/libro/465",
 "http://www.server.com/rest/libro/4342"]
```

Si la respuesta incluye también los datos de las entidades hijas, tendríamos:

```
HTTP/1.1 200 Ok
Content-Type: application/json;charset=utf-8

[ {
    "id": "http://www.server.com/rest/libro/45",
    "author": "Rober Jones",
    "title": "Living in Spain",
    "genre": "biographic",
    "price": { "currency": "$", "amount": 33.2}
  },
  {
    "id": "http://www.server.com/rest/libro/465",
    "author": "Enrique Gómez",
    "title": "Desventuras de un informático en Paris",
    "genre": "scifi",
    "price": { "currency": "€", "amount": 10}
  },
  {
    "id": "http://www.server.com/rest/libro/4342",
    "author": "Jane Doe",
    "title": "Anonymous",
    "genre": "scifi",
    "price": { "currency": "$", "amount": 4}
  } ]
```

Observen como vienen todos los datos del libro, pero además viene un campo extra `id`, con la URI de cada libro.

¿Qué es mejor? ¿Traernos enlaces a los miembros de la colección, o descargarnos también todos los datos?

En el primer caso la respuesta ocupa menos espacio y ahorramos ancho de banda. En el segundo se usa mayor ancho de banda, pero evitamos tener que volver a llamar a las URIs cada vez que queramos traernos los datos de cada entidad, es decir ahorramos en llamadas de red y por lo tanto en latencia. Según las características de nuestra red tendremos que tomar la decisión. En una red móvil, donde hay una gran latencia, probablemente sea mejor el enfoque de descargar todos los datos.

Otro tema a considerar es el uso de la *cache*. Si nuestras entidades cambian poco, seguramente las peticiones para recuperar el detalle de cada una de ellas nos lo sirva una cache. Desde este punto de vista, en algunos casos, la idea de descargar

sólo los enlaces puede ser mejor.

En cualquier caso, la latencia domina la mayoría de redes modernas. por lo tanto lo mejor sería usar por defecto el segundo diseño, y cambiar al primero sólo si se demuestra que es mejor (y realmente lo necesitamos).

Lo normal en todo caso, no es traerse todos los miembros de una colección, sino sólo los que cumplan unos criterios de búsqueda. La forma más sencilla es definir los criterios de búsqueda en la *query string*.

Petición para buscar libros de ciencia ficción con un precio máximo de 20 euros:

```
GET /rest/libro?precio_max=20eur&genero=scifi HTTP/1.1
Host: www.server.com
Accept: application/json

```

Y la respuesta:

```
HTTP/1.1 200 Ok
Content-Type: application/json;charset=utf-8

[ {
    "id": "http://www.server.com/rest/libro/4342",
    "author": "Jane Doe",
    "title": "Anonymous",
    "genre": "scifi",
    "price": { "currency": "€", "amount": 5}
  },
  {
    "id": "http://www.server.com/rest/libro/465",
    "author": "Enrique Gómez",
    "title": "Desventuras de un informático en Paris",
    "genre": "scifi",
    "price": { "currency": "€", "amount": 10}
  }]
```

Nótese el detalle de que los resultados viene ordenados por precio. Normalmente el servidor debería ordenar los resultados de alguna manera en función de la consulta. Si quisiéramos que el cliente definiera en un orden diferente al que proporcionamos por defecto, deberíamos dar soporte a consultas como esta: `/rest/libro?precio_max=20&genero=scifi&ordenarPor=genero&ascendiente=false`

¿Y si queremos buscar una entidad por identificador...? Simplemente hay que hacer un `GET` sobre la URI de la entidad, por lo que consultas por "clave primaria" no tienen sentido dentro de una colección REST. Petición para un libro en particular:

```
GET /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json

```

Y la respuesta:

```
HTTP/1.1 200 Ok
Content-Type: application/json;charset=utf-8

{
  "id": "http://www.server.com/rest/libro/465",
  "author": "Enrique Gómez",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency": "€", "amount": 10}
}
```

Consultas paginadas

Es muy común que una consulta devuelva demasiados datos. Para evitarlo podemos usar paginación. La forma más directa es añadir parámetros de paginación a la *query string*. Por ejemplo, si estuviéramos paginando una consulta sobre libros que en su descripción o título contuvieran el texto "el novicio", podríamos tener la siguiente petición para acceder a la segunda página de resultados:

```
GET /rest/libro?q=el%20novicio&minprice=12&fromid=561f3&max=10 HTTP/1.1
Host: www.server.com
Accept: application/json

```

Nótese el parámetro `max`, que indica al servidor cuantos resultados queremos como máximo. La paginación en si la hacemos usando los parámetros `minprice` y `fromid`, que indican cuál es el último resultado que se ha recibido. En el ejemplo los resultados están ordenados ascendentemente por precio. De esta forma el servidor debe realizar la consulta de forma que excluya dicho resultado y devuelva los siguientes 10 libros a partir del último mostrado. Estamos jugando con la ordenación de los datos para conseguir la paginación.

Existen muchas formas de implementar paginación, no sólo la que presento como ejemplo. Sin embargo la aproximación que aqui se presenta puede necesitar cambios en función de como hagamos las consultas e implementemos la paginación exactamente.

Esto es un problema de interoperabilidad, ya que estamos acoplando la implementación del servidor con el cliente. Éste debe conocer detalles como que parámetros usar, o como informarlos. En el momento que decidiéramos cambiar alguno de estos detalles por motivos técnicos, estariamos rompiendo nuestra API.

Para solucionar este problema, existe otra variante para implementar la paginación y consiste en modelar directamente las páginas de resultados como recursos REST y autodescubrirlas mediante enlaces. En el capítulo dedicado a hypermedia se hablará sobre este enfoque.

Tal como se han diseñado las consultas anteriormente, el servidor tiene que estar preparado para interpretar correctamente los parámetros de la *query string*. La ventaja es que es muy simple. La desventaja es que el cliente tiene que entender que parámetros hay disponibles y su significado, con lo que es menos interoperable.

Consultas predefinidas o Vistas

Existe otra forma de diseñar consultas, que consiste en modelarlas directamente como recursos REST. De esta forma podemos tener consultas que son recursos hijos de la colección principal. Se puede entender este enfoque como crear consultas predefinidas, filtros o vistas sobre la colección principal.

Como ejemplo de este enfoque podríamos hacer `GET` sobre `/rest/libro/novedades` y `/rest/libro/scifi` para consultar las novedades y los libros de ciencia ficción respectivamente. Sobre estas colecciones hijas podemos añadir parámetros en la *query string* para restringirlas más o para hacer paginación. Alternativamente podemos tener colecciones hijas anidadas hasta el nivel que necesitemos.

Esta forma de modelar consultas nos da una API mucho más limpia, y nos permite mayor interoperabilidad. Nos permite simplificar drásticamente el número de parámetros y significado de éstos. Como mayor inconveniente está que es un enfoque menos flexible, ya que se necesita pensar por adelantado que consultas va a tener el sistema. Por lo tanto suele ser un diseño muy apropiado en aplicaciones de negocio donde normalmente sabemos las consultas que vamos a tener, pero no es muy apropiado en aplicaciones donde el usuario define sus propias consultas en tiempo de uso de la aplicación.

La tendencia de diseño es mezclar ambas opciones. Por un lado modelar explícitamente la consultas más comunes e importantes. Por otro lado permitir una consulta genérica, normalmente de texto libre, al estilo de Google o Yahoo. Por ejemplo: `/rest/libro?description=el%20novicio`

A menos que estes diseñando una API rest para una base de datos, es mejor usar consultas predefinidas. Las razones son varias:

- **Rendimiento**. Podemos implementar nuestro sistema para que optimice las consultas que están predefinidas. Se pueden usar técnicas como definir índices apropiados para ellas, o usar vistas materializadas. En una consulta genérica no tenemos manera de optimimizar, ya que no sabemos a priori como va a ser la consulta que se va a procesar y cuál es el criterio de búsqueda.
- **Interoperabilidad**. Una consulta predefinida es mucho más sencilla de usar que una genérica. Implica menos parámetros que una consulta genérica o incluso ninguno. Además el hecho de definir parámetros en una consulta genérica viola hasta cierto punto la encapsulación de nuestro sistema. Expone que campos de información están disponibles para realizar búsquedas y cuales no.

4.3 Actualizando

A la hora de actualizar los datos en el servidor podemos usar dos métodos, `PUT` y `POST`. Según HTTP, `PUT` tiene una semántica de UPSERT, es decir, actualizar el contenido de un recurso, y si éste no existe crear un nuevo recurso con dicha información en la URI especificada. `POST` por el contrario puede usarse para cualquier operación que no sea ni segura ni idempotente, normalmente para añadir un trozo de información a un recurso o bien crear un nuevo recurso.

Si queremos actualizar una entidad lo más sencillo es realizar `PUT` sobre la URI de la entidad, e incluir en el cuerpo de la petición HTTP los nuevos datos. Por ejemplo:

```
PUT /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/json

{
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency": "€", "amount": 50}
}
```

Y la respuesta es muy escueta:

```
HTTP/1.1 204 No Content

```

Esto tiene como consecuencia que el nuevo estado del recurso en el servidor es exactamente el mismo que el que mandamos en el cuerpo de la petición. La respuesta puede ser `204` o `200`, en función de si el servidor decide enviarnos como respuesta el nuevo estado del recurso. Generalmente sólo se usa `204` ya que se supone que los datos en el servidor han quedado exactamente igual en el servidor que en el cliente. Con la respuesta `204` se pueden incluir otras cabeceras HTTP con metainformación, tales como `ETag` o `Expires`. Sin embargo en algunos casos, en los que el recurso tenga propiedades de sólo lectura que deban ser recalculadas por el servidor, puede ser interesante devolver un `200` con el nuevo estado del recurso completo, incluyendo las propiedades de solo lectura.

Es decir, la semántica de `PUT` es una actualización donde reemplazamos por completo los datos del servidor con los que enviamos en la petición.

También podemos usar `PUT` para actualizar una colección ya existente. Veamos un ejemplo:

```
PUT /rest/libro HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/json

{
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency": "€", "amount": 50}
}
```

Y la respuesta:

```
HTTP/1.1 204 No Content

```

En este caso `PUT` ha sobreescrito los contenidos de la colección por completo, borrando los contenidos anteriores, e insertando los nuevos.

Cuidado, este tipo de uso de `PUT` puede ser peligroso ya que se puede sobreescribir toda la colección, con el consiguiente riesgo de perder información.

En los casos en los que queramos actualizar sólo algunos miembros de la colección y no otros podríamos usar una *query string* para delimitar que miembros van a ser actualizados.

```
PUT /rest/libro?genero=scifi HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/json

{
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency": "€", "amount": 50}
}
```

La *query string* define un subconjunto de recursos sobre la colección, a los cuales se les aplicará la operación `PUT`. De esta forma conseguimos una manera sencilla de hacer una actualización masiva. ¡ Pero esto haría que todos los libros de ciencia ficción tuvieran los mismos datos ! Realmente esto no es muy útil en este contexto. Pero sí lo es cuando estemos haciendo actualizaciones parciales, como veremos en otra sección.

En algunos casos la actualización no se puede llevar a cabo debido a que el estado del recurso lo impide, tal vez debido a alguna regla de negocio (por ejemplo, no se pueden devolver artículos pasados 3 meses desde la compra). En estos casos lo correcto es responder con un `409`.

```
HTTP/1.1 409 Conflict

```

4.4 Borrando

Para borrar una entidad o una colección, simplemente debemos hacer `DELETE` contra la URI del recurso.

```
DELETE /rest/libro/465 HTTP/1.1
Host: www.server.com

```

Y la respuesta:

```
HTTP/1.1 204 No Content

```

Normalmente basta con un `204`, pero en algunos casos puede ser útil un `200` para devolver algún tipo de información adicional.

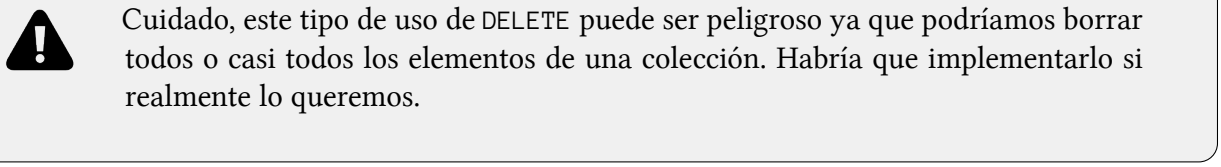
Hay que tener en cuenta que borrar una entidad, debe involucrar un borrado en cascada en todas las entidades hijas. De la misma forma, si borramos una colección se deben borrar todas las entidades que pertenezcan a ella.

Otro uso interesante es usar una *query string* para hacer un borrado selectivo. Por ejemplo:

```
DELETE /rest/libro?genero=scifi HTTP/1.1
Host: www.server.com

```

Borraría todos los libros de ciencia ficción. Mediante este método podemos borrar sólo los miembros de la colección que cumplen la *query string*.



4.5 Creando

Una forma de crear nuevos recursos es mediante `PUT`. Simplemente hacemos `PUT` a una URI que no existe, con los datos iniciales del recurso y el servidor creará dicho recurso en la URI especificada. Por ejemplo, para crear un nuevo libro:

```
PUT /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/json

{
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency": "€", "amount": 50}
}
```

Nótese que la petición es indistinguible de una actualización. El hecho de que se produzca una actualización o se cree un nuevo recurso depende únicamente de si dicho recurso, identificado por la URL, existe ya o no en el servidor. La respuesta:

```
HTTP/1.1 201 Created
Location: http://www.server.com/rest/libro/465
Content-Type: application/json;charset=utf-8

{
  "id": "http://www.server.com/rest/libro/465",
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency": "€", "amount": 50}
}
```

Nótese que el código de respuesta no es ni `200` ni `204`, sino `201`, indicando que el recurso se creó con éxito. Opcionalmente, como en el caso del ejemplo, se suele devolver el contenido completo del recurso recien creado. Es importante fijarse en la cabecera `Location` que indica, en este caso de forma redundante, la URL donde se ha creado el nuevo recurso.

Otro método para crear nuevos recursos usando `POST`. En este caso hacemos `POST` no sobre la URI del nuevo recurso, sino sobre la URI del recurso padre.

```
POST /rest/libro HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/json

{
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency": "€", "amount": 50}
}
```

Y la respuesta:

```
HTTP/1.1 201 Created
Location: http://www.server.com/rest/libro/3d7ef
Content-Type: application/json;charset=utf-8

{
  "id": "http://www.server.com/rest/libro/3d7ef",
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency": "€", "amount": 50}
}
```

En este caso la cabecera `Location` no es superflua, ya que es el servidor quien decide la URL del nuevo recurso, no el cliente como en el caso de `PUT`. Además cuando creamos un nuevo recurso con `POST`, éste siempre queda subordinado al recurso padre. Esto no tendría porque ser así con `PUT`.

`POST` tiene como ventaja que la lógica de creación URIs no está en el cliente, sino bajo el control del servidor. Esto hace a nuestros servicios REST más interoperables, ya que el servidor y el cliente no se tienen que poner de acuerdo ni en que URIs son válidas y cuáles no, ni en el algoritmo de generación de URIs. Como gran desventaja, `POST` no es idempotente.

La gran ventaja de usar `PUT` es que sí es idempotente. Esto hace que `PUT` sea muy útil para poder recuperarnos de problemas de conectividad. Si el cliente tiene dudas sobre si su petición de creación se realizó o no, sólo tiene que repetirla. Sin embargo esto no es posible con `POST`, ya que duplicaríamos el recurso en el

caso de que el servidor sí atendió a nuestra petición y nosotros no lo supiéramos.

4.6 Seguramente CRUD no sea lo mejor para tu API...

Hasta el momento hemos estado diseñando la API REST de una forma muy similar a como se diseñaría una BBDD. Algunos estarían tentados de ver las colecciones como "tablas" y las entidades como "filas", y pasar por alto el verdadero significado de lo que es un recurso REST. Este diseño ciertamente puede ser útil en casos sencillos, pero si queremos exprimir al máximo las capacidades de interoperabilidad del enfoque REST debemos ir más allá de esta forma de pensar. Más adelante veremos otras técnicas de diseño que maximizan la interoperabilidad.

Por otra parte, como se ha visto antes, no es bueno acoplar nuestro diseño de API REST a la implementación del sistema. En este sentido hay que tener cuidado con los frameworks. Por ejemplo, no es deseable el diseño de tu sistema REST se acople a tu diseño de tablas. En general el diseño de la API REST debe estar totalmente desacoplado de la implementación, y dejar que esta última pueda cambiar sin necesidad de alterar tu capa de servicios REST.

Sin embargo, en algunos escenarios sencillos, el enfoque CRUD es perfectamente válido. Por ejemplo, si simplemente queremos dotar de un API REST a una base de datos, o a nuestra capa de acceso a datos, el enfoque CRUD es perfectamente adecuado. En cualquier caso, incluso en estos escenarios, la API REST no debería exponer detalles de implementación, tales como el esquema de base de datos subyacente o las claves primarias. De este modo, si por ejemplo, decidimos desnormalizar nuestro esquema, nuestra API REST no debería tener que ser cambiada forzosamente (otra cosa es cambiar la implementación de esta).

Pero en el caso general, cuando definimos una API REST, lo que queremos exponer no es nuestra capa de acceso a datos, sino nuestra capa de lógica de aplicación. Esto implica investigar que casos de uso tenemos, cómo cambia el estado de nuestro sistema en función de las operaciones de negocio, y que información es realmente pública y cual no. Para diseñar nuestra API de forma óptima debemos ir más allá del paradigma CRUD, y empezar a pensar en casos de uso. Más adelante, en otro capítulo de este mismo libro, se explicará un enfoque mejor para este tipo de APIs: el enfoque de hypermedia o HATEOAS.

Pero antes necesitamos conocer un poco más las posibilidades que nos brinda HTTP y REST. En el siguiente capítulo veremos algunas técnicas más avanzadas que pueden ser usadas tanto en APIs orientadas a datos como en APIs basadas en hypermedia.

5 Buenas prácticas y patrones de diseño básicos

5.1 Respeta la semántica de HTTP

Lo más importante a la hora de diseñar una API REST es respetar la semántica de HTTP. Toda la infraestructura de internet está preparada para respetar dicha semántica, y la interoperabilidad de la web se basa en ella.

Si nuestra API ignora este aspecto, no podrá interoperar con caches, navegadores, CDNs, robots (*crawlers*), y en general, con cualquier otro sistema que esté operando en la web.

Aunque esto parece obvio, existen muchos sistemas que no respetaban este principio. Durante mucho tiempo la mayoría de las aplicaciones web han usado `POST` para leer y navegar con los consiguientes problemas (*back button* del navegador roto, problemas de cache, y efectos inesperados).

El otro día el autor escuchó un caso de desastre causado por no respetar la semántica de los verbos HTTP.

Al parecer la aplicación basecamp (`http://basecamp.com`), dentro de su funcionalidad de lista de tareas, implementaba la opción de borrar una tarea mediante un link HTML. Los links HTML, cuando se activan, siempre producen una petición `GET` al servidor, por lo tanto son considerados una forma de leer y navegar contenido, justo la semántica contraria del borrado.

Ocurrió que algunos usuarios empezaron a instalar en su navegador Google Web Accelerator (actualmente discontinuado). Este plugin exploraba la página web en la que se encuentra el usuario actualmente, y comienza a descargar en segundo plano los contenidos enlazados desde esa página. De esta forma cuando el usuario navege en un link la página ya estaría disponible.

Ya os podeis imaginar lo que ocurrión cuando dicho plugin empezó a descargarse los contenidos enlazados por los links “borrar tarea”…

En general debemos usar el verbo HTTP cuya semántica se aproxime más la operación que estamos modelando. Si queremos controlar el comportamiento de la cache, usemos las cabeceras de control de cache, en vez de renombrar las URIs con prefijos de versión, y así sucesivamente.

5.2 Servicios multimedia

Una ventaja de los servicios REST es que permite negociar el formato exacto en el que se va a intercambiar la información.

Como ya vimos, cada formato viene definido por un tipo MIME y podemos usar las cabeceras `Accept` y `Content-Type` para negociar qué tipo MIME se va a usar en la comunicación. Si no se puede llegar a un acuerdo entre el cliente y el servidor sobre el tipo MIME a usar, el servidor debe responder con un código `406` o `415`. Usará el código `406` cuando no pueda servir la respuesta en el tipo MIME especificado en la cabecera `Accept`. Usará `415` si no entiende el tipo mime del cuerpo de la petición, qie se encuentra especificado en la cabecera `Content-Type`.

Esto nos abre un nuevo campo con el que hacer nuestros servicios más potentes. Usando la misma URI y los mismos métodos HTTP, podemos consumir el recurso en distintos formatos, no es necesario tener una URI diferente para cada formato.

Por ejemplo podríamos consumir el recurso `http://www.server.com/rest/libro/3d-5FG-67` como JSON para poder mostrar información del libro dentro de una aplicación web rica. Al mismo tiempo podemos usar el formato PDF para descargar una copia del libro cuando alguien lo compre. También podríamos consumirlo como audio y descargar la versión audiolibro del mismo. Otra opción sería pedir el recurso en formato HTML, y entonces recibo una página web con la que ver los detalles del libro y poder comprarlo (o poder leerlo *online* de forma gratuita, según el caso de negocio). El límite realmente está en definir lo que tiene sentido desde un punto de vista de negocio.

Otro ejemplo sería un recurso colección con todos los movimientos bancarios de una cuenta. Lo puedo consumir como HTML y acceder a una página web donde visualizar y operar. ¿Y por qué no en forma Excel para descargarlos como una hoja de cálculo? ¿Y que tal en PDF para poder imprimir los recibos de todos los movimientos? ¿O quizás es mejor pedirlo en formato JPG o GIF para ver una bonita gráfica de gastos/ingresos?

En general, para maximizar la interoperabilidad, es mejor usar tipos MIME estandarizados, al fin y al cabo, de esta forma no tendremos que crear clientes específicos para consumir esos tipos MIME.

De forma opcional, se suelen definir tipos MIME específicos y propietarios para nuestros servicios REST. En estos casos, cuando el tipo MIME no es estándar, el nombre del tipo empieza por "`x-`". Estos tipos propietarios suelen estar muy optimizados para los casos de uso concreto de nuestro sistema. Por ejemplo, si quiero definir un formato especial para los datos de un libro, podría publicar el recurso `libro` de forma que soportara el tipo `application/x-libro`. De esta forma los clientes que soporten este tipo especial podrían aprovechar las características optimizadas de este tipo MIME.

En cualquier caso, crear tipos MIME propietarios de nuestra aplicación debe verse como una práctica a valorar con cuidado. Usar un tipo MIME propietario implica menor interoperabilidad ya que los posibles consumidores del recurso no lo entenderán a priori, y necesitarán hacer un desarrollo a medida para consumir dicho formato. Por lo tanto, en el caso de definir tipos MIME propietarios, se recomienda publicar además los recursos REST usando al menos un tipo MIME estándar en la medida de lo posible.

5.3 Concurrencia optimista

El problema

En sistemas distribuidos siempre debemos tener en cuenta los efectos de la concurrencia en la consistencia del estado del sistema. En el caso concreto de los servicios web, puede ocurrir que dos clientes lean el estado del mismo recurso y ambos manden modificaciones al servidor. Obviamente una modificación llegará al servidor antes que la otra. Esto causa que la modificación que llegue más tarde se base en datos obsoletos, lo que dependiendo del caso puede ser indeseable.

En una aplicación tradicional aplicaríamos una solución basada en el bloqueo de datos (concurrencia pesimista), pero en sistemas distribuidos de alta escalabilidad, como es el caso de los servicios web, la concurrencia pesimista no es apropiada. Si varios clientes quieren acceder a la misma información, un esquema de concurrencia pesimista bloquearía el acceso en escritura a todos los clientes excepto a uno. Este bloqueo se mantendría hasta que el cliente que lo posee deje de usar el recurso. Esta filosofía no se puede aplicar a un sistema web, ya que el sistema se volvería inusable. En la web es casi imposible imponer un límite máximo al número de clientes, tampoco sabemos cuanto tiempo puede tardar un cliente en dejar de usar el recuso (¿minutos, horas?). Imaginad una larga cola con clientes esperando en adquirir un bloqueo sobre un recurso web para poder realizar una operación.

Es mucho mejor un esquema de concurrencia optimista[1] donde no hay bloqueo de datos, y en caso de conflicto la petición de escritura es rechazada, y el cliente es notificado de tal hecho de forma ordenada. De esta forma los recursos siempre estan disponibles para cualquier operación todo el tiempo y nunca se bloquean. En el peor de los casos el cliente tendrá que reintentar una o dos veces, pero podrá realizar la operación sin esperar a otros clientes, de los que no se puede asegurar que terminen su operación en un tiempo razonable. Al fin y al cabo nuestra API funciona en la web, un mundo bastante salvaje e impredecible, y no en un entorno controlado como podría ser una intranet corporativa.

ETag y peticiones condicionales

En servicios REST podemos implementar la concurrencia optimista mediante el uso de la cabecera HTTP `ETag`. Esta cabecera permite al servidor indicar en la respuesta una *hash* o *fingerprint* del estado del recurso. La idea es que la `ETag` cambie si y sólo si el estado del recurso cambia. Hay dos formas sencillas de implementar una `ETag` en el servidor: mediante una hash resistente a colisiones y mediante un identificador de versión.

Existen dos tipos de `ETag`, las fuertes y las débiles. Si el estado del recurso cambia en al menos un bit, una `ETag` fuerte también cambia de valor. Las `ETag` fuertes nos permiten hacer una hash o fingerprint a nivel binario. Las `ETag` fuertes también tienen en cuenta los valores de las cabeceras HTTP y el formato concreto del documento. Por lo tanto si añadiésemos un espacio a un documento JSON, el valor de una `ETag` fuerte cambiaría, aunque la semántica del documento JSON no haya

[1] Concurrencia optimista: http://en.wikipedia.org/wiki/Optimistic_concurrency_control y también http://www.w3.org/1999/04/Editing

cambiado. Para solucionar esto existen las `ETag` débiles. Las `ETag` débiles no deberían cambiar con la mínima alteración del contenido de un mensaje HTTP, sino sólo si la semántica del estado del recurso ha cambiado. Sintácticamente podemos distinguir a una `ETag` fuerte de una débil, ya que la débil debe tener concatenado "W/" como prefijo. Por ejemplo, una `ETag` fuerte se representaría como `"sdl1kfj3FA"`, y una débil como `W/"sdl1kfj3FA"`.

Para hacer concurrencia optimista nos basta con `ETag` débiles. Como ya se dijo, podemos usar en el servidor una hash resistente a colisiones o un número de versión, que se actualizan cada vez que se modifiquen o cree un recurso. Es importante que el cálculo de la `ETag` sea consistente entre todas las máquinas servidoras en las que están distribuidos nuestros servicios REST. Es decir, la `ETag` debería ser la misma independientemente de que servidor responda a la petición. En el cálculo de una `ETag` débil sólo debe tenerse en cuenta la información que realmente es significativa para la semántica del recurso. Por ejemplo, los campos calculados, o aspectos de formato, no deberían entrar en el cálculo de la `ETag`. El valor de la `ETag` así generado podría persistirse junto con el estado del recurso, para futura referencia.

¿Y qué tiene todo esto que ver con la concurrencia optimista? Es simple, podemos usar la `ETag` para detectar los conflictos, y hacer peticiones HTTP condicionales.

Supongamos que hemos recibido lo siguiente del servidor al hacer una petición `GET`:

```
GET /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json

```

Y la respuesta:

```
HTTP/1.1 200 Ok
Content-Type: application/json;charset=utf-8
ETag: W/"686897696a7c876b7e"

{
  "id": "http://www.server.com/rest/libro/465",
  "author": "Enrique Gómez",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency":"€", "amount":10}
}
```

El cliente inspeccionando la `ETag` puede saber la "versión" de los datos y usarlo posteriormente en peticiones condicionales HTTP. Supongamos que el cliente desea realizar una actualización de la entidad recibida, y por lo tanto envía un `PUT`.

```
PUT /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
If-Match: W/"686897696a7c876b7e"
Content-Type: application/json

{
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency":"€", "amount":50}
}
```

En este caso, hace el `PUT` condicional mediante la cabecera `If-Match`. En esta cabecera se incluye el valor del `ETag` que posee el cliente. El servidor antes de realizar ninguna operación compara el `ETag` que tiene del recurso con el `ETag` enviado por el cliente en la cabecera `If-Match`. Si ambas coinciden, realiza la operación, como es el caso del ejemplo anterior. Si los `ETag` no coincidieran, el servidor no realizaría la petición, e informaría al cliente.

El mensaje de respuesta, en caso de éxito sería:

```
HTTP/1.1 204 No Content
ETag: W/"9082aff9627ab7cb60"

```

Obsérvese que se devuelve la nueva `ETag` del recurso modificado. Sin embargo podría haber ocurrido que el recurso hubiera sido modificado antes de que se realice nuestra petición de modificación. Estaríamos ante un conflicto de versiones. En este caso la respuesta sería diferente:

```
HTTP/1.1 412 Precondition Failed

```

La respuesta `412` puede ir acompañada de más cabeceras HTTTP.

El uso de `If-Match` nos permite asegurarnos que la petición se va a procesar sólo si los datos no han cambiado desde que fueron recuperados por el servidor. Si algún otro cliente hubiera cambiado el estado del recurso antes de que llegara nuestra petición de modificación, las `ETag` no coincidirían y el servidor no ejecutaría la acción. Es importante que el cliente sea notificado de tal circunstancia, ya que de esta forma le damos la posibilidad de tomar acciones para reparar el problema. La acción más sencilla es volver a pedir los datos frescos al servidor y notificar al usuario, que seguramente vuelva a reintentar su acción. Otra opción más compleja es pedir los datos frescos y permitir al usuario fusionar sus cambios con los nuevos datos. En función del escenario, la fusión de los cambios y/o el reintento de la petición se podría llevar a cabo automáticamente.

¿Por qué se le da prioridad a la primera modificación sobre la segunda? Un lector le comentó al autor que él solía hacer concurrencia optimista, pero haciendo que la última petición sobreescribiera a la primera. El autor considera que es mucho mejor implementar la concurrencia optimista tal y como se ha descrito en este capítulo: la primera modificación gana.

La justificación se centra en que el usuario o el sistema que realiza la petición lo hace en función de la versión del estado del recurso que posee, y en función de dicho estado decide realizar una modificación u otra. Si otro agente ha modificado mientras tanto el estado del recurso, la modificación que pide el cliente rezagado se basa en un estado que ya está obsoleto, y por lo tanto debe ser rechazada. De esta forma se le da la oportunidad al cliente de tomar una nueva decisión basada en el estado más actual del recurso.

Este enfoque permite una mayor facilidad a la hora de implementar un proceso de negocio, ya que nos asegura que ningún cliente va a poder ejecutar una operación de negocio si el estado del recurso que percibe ha quedado obsoleto.

5.4 Cache

Motivación

La cache es uno de los mecanismos más importantes que tiene la web para asegurar la alta escalabilidad. Permite que el servidor no tenga que procesar todas las peticiones, aumentando la escalabilidad de este. También permite a agentes intermedios, más cercanos (incluso en la mísma máquina) al cliente, responder en vez del servidor. De esta manera la latencia de las peticiones baja considerablemente.

Es importante saber cuándo un cliente, nodo intermedio, o un CDN, puede cachear la respuesta de una petición `GET` y cuando no. Al cachear la respuesta podemos evitar llamadas al servidor, por lo que aumentamos la escalabilidad de éste. Esto es clave si queremos diseñar APIs REST que tengan una alta escalabilidad. Sin embargo debemos ser capaces de especificar cuándo hay que descartar el contenido cacheado de un recurso, de otra manera el cliente no se enteraría de los cambios.

Control por versiones: If-None-Match y ETag

Una forma de controlar la cache es mediante `ETag`. La cache puede hacer un `GET` condicional al servidor mediante la cabecera `If-None-Match`. De esta forma el servidor puede responder con la nueva información (y la nueva ETag) en el caso de que ésta haya cambiado.

```
GET /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
If-None-Match: W/"686897696a7c876b7e"
Content-Type: application/json

```

El recurso sólo será devuelto por el servidor en el caso de que se haya producido algún cambio y el valor de la `ETag` haya cambiado. Por ejemplo:

```
HTTP/1.1 200 Ok
Content-Type: application/json;charset=utf-8
ETag: W/"9082aff9627ab7cb60"
Last-Modified: Wed, 01 Sep 2012 13:24:52 GMT
Date: Tue, 27 Dec 2012 05:25:19 GMT
Expires: Tue, 27 Dec 2012 11:25:19 GMT
Cache-Control: max-age=21600

{
  "id": "http://www.server.com/rest/libro/465",
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency":"€", "amount":50}
}
```

En el caso de que el recurso no haya cambiado el servidor respondería lo siguiente.

```
HTTP/1.1 304 Not Modified
Date: Tue, 27 Dec 2012 05:25:19 GMT
Expires: Tue, 27 Dec 2012 11:25:19 GMT
ETag: W/"686897696a7c876b7e"
Cache-Control: max-age=21600

```

El código `304` indica que el recurso que se pide no ha cambiado y realmente el contenido es el mismo. Con la respuesta se pueden enviar más cabeceras HTTP. De esta manera la cache se asegura que el recurso no ha cambiado, y ahorramos ancho de banda y procesamiento en servidor.

Control por tiempo: If-Modified-Since y Last-Modified

Otro método para gestionar la caché es usar las cabeceras `If-Modified-Since` y `Last-Modified`. En este caso no se mira el valor de `ETag`, sino sólo la última fecha de modificación del recurso que tenga el servidor (aunque la nueva versión del recurso se semánticamente equivalente a la que tiene el cliente).

El razonamiento es el siguiente: si se sabe que hay una muy baja probabilidad de que un recurso sea modificado en un espacio de tiempo de cinco minutos, no tiene mucho sentido pedir al servidor una copia cada cinco minutos o menos. De esta manera si configuraramos la cache para que sirva dicho recurso si no han pasado más de cinco minutos desde que se recibió una copia de él, ganaremos bastante rendimiento sin afectar mucho a la consistencia de los datos.

La petición tendría la siguiente forma:

```
GET /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
If-Modified-Since: Wed, 01 Sep 2012 13:24:52 GMT
Content-Type: application/json

```

En el caso que se hubiera producido una modificación posterior a la fecha especificada la respuesta sería:

```
HTTP/1.1 200 Ok
Content-Type: application/json;charset=utf-8
Date: Tue, 27 Dec 2012 05:25:19 GMT
Expires: Tue, 27 Dec 2012 11:25:19 GMT
Cache-Control: max-age=21600
Last-Modified: Tue, 27 Dec 2012 03:25:19 GMT

{
  "id": "http://www.server.com/rest/libro/465",
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency":"€", "amount":50}
}
```

Si por el contrario los datos siguieran siendo los mismos:

```
HTTP/1.1 304 Not Modified
Date: Tue, 27 Dec 2012 05:25:19 GMT
Expires: Tue, 27 Dec 2012 11:25:19 GMT
ETag: W/"686897696a7c876b7e"
Cache-Control: max-age=21600

```

Observese que se devuelve el nuevo valor de `Last-Modified`.

Es importante tener en cuenta que la interpretación de la fecha se hace con respecto al reloj del servidor, pudiendo producirse problemas si el cliente y el servidor tienen los relojes muy desincronizados.

Es un esquema totalmente equivalente al de `ETag` e `If-None-Match`, pero usando la fecha de modificación en vez de la `ETag`. En ambos casos sacamos partido de la capacidad de hacer un `GET` condicional usando las cabeceras `If-*`.

Control por versiones VS. Control por tiempo

Algunos os preguntareis por qué no usar `Last-Modified` en vez de `ETag`. Al fin y al cabo parece más simple de calcular en servidor. La razón es que algunas caches, notoriamente la cache de IE, no soportan correctamente `If-Modified-Since`, pero sí lo hacen bien con `If-None-Match`. Por otra parte algunos servidores de cache antiguos no soportan `ETag`. Por lo tanto se aconseja mezclar `ETag` y `Last-Modified` para un óptimo control de cache.

Sin embargo estos enfoques no impiden que la cache tenga que hacer una llamada a servidor y esperar la respuesta. Por muy sencillo que sea el procesamiento en servidor, y por muy poca información que transporte la petición y la respuesta, seguimos teniendo la penalización de la latencia, que en algunos casos, como en las redes móviles, suele ser alta. Podemos mejorar esto, y evitar peticiones `GET` condicionales innecesarias al servidor, si tenemos alguna idea de la frecuencia de cambio de los datos. Por ejemplo, en un servicio REST que sirva noticias para un periódico, podemos aguantar sin problemas aunque tengamos datos obsoletos de hace cinco minutos. Por lo tanto podríamos configurar la cache para que al menos aguante cinco, y no intente refrescar los datos en ese intervalo de tiempo. Esto se puede conseguir con el uso de `Cache-Control` y `Expires`. Se usa la cabecera Expires por compatibilidad con HTTP 1.0, y en caso de que ambas se contradigan `Cache-Control` tiene prioridad. Este es el motivo por el que en las respuestas de los anteriores ejemplos se incluyen estas cabeceras. Si pedimos los datos a un nodo de cache, y no se ha cumplido el plazo especificado mediante `Cache-Control` y/o `Expires`, la cache nos devolverá lo que tenga cacheado, y no se molestará en emitir una petición `GET` condicional al servidor. Sólo cuando este tiempo expire, la cache volverá a intentar comprobar si los datos han cambiado o no, normalmente usando tanto `If-Match` como `If-Modified-Since`.

Como se ve, el uso combinado de `GET` condicionales con las cabeceras de control de cache basada en tiempos, nos permiten un control fino de la cache. De esta forma podemos aprovechar la

infraestructura de cache de nuestra red, e incluso de internet, pero a la vez asegurarnos que el cliente de nuestro servicio REST no se quede con datos obsoletos y vea los cambios.

5.5 Multiidioma

Mediante el protocolo HTTP podemos construir un soporte multiidioma robusto en nuestros servicios REST.

Mediante la cabecera `Accept-Charset` el cliente puede especificar al servidor los juegos de caracteres que soporta por orden de preferencia. Podemos especificar una lista de juegos de caracteres, e incluso especificar preferencia mediante el parámetro `q`. Se trata de un mecanismo similar al usado en la cabecera `Accept` para especificar preferencia en tipos MIME. Si no se define el juego de caracteres se escogerá ISO-8859-1 por defecto.

Lo mismo podemos hacer con la cabecera `Accept-Language`, definir una lista por orden de preferencia de los lenguajes que desea el usuario. El servidor debe incluir en la respuesta la cabecera `Content-Language` indicando el idioma escogido.

Ejemplo:

```
1 GET /rest/libro/465 HTTP/1.1
2 Accept-Charset: iso-8859-5, unicode-1-1;q=0.8
3 Accept-Language: da, en-gb;q=0.8, en;q=0.7
4
```

De esta forma podemos elegir tanto el idioma como el juego de caracteres más adecuado a nuestro cliente, incluso si no tenemos en nuestro sistema información previa de él. Este mecanismo, en conjunción de otro más tradicional, como el de preferencias de usuario, nos permite implementar una internacionalización de nuestra API de forma bastante potente.

5.6 Prácticas básicas de seguridad en REST

Como en todo sistema de servicios web, debemos preocuparnos por la seguridad. Éste es un campo muy complejo, pero podemos cumplir algunas normas básicas de seguridad sin gran esfuerzo, y evitar un gran número de ataques.

Cuidado con los identificadores

Lo primero que hay que tener en cuenta es no usar identificadores predecibles, como los autoincrementados. Esto puede permitir a un posible atacante ir averiguando identificadores válidos de recursos que despues pueden ser usados en otro ataque.

Otro posible riesgo consiste en usar las claves primarias de nuestro sistema de persistencia para montar las URIs. Esto nos puede exponer a otros ataques como la inyección de SQL o de código.

Una posible solución es tener en formato persistente un mapeo entre las URIs y las verdaderas claves primarias.

Otra solución más eficiente puede ser construir la clave primaria a partir de la URI mediante una clave secreta. Se trataría de hacer una hash criptográfica de la URI con un secreto.

```
PK = HASH(URI+SECRETO)
```

Podemos reforzar este método teniendo un secreto distinto por "rango" de URIs, o quizás por colección.

Privacidad

La privacidad es otro aspecto a tener en cuenta. En recursos REST sensibles los datos no deberían transmitirse en claro por la red. La forma más obvia de solucionar esto es mediante el uso de HTTPS.

Si queremos un mayor nivel de seguridad, o incluso prescindir de HTTPS, podemos añadir una capa de encriptación a nivel de aplicación, de forma que la petición HTTP no esté encriptada, sino sólo su cuerpo. Esto sin embargo tiene como problema que disminuye la interoperabilidad al usar un sistema de encriptación no estándar.

Autenticación y autorización

Un punto importante de seguridad es conocer como realizar la autenticación y la autorización. HTTP nos proporciona un mecanismos desafío/respuesta mediante la cabecera `WWW-Authenticate`.

La idea es que cuando se realiza una petición a un recurso REST protegido, y la petición no lleva credenciales adecuadas, el servidor responde con un `401` que contiene una cabecera `WWW-Authenticate`. En dicha cabecera se detalla el tipo de credenciales que se necesita y el dominio de seguridad.

```
GET /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json

```

El servidor respondería:

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Basic realm="catalog"

```

En este caso el servidor indica que se necesitan las credenciales para el área de seguridad "catalog", credenciales que han de corresponder al protocolo "Basic"[2]. El cliente debe responder al desafío repitiendo la petición pero esta vez indicando en la cabecera `Authorization` las credenciales pertinentes:

```
GET /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
Authorization: Basic Y29udHJhc2XxYTpzZWNyZXRh

```

Si el servidor considera que las credenciales son suficientes, permite el acceso al recurso. Si por el contrario considera que el cliente, aunque posee credenciales válidas, no posee permisos suficientes para acceder al recurso, devuelve un `403`.

```
HTTP/1.1 403 Forbidden
Content-Type: application/json

{"reason":"Not enough security level"}
```

Puede ocurrir que el servidor dictamine que la credenciales no identifican a un usuario del sistema. En ese caso debe devolver un `401` con un nuevo `WWW-Authenticate`.

En la especificación se definen al menos dos algoritmos de autenticación. El primero es el llamado "Basic", y las credenciales de éste son simplemente la codificación en base 64 de la concatenación del usuario y password separados por ":". Al mandarse el usuario y contraseña en claro, nos vemos forzados a usar HTTPS si queremos tener algo de seguridad.

El otro algoritmo, "Digest", se considera más seguro y usa unas credenciales que no contienen la password en claro, como es el caso de "Basic". Esto permite autenticarse con seguridad sobre HTTP, sin necesidad de usar HTTPS. Más detalles en `http://tools.ietf.org/html/rfc2617`. Sin embargo "Digest" es susceptible de ataques "Man-In-The-Middle" y por otro lado no proteje el contenido del mensaje en si, sino sólo las credenciales. Es por esto que en la práctica se suele preferir el algoritmo "Basic" pero siempre sobre HTTPS.

[2] Autenticación "Basic" y "Digest" de HTTP: http://tools.ietf.org/html/rfc2617

¿Necesitamos autenticarnos una y otra vez?

En principio el protocolo HTTP guarda silencio con respecto a esta pregunta. El diseño más simple consiste en enviar una y otra vez la cabecera `Authorization` en cada petición. Esto puede tener algunos problemas prácticos de implementación.

Normalmente los servidores no guardan las credenciales, tales como las contraseñas, en claro dentro su base de datos. Si hicieran esto y algún atacante consiguiera acceder a las máquinas servidoras, se podría hacer con las contraseñas de todos los usuarios. La práctica habitual no es guardar la contraseña, sino una hash criptográfica de ésta concatenada con alguna "sal" (una clave secreta aleatoria, única para cada contraseña). Para autenticar a un usuario, se concatena la contraseña que nos mánda en la cabecera `Authorization` con la "sal" que corresponde al usuario, se hace la hash criptográfica de esto, y se compara con el valor almacenado en la BBDD. De esta forma las claves están seguras en la BBDD… o no. El problema reside en que las hash criptográficas están diseñadas para ser calculadas de forma muy eficiente. Esto hace posible un ataque por fuerza bruta por parte de un hacker (existe hardware especial, relativamente barato, para hacer esto). ¿Cómo podemos protegernos de esto?

La idea es usar una hash criptográfica *difícil* de calcular. En concreto hay dos algoritmos similares: BCRYPT[3] y PBKDF2[4]. Estas hashes toman un parámetro extra que es el "coste" de calcular la hash. Cuanto más coste más lenta serán de calcular. De esta manera podemos usar un valor para este parámetro que haga que la hash tarde entre 250 y 500 milisegundos en ser calculadas en el servidor. Esto soluciona el problema de los ataques de fuerza bruta, porque si el hardware mejora, podemos aumentar el valor del parámetro "coste" para que dichos ataques no sean factibles.

Sin embargo esto nos genera otro problema, si calcular la hash es costoso, entonces el proceso de autenticación será lento también, y por lo tanto no podemos autenticarnos en cada petición si queremos que nuestros servidores escalen.

La solución es obviamente autenticarse sólo la primera vez, y para ello usaremos un token de seguridad. Si la petición incorpora dicho token, el servidor confiará en ella y le dará acceso. Si no se incorpora dicho token se procede a la autenticación tal y como vimos en la sección anterior. Dicho token se puede incorporar en la petición ya sea en una *cookie* o en una cabecera HTTP especial (podríamos usar `Authenticate` con nuestro esquema propio de seguridad), y debería caducar cada cierto tiempo, normalmente a los pocos minutos.

¿Cómo generar dicho token? Ya hemos visto algunas técnicas, la más popular es usar un HMAC:

```
1 PUBLIC_PART = UUID() + ":" + SECURITY_LEVEL + ":" + TIMESTAMP
2 SIGNATURE = HMAC(SERVER_SECRET, PUBLIC_PART)
3 TOKEN = SIGNATURE + "_" + PUBLIC_PART
```

Mediante `UUID` obtenemos un identificador único universal, `SECURITY_LEVEL` indica el rol de seguridad que tenemos, y `TIMESTAMP` nos permite detectar y descartar los token antiguos. Si el token

[3] http://en.wikipedia.org/wiki/Bcrypt

[4] http://en.wikipedia.org/wiki/PBKDF2

está caducado o es inválido, se requiere pasar por el proceso de autenticación de nuevo. Si el token es válido en la respuesta incluimos un nuevo token, con un `TIMESTAMP` más reciente y un nuevo `UUID`.

Algunos lectores pensarán, "pero si este token es exactamente lo mismo que un token o identificador de sesión". Realmente los identificadores de sesión se suelen implementar de la misma manera, pero hay una diferencia sutil entre un token de sesión y uno de seguridad: los tokens de seguridad no representan un espacio dedicado de memoria en el servidor. En una API REST no hay sesión, por lo tanto no hay que almacenar en ningún sitio los token generados, ni reservar un espacio de memoria especial para cada cliente. Lo bueno de esta forma de construir tokens es que podemos validarlos sin necesidad de todo esto, basta calcular la `HMAC` de la parte pública y ver si el resultado concuerda con la firma del token. De hecho en el párrafo anterior, el autor recomienda *cambiar el token en cada ciclo petición/respuesta.*

5.7 Actualizaciones parciales

El problema

Las técnicas de actualización explicadas en el capítulo anterior nos sirven sólo si queremos actualizar por completo un recurso, pero no son válidas si necesitamos actualizar sólo unos campos y otros no.

Si lo que queremos hacer realmente es una actualización parcial, la mejor práctica en este caso es revisar el modelo de nuestros recursos. Normalmente cuando surge esta necesidad, es porque queremos operar con trozos más pequeños del recurso original. En estos casos es mejor dividir el recurso principal, y crear recursos detalle con sus propias URI. El recurso principal contendrá sólo la información más esencial y enlaces a todos los recursos que contienen información más detallada o secundaria. De esta forma cada recurso detalle tiene su propia URI y puede ser actualizado de forma independiente.

Sin embargo, pueden existir algunos casos de uso donde este enfoque no sea práctico. Supongamos que por motivos de rendimiento queremos intercambiar entre el cliente y el servidor únicamente los cambios que se produzcan en el recurso, y ahorrar así ancho de banda.

En estos casos estaríamos tentados de usar `PUT` para mandar sólo esos cambios, algo como lo que sigue:

```
PUT /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
If-Match: W/"686897696a7c876b7e"
Content-Type: application/json

{
  "price": { "currency":"€", "amount":100},
  "author": [ "Enrique Gómez Salas", "Juan Pérez" ]
}
```

Como vemos en este ejemplo, sólo enviamos un documento parcial, con sólo los campos a actualizar. El servidor interpretaría que los campos que no se encuentren en la petición debe dejarlos intacto. Pero esta forma de usar `PUT` no es REST, ya que la especificación HTTP dice que el contenido del cuerpo de la petición pasa a ser el nuevo estado del recurso, es decir, el estado del recurso se debe sobreescribir por completo con lo que viene en la petición. Por lo tanto debemos buscar otra alternativa.

Solución de la vieja escuela: `POST`

Esa alternativa sería `POST`. Normalmente se interpreta que `POST` va a añadir contenido por el final cuando se usa para actualizar, pero la especificación sólo nos indica que esa es una de las posibles acciones que se pueden admitir. El método `POST` sobre una URI se puede interpretar según el estándar de otras formas. En el caso que nos ocupa se puede usar la interpretación de "añadir" en vez de crear un recurso subordinado. La idea es que podemos ir "añadiendo" información al recurso poco a poco. El truco para hacer una actualización parcial es usar un tipo MIME que representa un cambio de estado, o "diff", que se "añade" al estado del servidor. Si mandamos varios de estos "diff" al servidor, éste los va "sumando" y el recurso se va actualizando incrementalmente.

```
POST /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/book-diff+json

[{
    "change-type":"replace",
    "location":"price/amount",
    "value":100
 },
 {
   "change-type":"append",
   "location":"author",
```

```
    "value":"Juan Pérez"
 }]
```

Sin embargo hay que tener cuidado, `POST` al contrario que `PUT` no es idempotente, ¿qué debería hacer el servidor si vuelvo a repetir la petición pensando que no se ejecutó correctamente en el servidor? Una implementación poco sofisticada del servidor podría volver a aplicar los cambios:

```
HTTP/1.1 200 Ok
Content-Type: application/json;charset=utf-8

{
  "id": "http://www.server.com/rest/libro/465",
  "author": [ "Enrique Gómez Salas", "Juan Pérez", "Juan Pérez" ],
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency":"€", "amount":100}
}
```

En este caso hemos enviado el mismo "diff" de forma duplicada. Esto hace que la segunda petición sea inconsistente, ya que al haberse procesado el primer "diff" (sin que el cliente lo supiera), el segundo no tiene sentido. En estos casos el servidor debería fallar, con un `409`, y no duplicar la petición.

```
HTTP/1.1 409 Conflict

```

Se falla con `409` para indicar que existe un conflicto entre la petición y el estado del recurso. Sin embargo, no se especifica en qué consiste este conflicto, ya que `409` es un código de error bastante genérico. Otro problema es que a veces el servidor no podría detectar que la petición es duplicada inspeccionando sólo el cuerpo de esta. En este caso es sencillo, ya que el autor está duplicado, pero en otros casos la cosa no sería tan sencilla.

Debido a estos problemas es mejor usar peticiones condicionales mediante `ETag` e `If-Match` cuando hagamos actualizaciones parciales. Por un lado es más explícito y por otro el servidor puede detectar las peticiones duplicadas de forma más sencilla.

```
POST /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
If-Match: W/"686897696a7c876b7e"
Content-Type: application/book-diff+json

[{
    "change-type":"replace",
    "location":"price/amount",
    "value":100
 },
 {
   "change-type":"append",
   "location":"author",
   "value":"Juan Pérez"
 }]
```

Y si existe una duplicación o un problema de carrera el servidor responde con:

```
HTTP/1.1 412 Precondition Failed

```

Que es mucho más explícito, sencillo de implementar y de depurar.

Otro punto importante a la hora de hacer actualizaciones parciales es usar un tipo MIME que represente explícitamente un "diff". Si no nuestra API sería confusa. Ya existen tipos MIME en proceso de estandarización para representar "diff" entre documentos. Algunos están basados en XPath[5], y otros en formato JSON[6].

Solución a la última moda: PATCH

El problema del sistema anteriormente descrito para realizar actualizaciones parciales es que `POST` es un método genérico que admite casi cualquier tipo de operación que no sea ni segura ni idempotente. En este sentido se está convirtiendo en una especie de cajón de sastre donde podemos modelar casi cualquier operación que no encaje en el resto del marco de REST. Esto es un problema y puede llegar a hacer que los consumidores de nuestros recursos se confundan al usar `POST`.

Para solucionar este problema se ha propuesto añadir un nuevo método, el método `PATCH`, que todavía se encuentra en proceso de estandarización[7]. Al igual que `POST` es un método no seguro

[5] An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors: http://tools.ietf.org/html/rfc5261

[6] JSON Patch: http://tools.ietf.org/html/draft-ietf-appsawg-json-patch-01

[7] PATCH Method for HTTP: http://tools.ietf.org/html/rfc5789

y no idempotente, pero tiene una semántica explícita de actualización parcial y sólo admite tipos MIME que representen "diffs". Al ser más explícito la API es más clara.

Todo lo explicado anteriormente sobre actualizaciones parciales se aplica a `PATCH`. Así el ejemplo anterior quedaría:

```
PATCH /rest/libro/465 HTTP/1.1
Host: www.server.com
Accept: application/json
If-Match: W/"686897696a7c876b7e"
Content-Type: application/book-diff+json

[{
    "change-type":"replace",
    "location":"price/amount",
    "value":100
 },
 {
   "change-type":"append",
   "location":"author",
   "value":"Juan Pérez"
 }]
```

Es exactamente igual que `POST` pero usando `PATCH` que es mucho más explícito y deja menos a la interpretación.

Una forma de saber si un recurso admite el método `PATCH` es hacer una petición con el método `OPTIONS`:

```
OPTIONS /rest/libro/465 HTTP/1.1
Host: www.server.com

```

Y la respuesta:

```
HTTP/1.1 200 OK
Allow: GET, PUT, POST, OPTIONS, HEAD, DELETE, PATCH
Accept-Patch: application/book-diff+json
Content-Length: 0

```

La respuesta indica los métodos aceptables por el recurso. En el caso de que se soporte `PATCH`, se debe incluir la cabecera `Accept-Patch` que indica que formatos mime se pueden usar con el método `PATCH`.

Sin embargo al ser un método nuevo que todavía no está estandarizado podemos tener problemas de interoperabilidad. Puede ser que el servidor no lo implemente, o que algún firewall bloquee dicho método. Para evitar esto podemos usar la cabecera `X-HTTP-Method-Override`. Esta cabecera es ignorada por los nodos y servidores antiguos, pero permite indicar a los servidores más modernos que queremos usar un método diferente del especificado en la petición HTTP. Así, el ejemplo anterior quedaría como:

```
POST /rest/libro/465 HTTP/1.1
Host: www.server.com
X-HTTP-Method-Override: PATCH
Accept: application/json
If-Match: W/"686897696a7c876b7e"
Content-Type: application/book-diff+json

[{
    "change-type":"replace",
    "location":"price/amount",
    "value":100
 },
 {
   "change-type":"append",
   "location":"author",
   "value":"Juan Pérez"
 }]
```

Como se observa, para cualquier servidor o nodo que no entienda `PATCH`, la petición sería un `POST`. Pero en los casos en los que el servidor sí entienda `PATCH`, ignorará `POST`.

Actualmente `PATCH` está soportado por las APIs públicas de grandes compañías como Google o GitHub.

5.8 Versionado de API

Como en cualquier sistema de servicios web, cuando usamos REST estamos publicando una API. Si cambiamos esa API entonces vamos a romper la interoperabilidad con los clientes antiguos. ¿Cómo podemos cambiar la API de forma que tenga el mínimo impacto posible?

Uno de las posibles cosas que pueden cambiar es el formato. Es muy común que se añadan nuevos campos para reflejar nueva información, o que empecemos a soportar nuevos tipos MIME.

Realmente esto no suele ser un problema, basta con que el cliente ignore los campos que no entienda o no pueda procesar.

Si usamos XML hay que tener en cuenta que a la hora de definir su XML Schema, hay que dejar puntos de extensión. Estos puntos de extensión relajan la validación de XML de tal manera que cuando se encuentra un elemento no esperado, el validador no falle. En el futuro podemos evolucionar el XML Schema y añadir nuevas estructuras, pero siempre conservando estos puntos de extensión. Normalmente estos puntos de extensión dentro del XML Schema se realizan con las etiquetas `<any>` y `<anyAttribute>`, que permiten al documento XML contener cualquier elemento y cualquier atributo en los puntos donde se definió. Un ejemplo:

```
<xsd:complexType name="BookType">
  <xsd:sequence>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:element name="ISBN" type="xsd:string"/>
    <xsd:element name="author" type="xsd:string"/>
    <xsd:element name="date" type="xsd:gYear"/>
    <xsd:element name="description" type="xsd:string"/>
    <xsd:any namespace="##any" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

El último elemento `<xsd:any>` en el XML Schema anterior permite que documentos XML de tipo `BookType`. De esta manera este tipo podría evolucionar a contener nuevos elementos hijos, siempre y cuando aparezcan después de `description`.

Si por cualquier razón hacemos un cambio en el formato de datos que vaya a romper los clientes de forma inevitable, podemos aprovechar la capacidad multimedia de HTTP en nuestro provecho. Podemos publicar el nuevo formato con otro nombre, y seguir soportando el antiguo. El formato antiguo lo podemos marcar como obsoleto en la documentación, pero seguir sirviéndolo mientras los clientes que lo consuman no sean una ínfima minoría.

Por ejemplo, si soportábamos `application/x-libro+xml`, y hacemos un cambio que inevitablemente no pueda ser retrocompatible, haremos este cambio en otro tipo MIME `application/x-libroV2+xml` y dejaremos el original tal como estaba. De esta forma los clientes antiguos pueden seguir consumiendo el formato antiguo, y los nuevos aprovechar las ventajas del nuevo tipo MIME.

Otra cosa que puede cambiar es la URI. Realmente esto hay que evitarlo en la medida de lo posible, ya que puede romper los enlaces que nuestros clientes tuvieran guardados. En cualquier caso, si por cualquier razón, nos vemos obligados a modificar la URI, podemos solucinar el problema de forma sencilla. Sólo hay que configurar una nueva redirección permanente a la nueva URI y asunto resuelto.

5.9 ¿Necesito una sesión HTTP?

Es importante enfatizar que el protocolo HTTP no tiene sesión. La mal llamada sesión HTTP no es más que un truco implementado por los servidores de aplicaciones, donde se reserva un espacio de memoria que puede ser referenciado mediante un identificador. Este identificador se pasa una y otra vez entre el servidor y el cliente, ya sea mediante una cabecera, ya sea dentro de la URI o mediante cookies.

Desde el punto de vista de servicios web REST esto no tiene mucho sentido, ya que los servicios son *stateless* y no hay que almacenar en ningún sitio la historia de la conversación entre cliente y servidor.

Quizás algún lector estuviera tentado de usar esta pseudosesión HTTP como una forma de no tener que mandar una y otra vez el token de autenticación y simplificar la programación. Sin embargo el autor desaconseja dicha práctica por varias razones:

- Es una práctica no estándar desde el punto de vista de HTTP y la web.
- Es poco escalable, ya que impone al servidor una sobrecarga, al tener que reservar memoria para la sesión.
- Abre la posibilidad a implementar servicios web *stateful*, con memoria. Este tipo de servicios, a parte de no ser tan escalables, imponen un mayor acoplamiento entre cliente y servidor disminuyendo la interoperabilidad. Los servicios *stateless* son más sencillos, ya que sólo necesitan declarar explícitamente toda la información necesaria para consumirlos en forma de parámetros. Los servicios *stateful* necesitan además publicar un modelo de cómo cambia la respuesta del servicio en función del estado de la conversación.
- No se gana nada en rendimiento, ya que si antes tenemos que estar mandando continuamente el token de seguridad, usando sesión deberíamos mantar continuamente el identificador de sesión, ya sea en una *cookie* o en la URI.

Por lo tanto el uso de sesión HTTP en servidor es una mala práctica desde el punto de vista de los servicios REST.

5.10 Peticiones asíncronas o de larga duración

Algunas operaciones pueden representar acciones asíncronas, de larga duración, o que tal vez necesitan ser encoladas y realizadas por un proceso *batch* por la noche. En estos casos puede ser interesante no responder con `201` o `200`, sino con `202`.

El código `202` indica que la operación ha sido aceptada por el servidor pero que tardará un tiempo en realizarse, y es ideal para modelar peticiones asíncronas. Normalmente en la respuesta se incluye una referencia a una URI que nos indicará el progreso de la operación, y que podremos consultar cada cierto tiempo.

Supongamos que queremos realizar el pago de un pedido:

```
POST /rest/pago HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/json

{
  "payment-method":"visa",
  "cardnumber": 1234567812345678,
  "secnumber":333,
  "expiration": "08/2016",
  "cardholder": "Pepe Pérez",
  "amount": 234.22
}
```

Si la pasarela de pago va a tardar o simplemente se debe producir un workflow interno de aprobación manual, la petición no se puede dar por terminada, asi que lo mejor es avisar al cliente que estamos trabajando en ella:

```
HTTP/1.1 202 Accepted
Location: /rest/jobs/XDC3WD

```

El servidor responde con `202` indicando que la petición ha sido aceptada pero no se ha procesado todavía. En la cabecera `Location` se devuelve la URL de un recurso que podemos consultar para informarnos del progreso de dicha petición. Para ello basta con hacer `GET` de manera periódica a dicha URI.

```
GET /rest/jobs/XDC3WD HTTP/1.1
Host: www.server.com
Accept: application/json

```

En el caso de que el pago no estuviera listo, el servidor respondería con:

```
HTTP/1.1 200 Ok
Content-Type: application/json

{
  "status":"working"
}
```

Tarde o temprano el proceso terminaría. Si acaba con éxito:

```
HTTP/1.1 200 Ok
Content-Type: application/json

{
  "status":"ok",
  "result":"/rest/pago/DEF245SW"
}
```

Y bastaría hacer un `GET` sobre `/rest/pago/DEF245SW` para obtener el recurso que representa que el pago se realizó satisfactoriamente.

```
GET /rest/pago/DEF245SW HTTP/1.1
Host: www.server.com
Accept: application/json

```

Y la respuesta:

```
HTTP/1.1 200 Ok
Content-Type: application/json

{
  "payment-method":"visa",
  "cardnumber": 1234567812345678,
  "secnumber":333,
  "expiration": "08/2016",
  "cardholder": "Pepe Pérez",
  "amount": 234.22,
  "status":"ok"
}
```

Es importante tener en cuenta que la petición no tiene porque terminar con éxito, sino que tal vez acabe con un error. En este caso hay dos opciones. La primera es simpliemente devolver el detalle del error en el propio recurso `/rest/jobs/XDC3WD`.

```
HTTP/1.1 200 Ok
Content-Type: application/json

{
  "status":"error",
  "errorDetail":"No money"
}
```

Otra opción sería indicar el error en el propio recurso "pago".

```
HTTP/1.1 200 Ok
Content-Type: application/json

{
  "status":"error",
  "result":"/rest/pago/DEF245SW"
}
```

Si hicieramos `GET` sobre `/rest/pago/DEF245SW` obtendríamos:

```
HTTP/1.1 200 Ok
Content-Type: application/json

{
  "payment-method":"visa",
  "cardnumber": 1234567812345678,
  "secnumber":333,
  "expiration": "08/2016",
  "cardholder": "Pepe Pérez",
  "amount": 234.22,
  "status":"error",
  "errorDetail":"No money"
}
```

Aunque no es habitual, el servidor podría ofrecer la posibilidad de cancelar el proceso haciendo un `DELETE` sobre `/rest/jobs/XDC3WD`. Aunque si esto se puede hacer o no, depende de la lógica de negocio.

Antes de responder con `202` sería deseable que el servidor hiciera una validación preliminar de la petición, para que en el caso de que esta no fuera válida, responder directamente con un error `4xx`. Sólo deberíamos devolver `202` si el servidor determina que la petición parece válida.

5.11 URIs desechables y recursos “virtuales”

Concepto

Anteriormente se ha comentado que es deseable siempre usar métodos HTTP idempotentes, ya que nos permiten una recuperación más simple de los posibles cortes y congestiones de la red.

El único método HTTP que no es idempotente es `POST`, ¿existirá alguna forma de prescindir de él? ¿O quizás se puede usar `POST` de alguna manera que no sea tan peligroso? Son preguntas importantes ya que `POST` es la manera más simple de crear un recurso, y de modelar operaciones genéricas.

La versión idempotente de `POST` podría ser `PUT`, pero tiene como problema que el consumidor del recurso REST necesita poder definir las URIs de los nuevos recursos. Como vimos anteriormente esto podría disminuir la interoperabilidad, ya que el algoritmo de generación de URIs tendría que estar implementado en el cliente y ser consistente con el servidor. Por la misma razón esto puede generar problemas de seguridad.

La forma de solucionar esto es haciendo trabajar juntos a `PUT` y a `POST`. En este diseño, cuando el cliente quiere crear un nuevo recurso, o quizás ejecutar un comando arbitrario, le pide permiso al servidor mediante una llamada `POST`. El servidor puede rechazar la llamada o bien concederle permiso. Si le concede permiso, el servidor genera una nueva URI que es devuelta al cliente. Esta URI representa un nuevo recurso que está vacio, y por lo tanto el servidor no necesita ni persistir ni almacenar en memoria. Se trata pues de un recurso “virtual” que no ocupa espacio en servidor. De esta forma si el cliente piensa que la petición se ha perdido, puede repetir la petición. Lo peor que podría pasar es que se creara una nueva URI para un nuevo recurso, pero como este recurso no ocupa nada en el servidor no hay peligro. El cliente puede después usar dicha URI para actualizar el recurso vacío mediante `PUT`. Es en ese momento cuando el servidor asigna memoria y realiza persistencia, y cuando se crea realmente el recurso de forma física. Si el cliente piensa que esta última petición falló, puede repetir la petición sin problemas, ya que `PUT` es idempotente.

Resumiendo:

1. El cliente hace `POST` a un recurso “factoría”, encargado de crear recursos “virtuales”.
2. El servidor devuelve la URI de ese nuevo recurso “virtual”
3. El cliente hace `PUT` a la URI devuelta con los datos que el considera debe tener el recurso
4. El servidor actualiza el recurso con los datos transmitidos en el `PUT`, y lo persiste.

Como se aprecia el truco consiste en que el recurso no se crea realmente hasta que no se recibe el `PUT`. Sin embargo queda un problema por resolver: si no almacenamos nada ni en memoria ni en soporte persistente, ¿cómo puede el servidor saber que la URI que le llega en el `PUT` es legítima? Al fin y al cabo no queremos que el cliente genere las URIs, sino que el servidor controle tal cosa. Lo ideal sería que si el cliente nos manda un `PUT` a una URI que no existe devolvamos un código `403`, indicando que el cliente no tiene permiso para generar recursos en URIs que no sean asignadas por el servidor.

Una solución inocente sería almacenar en algún sitio, en memoria tal vez, las URIs de los recursos, y chequear contra esta información la URI de la petición `PUT`. Sin embargo esto contradice el principio de que los recursos "virtuales" no deben ocupar espacio en servidor. De lo contrario el cliente no podría reintentar el `POST` con la seguridad de que no va a pasar nada malo.

La solución consiste en generar una URI desechable. Las URIs desechables pueden ser **reconocidas como legítimas por el servidor a simple vista, y no pueden ser falsificadas por el cliente**. De esta forma no se se necesita almacenar la URI ni en memoria ni en soporte persistente. ¿Cómo generar una URI desechable? La forma más simple es tener en el servidor una clave secreta y un generador de UUIDs. La idea sería que cada vez que el servidor reciba un `POST`, genere un nuevo UUID, le concatene la clave secreta y el resultado lo pase por una función hash criptográfica. El resultado de este proceso es una firma de la URI. Esta firma es concatenada con el UUID anteriormente generado. Esto termina generando un identificador que se transforma en el último segmento de la ruta de la URI.

```
1 FIRMA = BASE64(HASH(UUID+SECRETO))
2 URI = URI_RECURSO_PADRE + "/" + UUID + "_" + FIRMA
```

Cuando el servidor reciba un `PUT`, es tan sencillo como extraer el UUID del último segmento de la ruta de la URI, calcularle la firma, y ver si coinciden. Si es así la URI es legítima, si no, debemos devolver `403`. Esta lógica puede hacerse genérica a todas las peticiones `PUT` e implementarse en un filtro común a todas las URIs.

Una ventaja añadida de esta forma de generar URIs es que ni los `UUIDs` ni la `HASH` ni el `SECRETO` son predecibles o accesibles desde el exterior[8]. De esta forma eliminamos el problema de que nuestras URIs sean predecibles.

Creando recursos con URIs desechables

Es sencillo crear recursos con URIs desechables. Primero el cliente hace un `POST` al recurso colección, sin datos:

```
1 POST /rest/libro HTTP/1.1
2 Host: www.server.com
3 Accept: application/json
4
```

Y la respuesta:

[8]Esto no es estrictamente cierto, casi todas las técnicas de seguridad criptográfica pueden ser atacadas, incluida esta. Para una forma aun más segura de generar URIs o tokens se puede usar una evolución de esta técnica conocida como HMAC: http://en.wikipedia.org/wiki/Hash-based_message_authentication_code

```
HTTP/1.1 201 Created
Location: http://www.server.com/rest/libro/BlVX4BmJJFch7sF_C4DUtaWmJLOf+Cz

```

El cliente, usando la URI que le ha sido devuelta en la cabecera location, realiza un `PUT` con los datos de la nueva entidad.

```
PUT /rest/libro/BlVX4BmJJFch7sF_C4DUtaWmJLOf+Cz HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/json

{
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency":"€", "amount":50}
}
```

Con esta petición el "recurso virtual" representado por la URI se transforma en un recurso con los datos especificados en la petición `PUT`. La respuesta:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=utf-8

{
  "id": "http://www.server.com/rest/libro/BlVX4BmJJFch7sF_C4DUtaWmJLOf+Cz",
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { "currency":"€", "amount":50}
}
```

De esta forma podemos crear un recurso en dos fases. Primero solicitar la URI, y después confirmar mediante un `PUT` De esta forma podemos repetir todas las veces que queramos la petición `POST` porque generar URIs de "recursos virtuales" no consumen apenas recursos, y por supuesto podemos repetir la petición `PUT` para transformar el "recurso virtual" en uno real, ya que `PUT` es idempotente.

5.12 Procesos de negocio

Una forma de modelar operaciones de negocio genéricas es transformar las operaciones en recursos. Por ejemplo, si queremos comprar un libro, en vez de tener una URI `/rest/comprar`, la URI debería

ser `/rest/compra`. Nótese la diferencia entre el verbo y el nombre. De esta forma el proceso se transforma en un recurso REST que puede pasar por varios estados, en función de las peticiones que reciba y del procesamiento interno del servidor.

¿Como haríamos una compra? Vamos a usar el patrón de URI desechable. Primero haríamos un `POST` al recurso "compra" para que nos asignase un identificador de nueva compra en forma de URI:

```
POST /rest/compra HTTP/1.1
Host: www.server.com
Accept: application/json

```

Y la respuesta:

```
HTTP/1.1 201 Created
Location: http://www.server.com/rest/compra/BlVX4BmJJFch7sF_C4DUtaWmJLOf+Cz

```

Después podemos confirmar la compra mediante un `PUT` a dicha URI, indicando los datos de compra en el cuerpo del `PUT`

```
PUT /rest/compra/BlVX4BmJJFch7sF_C4DUtaWmJLOf+Cz HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/json

{
  "payment-method":"visa",
  "cardnumber": 1234567812345678,
  "secnumber":333,
  "expiration": "08/2016",
  "cardholder": "Pepe Pérez",
  "order": "http://www.server.com/rest/orders/345",
  "status": "created"
}
```

Lo interesante es que podríamos consultar el estado de la compra haciendo un `GET` a dicha URI.

```
GET /rest/compra/BlVX4BmJJFch7sF_C4DUtaWmJLOf+Cz HTTP/1.1
Host: www.server.com
Accept: application/json

```

Y la respuesta indicando que la compra está en espera de que haya existencias para nuestro pedido:

```
HTTP/1.1 200 Ok
Content-Type: application/json

{
  "payment-method":"visa",
  "cardnumber": 1234567812345678,
  "secnumber":333,
  "expiration": "08/2016",
  "cardholder": "Pepe Pérez",
  "order": "http://www.server.com/rest/orders/345",
  "status": "awaiting-stock"
}
```

De esta manera podemos hacer *polling* sobre el recurso para saber en que estado se encuentra. En función del estado en que se encuentre el proceso y las reglas de negocio, el cliente podría intentar modificarlo mediante nuevas peticiones. Tal vez podría actualizar el recurso de pedido añadiendo más productos o alterando la dirección de entrega, o quizás cancelar la compra.

Por ejemplo, para solicitar la anulación de la compra haríamos un nuevo PUT:

```
PUT /rest/compra/BlVX4BmJJFch7sF_C4DUtaWmJLOf+Cz HTTP/1.1
Host: www.server.com
Accept: application/json
Content-Type: application/json

{
  "payment-method":"visa",
  "cardnumber": 1234567812345678,
  "secnumber":333,
  "expiration": "08/2016",
  "cardholder": "Pepe Pérez",
  "order": "http://www.server.com/rest/orders/345",
  "status": "canceled"
}
```

El si se puede o no cancelar la compra y que implica esto depende de la lógica de negocio y en que estado se encuentre. Otra forma de cancelar la compra sería hacer una actualización parcial del campo "status", ya sea usando `PATCH` o `POST`.

Si la compra se encontrara en un estado tal que ya no se pudiera cancelar o modificar alguno de sus detalles, se debería devolver `409` (Conflict). En este tipo de escenarios de nuevo es mejor hacer peticiones condicionales mediante `If-Match`, `ETag`, y el código `412` en caso de conflicto. De esta forma el diálogo entre cliente y servidor sería mucho más explícito. Téngase en cuenta que conforme el servidor avance en la ejecución del proceso de negocio, el estado de este va a cambiar, con lo cual su `ETag` también lo hará.

No es el caso de la "compra", pero a veces la operación se puede deshacer por completo sin ningún problema. Si se puede hacer esto o no depende del dominio de negocio. En estos casos lo más simple sería usar `DELETE` para borrar el comando.

Como vemos, este diseño nos da una gran flexibilidad, no sólo a la hora de implementar operaciones arbitrarias, sino también procesos de negocio.

5.13 Procesos VS. Peticiones asíncronas

Como hemos visto, operaciones de larga duración o que pasan por varias fases pueden ser modeladas en nuestra API de dos formas.

La primera es modelarlas mediante operaciones asíncronas, donde el servidor responde con `202` y debemos hacer *polling* sobre un recurso que nos indique el estado actual en el que se encuentra la petición. La segunda es modelar explicitamente los estados por los que puede pasar la operación, y exponer el proceso de negocio como un recurso. ¿Cuál usar?

Es mejor usar un enfoque de petición asíncrona cuando el cliente no está interesado en los estados intermedios de la petición, y sólo pretende esperar a que esta se complete.

Por el contrario si los estados intermedios son significativos, y el cliente podría intentar cambiar o interactuar con el proceso mediante nuevas peticiones, es mejor modelar el proceso de forma explícita. Como veremos, el enfoque hypermedia se adapta muy bien a esta última filosofía.

6 Hypermedia APIs

6.1 Introducción

La facilidad que nos da REST de modelar APIs orientadas a datos, o CRUD, nos puede tentar a dejar nuestra API a ese nivel. Cada URI representa una tabla, o entidad de negocio, y mediante los verbos HTTP definimos las operaciones de edición y lectura. Es tan secillo que incluso podemos automatizar la publicación de nuestra API a partir del esquema de nuestra base de datos, suponiendo que no nos importe exponer este detalle de implementación interna al mundo entero claro. Este enfoque puede ser peligroso ya que podemos acoplar nuestra API web a la implementación concreta de nuestro modelo de persistencia.

Sin embargo el principal problema es que con una API orientada a dato, en realidad no estamos diseñando una web de recursos de información, sino un conjunto de operaciones desconectadas entre si que no representa realmente nuestro negocio. En este sentido, al diseñar CRUD, lo que estamos haciendo es exponer nuestra capa de datos mediante una API pública, y hacemos que todo el peso de interpretar e implementar la lógica de negocio caiga en el consumidor de nuestra API, con los lógicos problemas que esto acarrea. La API de nuestros servicios web deberían exponer los distintos casos de uso de nuestro sistema, y no una mera interfaz de persistencia de datos. Es exactamente la misma diferencia que hay entre una aplicación de mantenimiento de tablas, y una aplicación con una *UX* orientada a que los usuarios completen casos de uso de nuestro negocio de la forma más efectiva posible.

Por supuesto esto no es ningún problema si lo que queremos hacer es precisamente eso, exponer una capa de persistencia o acceso a datos sin ningún tipo de lógica más allá de validaciones básicas. Pero normalmente no es eso lo que se persigue, y nuestra API debería ser capaz de ofrecer un poco más de inteligencia.

Durante la primera fase de adopción de REST la mayoría de los proyectos definieron interfaces CRUD para sus recursos. Esto provocó que numerosos autores acuñara un nuevo término para REST: *Hypermedia As The Engine Of Application State* (**HATEOAS**), para señalizar que la API se basa en un enfoque de hypermedia y no CRUD. No es que realmente HATEOAS sea una cosa diferente de REST, sino que más bien es una aclaración al concepto de REST, haciéndo énfasis en el desarrollo de una web de servicios, en contraposición a diseñar un conjunto inconexo de fuentes de datos. Pero realmente la visión original de REST[1] no se distingue de HATEOAS, aunque lamentablemente la parte más importante de esta filosofía, el crear una web de recursos de información interrelacionadas, fue ampliamente ignorada en un principio.

Con el transcurrir del tiempo el acrónimo HATEOAS ha declinado en popularidad. Para mayor problema al hablar de *Application* se creó la confusión de si realmente HATEOAS permitía o no aplicaciones compuesta o *mashups* o si la API debía modelar el flujo de uso de la UI de la aplicación o no. Actualmente se prefiere hablar de APIs *hypermedia* sin más.

[1] El articulo original describiendo REST está públicamente accesible aquí: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

6.2 El concepto de hypermedia

El concepto de *hypermedia* es sencillo de entender. Se basa en hacer énfasis en el aspecto más esencial de la web: los recursos no están aislados, sino que están interrelacionados mediante *hiperenlaces*. De esta forma a partir de un recurso podemos descubrir como acceder al resto de ellos siguiendo los enlaces, y averiguar que acciones podemos realizar mediante los controles presentes en cada recurso.

Decimos que una API REST es *hypermedia*, cuando:

- No es un conjunto de puntos de entrada desconectados entre si, sino una **red de recursos** de información **conectados mediante hiperenlaces**. De esta forma el consumidor de nuestros servicios web no necesita saber todas las URIs de nuestros recursos REST, sino que sólo tendría que conocer una única URI, quizás de un recurso "home" de nuestro sistema, y seguir los enlaces entre recursos para adquirir la información que vaya necesitando.
- Cada recurso debería poder ser servido en tantos formatos (tipos MIME) como tenga sentido. Es decir, es **multimedia**
- Las **operaciones admisibles sobre cada recurso estan codificadas en forma de controles dentro de los documentos** que sirven como representación de éste. De esta forma **sólo podremos cambiar el estado del sistema mediante un control presente en el propio recurso**.

La idea general es aumentar al máximo la interoperabilidad disminuyendo la cantidad de información que necesitamos documentar para que un posible programador implemente un cliente de nuestra API. Esto sólo se puede conseguir si las capacidades de nuestro sistema se pueden descubrir siguiendo un conjunto sencillo de reglas o convenciones. El uso de *hypermedia* nos permite hacer precisamente esto: por un lado no es necesario documentar todas las URIs de tu sistema, sino **sólo una URI principal**, desde la cual puedes descubrir el resto del sistema; por otro lado no hay que describir si para un determinado recurso la inserción o la actualización se realiza con `PUT` o con `POST`, o si para borrarlo basta hacer `DELETE` o si hay que seguir otro proceso más complejo. Simplemente hay que examinar el documento obtenido por la anterior petición, y **buscar que enlaces y controles** hay presentes. Estos enlaces y controles son los que me indican qué conjunto de operaciones y recursos puedo hacer a continuación.

En resumen, *hypermedia* significa que sólo necesito un único punto de entrada (URI) a la API, y que puedo descubrir las capacidades de ésta simplemente inspeccionando los documentos recibidos en busca de enlaces y controles. Para descubrir nuevos recursos sólo hace falta navegar siguiendo los enlaces.

Por ejemplo en una tienda, podríamos tener una URI principal, `http://www.shop.com/home`, con enlaces a todos los subsistemas que ofrezcamos mediante REST.

```
GET /home HTTP/1.1
Host: www.shop.com
Accept: application/x-shop.home+json,application/json

```

Y la respuesta:

```
HTTP/1.1 200 Ok
Content-Type: application/x-shop.home+json

{
  "links": [
    {
      "href": "/home",
      "title": "Shop home",
      "rel": "self"
    },
    {
      "href": "/shop/products",
      "title": "Product catalog",
      "rel": "http://www.shop.com/rels/products"
    },
    {
      "href": "/shop/orders",
      "title": "Orders system",
      "rel": "http://www.shop.com/rels/orders"
    }
  ]
  //... other data ...
}
```

En este ejemplo la página principal actúa como directorio, y consiste principalmente en una lista de enlaces a los distintos subsistemas. ¿En qué consisten estos enlaces? Basicamente un enlace tiene cuatro atributos: `title`, `href`, `rel` y `type`. Sólo los atributos `href` y `rel` son obligatorios.

El atributo `href` simplemente indica la URI que tenemos que seguir.

El atributo `rel` es realmente importante. En este atributo se especifica el significado de seguir ese link. Son los posibles valores que puede tomar `rel` dentro de nuestro sistema lo que tendremos que documentar con detalle. Ya existen algunos valores estandarizados[2], pero en este caso se ha optado por usar valores propietarios del sistema, indicándolo mediante el prefijo `http://www.shop.com/rels`. Se ha usado una URL y no una URN, porque el significada de cada tipo de enlace debe ser

[2]Algunos valores del atributo "rel" estándarizados: http://www.iana.org/assignments/link-relations/link-relations.xml

documentado. Al usar una URL podemos indicar no sólo el tipo de enlace, sino la página web donde se encuentra la documentación sobre ese tipo de enlace. Es decir, usamos la URL no sólo como indicador de tipo, sino como página de documentación. En dicha documentación deberíamos especificar el significado exacto de la relación, y que métodos HTTP son admisibles.

En el atributo `title` se especifica una descripción para humanos de lo que representa el enlace. ¿Realmente necesitamos este atributo? No, pero es útil. Si el consumidor del recurso es una UI, podría usar este atributo para generar de forma automática una interfaz de usuario para este enlace, tal vez un botón o un enlace HTML, de la acción representada por el enlace. Por otro lado, desde el punto de vista de documentación es muy útil al desarrollador para averiguar para que puede servir dicho enlace. De esta forma, de un sólo vistazo, el desarrollador puede tener una comprensión básica de lo que hace el enlace. Otro tema es el de la internacionalización. Si se considera que es el servidor el encargado de gestionar este aspecto, tiene sentido que el título se presente en el idioma más adecuado posible.

Finalmente algunos autores usan un cuarto atributo, `type` que indica el tipo mime que dicho enlace soporta. Esto sólo tiene sentido si la operación que representa el enlace sobre el recurso apuntado sólo soporta un tipo mime. Una variante sería especificar el un *media range* como vimos en capítulos anteriores. El autor considera que este atributo normalmente no es necesario, con lo que no lo incluirá en los ejemplos.

Para resumir, en el enfoque *hypermedia* la interfaz de nuestra API, que es lo que tenemos que documentar, consta de:

- Una URI principal de entrada a nuestro sistema.
- Los tipos MIME que vamos a usar.
- Una serie de convenciones sobre cómo consumir y representar dentro de cada documento, las acciones disponibles en el recurso que estamos accediendo.
- La forma más extendida de hacer lo anterior es mediante enlaces tipados mediante el atributo `rel`. Habría que documentar pues lo que significa cada tipo de enlace y cómo consumirlo.

Veamos todo esto en más detalle.

6.3 Consultas autodescubribles y URI Templates

Lo más sencillo de modelar son las consultas. Como se explicó antes, las consultas se realizan mediante una petición que use el método `GET` a la URI del recurso. Sin embargo, en una API *hypermedia* en vez de tener que especificar una URI para cada recurso, éstas deberían ser descubribles como enlaces dentro de otros recursos.

Supongamos el ejemplo anterior de la tienda, si seguimos el enlace marcado como `rels/products`, llegamos al siguiente documento:

```
HTTP/1.1 200 Ok
Content-Type: application/x-shop.collection+json

{
  "links": [
    {
      "href": "/shop/products",
      "title": "Product catalog",
      "rel": "self"
    },
    {
      "href": "/shop/basket/234255",
      "title": "My shopping list",
      "rel": "http://www.shop.com/rels/basket"
    },
    {
      "href": "/shop/books{?pagesize}",
      "title": "Books available in the shop",
      "rel": "http://www.shop.com/rels/products/search/books",
      "template": true
    },
    {
      "href": "/shop/offers{?pagesize}",
      "title": "Special offers",
      "rel": "http://www.shop.com/rels/products/search/offers",
      "template": true
    }
    {
      "href": "/shop/products{?maxprice,minprice,description,pagesize}",
      "title": "Search products",
      "rel": "http://www.shop.com/rels/search",
      "template": true
    }
  ],
  //... other data ...
}
```

En este documento existen enlaces a algunas consultas frecuentes, como `rels/products/search/books`, o `rels/products/search/offers`. Basta con hacer un `GET` a la URI de dichos enlaces para acceder a las consultas de "libros disponibles en la tienda" y "productos de oferta".

Un tipo de enlace curioso es `rel="self"`, que es estándar y que indica la URL del recurso en si. Es útil cuando encontramos el recurso embebido dentro de otro recurso, y queremos saber su URI. No

es estrictamente necesario para el recurso que estamos pidiendo, pero por homogeneidad y sencillez se incluye también.

Si nos fijamos algunas de las URIs son algo raras. En este caso no se tratan URIs en sí, sino de *URIs Template*[3]. Para indicar si un enlace o no es una *URI Template* se usa el atributo `template` con el valor `true`. Las *URIs Template* nos permiten averiguar las reglas según las cuales podemos construir una URI en función a los valores de una serie de parámetros. En este caso concreto usamos una *URI template* para saber que parámetros de consulta están disponibles y cómo construir una URI de consulta a partir de éstos. Actualmente se está trabajando en la estandarización de las *URI Template*[^9], y si usamos una de las múltiples implementaciones que existen de este mecanismo[4], podemos aprovecharnos de este incipiente estándar sin apenas esfuerzo.

Un caso de uso de *URI Template* lo constituye los enlace de tipo `/rels/products/search`, este tipo de enlace representan una consulta genérica que admite parámetros. En el caso del ejemplo, la *URI template* es `/shop/products{?maxprice,minprice,description,pagesize}`, con lo cual podremos construir una consulta como la siguiente `/shop/products?maxprice=100&description=vampires`. Las *URI Template* tienen una sintaxis muy rica que nos permiten una gran flexibilidad a la hora de definir nuestro esquema de URIs, para más detalles ver [^9]. De esta forma podemos generar la URI que necesitamos para realizar una acción en función de los parámetros que nos interesen.

En el caso de las consultas definidas en `rels/products/search/books`, o `rels/products/search/offers` podemos especificar el parámetro `pagesize` para indicar cuantos resultados queremos en cada página. En cualquier *URI Template* los parámetros son opcionales, de forma que si el cliente no los especifica, ya sea porque no le interesa o no los entiende, el parámetro no está presenta en la URI resultante.

Supongamos que seguimos el enlace para buscar las ofertas (`rel="rels/products/search/offers"`), y no especificamos el parámetro `pagesize`. El servidor escoge en ese caso un tamaño de paginación por defecto de 10. El documento que nos devolvería sería el siguiente:

```
HTTP/1.1 200 Ok
Content-Type: application/x-shop.collection+json

{
  "links": [
    {
      "href": "/shop/offers",
      "title": "Special offers (1st page)",
      "rel": "self"
    },
    {
      "href": "/shop/basket/234255",
      "title": "My shopping list",
```

[3]Especificación de URI Templates: http://tools.ietf.org/html/rfc6570

[4]Algunas implementaciones de URI Templates: http://code.google.com/p/uri-templates/wiki/Implementations

```
      "rel": "http://www.shop.com/rels/basket"
    },
    {
      "href": "/shop/products{?pagesize}",
      "title": "Product catalog",
      "rel": "http://www.shop.com/rels/products",
      "template": true
    },
    {
      "href": "/shop/books{?pagesize}",
      "title": "Search books available in the shop",
      "rel": "http://www.shop.com/rels/products/search/books",
      "template": true
    },
    {
      "href": "/shop/offers/page/2?pasesize=10",
      "title": "Next page",
      "rel": "next"
    },
    {
      "href": "/shop/offers/page/{n}?pagesize=10",
      "title": "Go to page",
      "rel": "http://www.shop.com/rels/goto",
      "template": true
    },
    {
      "href": "/shop/offers{?orderby}",
      "title": "Offers ordered by",
      "rel": "http://www.shop.com/rels/orderby",
      "template": true
    }
  ],
  "items": [
    {
      "links": [
        {
          "href": "/shop/products/43",
          "title": "Fantastic book",
          "rel": "self"
        },
        {
          "href": "/shop/products/43/orderform?basket=234255",
```

```
          "title": "Add this product to my shopping list",
          "rel": "http://www.shop.com/rels/form"
        }
      ],
      "author": "Enrique Gómez Salas",
      "title": "Desventuras de un informático en Paris",
      "genre": "scifi",
      "price": { currency:"€", amount:5}
    },
    //.... Más productos, hasta completar 10 (tamaño de la página) ...
  ]
}
```

Obsérvese que en vez de devolver una lista de productos, el servidor devuelve una página de resultados. En esta página de resultados podemos ver que tenemos más enlaces: a la siguiente página (`rel="next"`), saltar directamente a una página (`rel="rels/goto"`), o cambiar el orden en el que se nos presentan los datos (`rel="rels/orderby"`). Además cada entrada de resultados viene anotado con sus propios enlaces, que nos permiten realizar acciones sobre éstos.

Si seguimos el enlace (`rel="next"`) se nos presenta el siguiente documento:

```
HTTP/1.1 200 Ok
Content-Type: application/x-shop.collection+json

{
  "links": [
    {
      "href": "/shop/offers/page/2",
      "title": "Special offers (2nd page)",
      "rel": "self"
    },
    {
      "href": "/shop/basket/234255",
      "title": "My shopping list",
      "rel": "http://www.shop.com/rels/basket"
    },
    {
      "href": "/shop/products{?pagesize}",
      "title": "Product catalog",
      "rel": "http://www.shop.com/rels/products",
      "template": true
    },
    {
```

```
      "href": "/shop/books{?pagesize}",
      "title": "Search books available in the shop",
      "rel": "http://www.shop.com/rels/products/search/books",
      "template": true
    },
    {
      "href": "/shop/offers",
      "title": "First page",
      "rel": "first"
    },
    {
      "href": "/shop/offers",
      "title": "Previous page",
      "rel": "prev"
    },
    {
      "href": "/shop/offers/page/3?pasesize=10",
      "title": "Next page",
      "rel": "next"
    },
    {
      "href": "/shop/offers/page/{n}?pagesize=10",
      "title": "Go to page",
      "rel": "http://www.shop.com/rels/goto",
      "template": true
    },
    {
      "href": "/shop/offers{?orderby}",
      "title": "Offers ordered by",
      "rel": "http://www.shop.com/rels/orderby",
      "template": true
    }
  ],
  "items": [
   // .... Más productos, hasta completar 10 (tamaño de la página) ...
  ]
}
```

En este documento aparecen nuevos enlaces, como la posibilidad de volver atrás (`rel="prev"`) o a la primera página (`rel="first"`). Estos enlaces no aparecían en el anterior documento, ya que éste representaba la primera página de resultados. Esto es un detalle importante, ya que en una API *hypermedia* no se deben incluir enlaces que representen acciones no admisibles por el estado actual

del sistema. Únicamente aparecerán los enlaces para las acciones que sean consistentes con el estado actual.

Al usar *hypermedia* las consultas y la paginación están integradas. Los resultados de las búsquedas son más que meros contenedores pasivos de datos, sino que incluyen posibles acciones sobre éstos en forma de links. Por lo tanto es prácticamente gratis añadir enlaces que soporten un modelo de consultas paginadas.

6.4 Controles hypermedia

Parece que la parte de consulta de nuestro sistema es bastante intuitiva, pero, ¿cómo modelamos acciones que puedan cambiar el estado del sistema? Es simple, por cada operación consistente con el estado actual del sistema, debemos incluir un "control" que nos permita realizarla. Sólo debemos inspeccionar el documento que representa un recurso en busca de controles para averiguar que operaciones podemos realizar sobre un recurso. Qué controles están presente y cuales no varían tanto con el tipo del recurso como con el estado concreto del sistema. Para un mismo recurso algunos controles pueden estar disponibles, o simplemente no aparecer en el documento, en función de si su ejecución es compatible con el estado del sistema. Por lo tanto si una acción no puede ser ejecutada actualmente, el control correspondiente no se incluye en el documento.

A continuación se hace una introducción a como implementar controles en una API *hypermedia*.

Acciones como enlaces

La forma más sencilla de control es el enlace. Ya hemos visto estos enlaces en acción cuando se explicó las consultas, pero ahora nos ocuparemos de como modelar otras operaciones usando enlaces.

Un enlace, además de con el método `GET`, puede ser seguido usando un verbo HTTP que represente un cambio de estado, como `PUT`, `POST` y `DELETE`. Qué verbo usar, y qué efecto se produce, debería estar documentado para cada tipo de enlace. En cualquier caso yo recomiendo respetar la semántica original de los verbos HTTP[5], para ello podemos seguir las indicaciones dadas en capítulos anteriores sobre cuando usar cada verbo.

Sin embargo, por claridad e interoperabilidad, se suele añadir un atributo `rel` a los enlaces, indicando cuál es la semántica de seguir ese enlace. En nuestra API deberemos documentar cada valor que puede aparecer en `rel`, indicando que significa seguir un enlace marcado como tal y que verbos HTTP se soportan.

El diseño de APIs *hypermedia* trabaja especialmente bien con la técnica explicada en el capítulo anterior de "URIs desechables". En el ejemplo anterior se observa que el resultado incluye un enlace de tipo `rels/basket`. Si consultamos la documentación sobre este tipo de enlaces, vemos que el primero representa la cesta de la compra del usuario. Se puede consultar con `GET`, y sobreescribir por completo con `PUT`. Veamos que ocurre si hacemos `GET` sobre la cesta:

[5] Aclaración sobre la semántica de los métodos HTTP: http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-18

```
HTTP/1.1 200 Ok
Content-Type: application/x-shop.basket+json

{
  "links": [
    {
      "href": "/shop/basket/234255",
      "title": "My shopping list",
      "rel": "self"
    },
    {
      "href": "/shop/basket/234255/items",
      "title": "Products in the shopping list",
      "rel": "http://www.shop.com/rels/items"
    },
    {
      "href": "/shop/products{?pagesize}",
      "title": "Product catalog",
      "rel": "http://www.shop.com/rels/products",
      "template": true
    },
    {
      "href": "/shop/books{?pagesize}",
      "title": "Search books available in the shop",
      "rel": "http://www.shop.com/rels/products/search/books",
      "template": true
    },
    {
      "href": "/shop/offers{?pagesize}",
      "title": "Special offers",
      "rel": "http://www.shop.com/rels/products/search/offers",
      "template": true
    }
  ],
  "items": [],
  "total": { currency: "€", amount: 0 }
}
```

Obtenemos una cesta vacía. No hemos tenido que crear la cesta de ninguna forma. Es un caso claro de uso de "URIs desechables". Mientras no rellenemos la cesta, esta seguirá siendo un recurso vacío y no necesitaríamos almacenarla de forma persistente ya que su representación se puede generar desde cero.

Para añadir productos a la cesta podemos usar el enlace con `rel="/rels/items"`. La documentación nos indica que hagamos un `POST` para crear una linea de pedido y que después hagamos un `PUT` indicando el identificador de producto y la cantidad.

Primero pedimos abrir una nueva línea de pedido en la cesta de la compra:

```
POST /shop/basket/234255/items HTTP/1.1
Host: www.shop.com
Accept: application/json

```

Y la respuesta:

```
HTTP/1.1 201 Created
Location: http://www.shop.com/shop/basket/234255/items/ecafadfj312_dad0348

```

Ahora confirmar la línea de pedido, indicando producto y cantidad

```
PUT /shop/basket/234255/items/ecafadfj312_dad0348 HTTP/1.1
Host: www.shop.com
Accept: application/json
Content-Type: application/x-shop.orderitem+json

{
  "quantity": 2,
  "product-ref": "http://www.shop.com/shop/products/43"
}
```

Y la respuesta del servidor, con la linea de pedido creada:

```
HTTP/1.1 200 OK
Content-Type: application/x-shop.orderitem+json;charset=utf-8

{
  "links": [
    {
      "href": "/shop/basket/234255/items/ecafadfj312_dad0348",
      "title": "this order line",
      "rel": "self"
    },
    {
```

```
      "href": "/shop/basket/234255",
      "title": "My shopping list",
      "rel": "http://www.shop.com/rels/basket"
    }
  ],
  "quantity": 2,
  "product-ref": "http://www.shop.com/shop/products/43"
}
```

Para ver como queda la lista de la compra seguimos el enlace de tipo `rel="/rels/basket"`:

```
HTTP/1.1 200 Ok
Content-Type: application/x-shop.basket+json

{
  "links": [
    {
      "href": "/shop/basket/234255",
      "title": "My shopping list",
      "rel": "self"
    },
    {
      "href": "/shop/basket/234255/items",
      "title": "Products in the shopping list",
      "rel": "http://www.shop.com/rels/items"
    },
    {
      "href": "/shop/basket/234255/payment",
      "title": "Products in the shopping list",
      "rel": "http://www.shop.com/rels/basket/payment"
    },
    {
      "href": "/shop/products{?pagesize}",
      "title": "Product catalog",
      "rel": "http://www.shop.com/rels/products",
      "template": true
    },
    {
      "href": "/shop/books{?pagesize}",
      "title": "Search books available in the shop",
      "rel": "http://www.shop.com/rels/products/search/books",
      "template": true
```

```
    },
    {
      "href": "/shop/offers{?pagesize}",
      "title": "Special offers",
      "rel": "http://www.shop.com/rels/products/search/offers",
      "template": true
    }
  ],
  "items": [
    {
      "links": [
        {
          "href": "/shop/basket/234255/items/ecafadfj312_dad0348",
          "title": "this order line",
          "rel": "self"
        },
        {
          "href": "/shop/basket/234255",
          "title": "My shopping list",
          "rel": "http://www.shop.com/rels/basket"
        }
      ],
      "quantity": 2,
      "product-ref": "http://www.shop.com/shop/products/43"
    }
  ],
  "total": { currency: "€", amount: 10 }
}
```

Como veis hemos añadido 2 ejemplares de un libro. Lo más importante es que ha aparecido un enlace de tipo `rels/payment` que nos permitirá comprar. Más adelante veremos cómo. Si quisiéramos editar la línea de pedido basta con hacer `PUT` o `PATCH` a la URL de la línea. No es necesario memorizar dicha URL ya que viene siempre en los enlaces tipo `rel="self"`. Si queremos anular la línea de pedido es tan simple como hacer `DELETE`.

Ahora sólo nos queda pagar. Tan sencillo como hacer `PUT` sobre la URL especificada en el enlace "rels/basket/payment" con la información pertinente:

```
PUT /shop/basket/234255/payment HTTP/1.1
Host: www.shop.com
Accept: application/x-shop.payment+json
Content-Type: application/x-shop.payment+json

{
  "method":"visa",
  "card": {
    "cardnumber": 1234567812345678,
    "secnumber":333,
    "expiration": "08/2016",
    "cardholder": "Pepe Pérez",
  }
}
```

La respuesta:

```
HTTP/1.1 200 Ok
Date: Tue, 27 Dec 2012 05:25:19 GMT
Expires: Tue, 27 Dec 2012 05:25:39 GMT
Cache-Control: max-age=20
Content-Type: application/x-shop.payment+json

{
  "links":[
    {
      "href": "/shop/basket/234255/payment",
      "title": "Payment",
      "rel": "self"
    },
    {
      "href": "/shop/basket/234255",
      "title": "My shopping list",
      "rel": "http://www.shop.com/rels/basket"
    }
  ],
  "method":"visa",
  "card": {
    "cardnumber": 1234567812345678,
    "secnumber":333,
    "expiration": "08/2016",
    "cardholder": "Pepe Pérez",
```

```
  },
  "status":"in progress"
}
```

El pago ha sido aceptado, pero todavía no esta confirmado. A partir de ahí es cuestión de hacer polling sobre el recurso hasta que el pago finalize. Podemos usar la cabecera `Cache-Control` para saber con que frecuencia deberíamos hacer polling. En este caso cada 20 segundos. Finalmente cuando el pago se confirma:

```
HTTP/1.1 200 Ok
Date: Tue, 27 Dec 2012 05:30:00 GMT
Expires: Tue, 27 Dec 2012 11:30:00 GMT
Cache-Control: max-age=21600
Content-Type: application/x-shop.payment+json

{
  "links":[
    {
      "href": "/shop/basket/234255/payment",
      "title": "Payment",
      "rel": "self"
    },
    {
      "href": "/shop/basket/234255",
      "title": "My shopping list",
      "rel": "http://www.shop.com/rels/basket"
    },
    {
      "href": "/shop/basket/234255/invoice",
      "title": "Invoice",
      "rel": "http://www.shop.com/rels/invoice"
    }
  ],
  "method":"visa",
  "card": {
    "cardnumber": 1234567812345678,
    "secnumber":333,
    "expiration": "08/2016",
    "cardholder": "Pepe Pérez",
  },
  "status":"confirmed"
}
```

Observemos que ahora aparece un enlace "/rels/invoice" para obtener la factura. Este es un caso típico en el que podemos usar varios tipos mime, en función del formato en el que el cliente quiera la factura: PDF, HTML, DOC, etc.

Obviamente estoy simplificando. En realidad el workflow de pago es probablemente mucho más complejo, y por supuesto cada petición debería ir protegida con las técnicas explicadas en las secciones anteriores (al menos HTTPS con autenticación digest).

Acciones como formularios

El uso de enlaces como controles está bien para acciones simples, normalmente de estilo CRUD y que sólo requieran parámetros sencillos que puedan ser enviados mediante la *query string*. Como nivel de complejidad máximo podemos usar *URI Templates*.

Sin embargo para operaciones más complejas, y que normalmente necesitan enviar un documento en el cuerpo de la petición HTTP es más apropiado usar formularios.

Otra forma de añadir un producto al carrito hubiera sido usar el enlace tipo `rel="/rels/form"`. Recordemos el producto 43 tenía la siguiente información:

```
HTTP/1.1 200 Ok
Content-Type: application/x-shop.product+json

{
  "links": [
    {
      "href": "/shop/products/43",
      "title": "Fantastic book",
      "rel": "self"
    },
    {
      "href": "/shop/products/43/orderform?basket=234255&product=43",
      "title": "Add this product to my shopping list",
      "rel": "http://www.shop.com/rels/form"
    }
  ],
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
  "genre": "scifi",
  "price": { currency:"€", amount:5}
}
```

Si seguimos el enlace `rel="/rels/form"` usando `GET`, obtenemos:

```
HTTP/1.1 200 Ok
Content-Type: application/x-shop.form+json

{
  "links": [
    {
      "href": "/shop/products/43/orderform?basket=234255&product=43",
      "title": "Order product form",
      "rel": "self"
    },
    {
      "href": "/shop/basket/234255/items/ecafadfj312_dad0348",
      "title": "Order product form target",
      "rel": "http://www.shop.com/rels/target"
    },
    {
      "href": "/shop/basket/234255",
      "title": "My shopping list",
      "rel": "http://www.shop.com/rels/basket"
    },
    {
      "href": "/shop/products/43",
      "title": "Product to add",
      "rel": "http://www.shop.com/rels/product"
    }
  ],
  "method": "PUT",
  "type": "application/x-shop.orderitem+json",
  "body": {
    "quantity": 1,
    "product-ref": "http://www.shop.com/shop/products/43"
  }
}
```

El documento resultante no es más que la versión JSON de un formulario ya relleno con información. Veamos de que consta un formulario JSON.

- Dentro de `links` el enlace de tipo `rel="/rels/target"` nos indica cual es la URL sobre la que debemos la realizar la petición para procesar este "formulario".
- El campo `method` nos indica el método HTTP a utilizar cuando hagamos la petición a la URL definida en `rel="/rels/target"`
- El campo `type` nos indica el tipo mime que debe tener el cuerpo de la petición. También podríamos indicar aquí un *media range*

- El campo `body` nos proporciona una instancia de datos que podemos usar en el cuerpo de la petición.

En este caso el campo `body` y el link `target` aparecen ya informados. De esta forma simplemente sería hacer un `PUT` a la URL que nos indican con los datos que aparecen en el `body`. Esto tendría como resultado añadir un ejemplar del producto "43" a la cesta de la compra, tal y como hicimos en el ejemplo de la sección anterior. Hemos evitado tener que usar un `POST` gracias a que la URI apuntada por `target` es desechable. Si pidiéramos dicha URI con GET nos aparecería una linea de pedido vacía.

El uso de formularios es un poco más avanzado que el uso de links sencillos. Pero a la vez nos permite más potencia. En este ejemplo tenemos dos ventajas:

- Nos ahorramos hacer un `POST` a la cesta de la compra para conseguir crear una nueva linea de pedido "virtual". En el propio formulario nos viene la URL de una nueva linea de pedido.
- Los datos a informar en la petición `PUT` ya nos vienen completados desde el servidor.

En general el uso de la técnica de formularios nos simplifica el uso de URIs desechables, ya que no necesitamos en ningún momento hacer un `POST`. También nos ayudan a autodescribir mejor nuestra API.

Veamos un ejemplo más, supongamos que para tomar partido de esta técnica ahora el carrito de la compra es como sigue:

```
HTTP/1.1 200 Ok
Content-Type: application/x-shop.basket+json

{
  "links": [
    {
      "href": "/shop/basket/234255",
      "title": "My shopping list",
      "rel": "self"
    },
    {
      "href": "#orderform",
      "title": "Add a product to this shopping list",
      "rel": "http://www.shop.com/rels/form"
    },
    {
      "href": "/shop/products{?pagesize}",
      "title": "Product catalog",
      "rel": "http://www.shop.com/rels/products",
```

```
      "template": true
    },
    {
      "href": "/shop/books{?pagesize}",
      "title": "Search books available in the shop",
      "rel": "http://www.shop.com/rels/products/search/books",
      "template": true
    },
    {
      "href": "/shop/offers{?pagesize}",
      "title": "Special offers",
      "rel": "http://www.shop.com/rels/products/search/offers",
      "template": true
    }
  ],
  "items": [],
  "total": { currency: "€", amount: 0 },
  "orderform": {
    "links": [
      {
        "href": "/shop/basket/234255#orderform",
        "title": "Order product form",
        "rel": "self"
      },
      {
        "href": "/shop/basket/234255/items/ecafadfj312_dad0348",
        "title": "Order product form target",
        "rel": "http://www.shop.com/rels/target"
      }
    ],
    "method": "PUT",
    "type": "application/x-shop.orderitem+json"
  }
}
```

Ahora añadir productos al carrito es mucho más sencillo que antes. No necesitamos hacer un `GET` a la colección de "items", para después hacer un `POST` y conseguir una URL desechable. Simplemente basta con rellenar el formulario incrustado en el carrito y ejecutarlo. Además de que el procesamiento es más sencillo, ahora nos ahorramos dos llamadas de red.

En este caso el formulario no tiene campo `body`, por lo que el cliente deberá indicar toda la información. Afortunadamente el campo `type` nos indica el tipo mime apropiado, cuya documentación debemos consultar para saber como implementar el cliente.

6.5 Web Linking

El problema

Hasta ahora se ha supuesto que el tipo mime usado en la comunicación REST tiene capacidad para modelar enlaces y controles web. Algunos tipos como XML o HTML lo permiten de forma estandarizada, otros como JSON no tienen una noción estandarizada de enlaces y controles web, pero ciertamente tienen la capacidad de modelar tales artefactos.

Sin embargo existen tipos mime, como los destinado a audio e imágenes, que no nos permiten ni insertar enlaces ni controles web dentro de ellos. ¿Debemos renunciar al hypermedia en estos casos?

Para solventar este problema Mark Nottingham inpulsó la creación de un estandar: Web Linking[6]

La propuesta

Esta propuesta de estándar se basa en crear una nueva cabecera HTTP, la cabecera `Link`. Dentro de esta cabecera se indican todos los enlaces activos para el recurso. De esta forma si un tipo mime no soporta de forma nativa los enlaces (imágenes, audio, JSON...), dichos enlaces se pueden introducir dentro de la cebecera `Link`. En esta especificación se define que la cabecera `Link` tiene una semántica equivalente al elemento `<link>` de HTML y de ATOM. Este estándar viene acompañado de un registro estandarizado de tipos de enlaces[7], es decir, los valores que el atributo `rel` puede tomar.

¿Cómo debemos representar los enlaces dentro de la cabecer `Link`? La propuesta es encerrar la URI a la que apunta el enlace entre los caracteres `"<"` y `">"`, y concatenar los parámetros usando el caracter `";"`. Por ejemplo: `Link: </shop/basket/234255/items/ecafadfj312_dad0348> rel="/rels/target"; title="Order product form target"`

El estandar, aunque deja abierta la posibilidad a otros parámetros, define explicitamente los siguientes:

- `rel`. Para el tipo de enlace, como hemos visto hasta ahora.
- `rev`. Para el tipo simétrico del enlace. Algunos enlaces pueden ser bidireccionales y necesitamos modelar ambos lados de la relación.
- `anchor`. Por defecto, si la URI del enlace es relativa, esta se resuelve usando la URI del recurso actual como base. El parámetro `anchor` nos permite cambiar este comportamiento, y permitir que la URI del enlace se resuelva usando el valor de este parámetro como base.
- `type`. Cuando se encuentra presente indica el tipo mime soportado por el recurso destino. Sólo puede estar presente una vez. De esta forma el cliente pude decidir no consumir el enlace si no soporta dicho tipo mime.

[6] Web Linking standard: http://tools.ietf.org/html/rfc5988

[7] http://www.iana.org/assignments/link-relations/link-relations.xml

- `media`. Cuando se encuentra presente indica las características de presentación para el cual el recurso destino está diseñado. Esto se especifica mediante una *media query*[8]. La idea es que algunos recursos pueden estar pensados sólo para algunos medios de presentación (impresoras, pantallas, determinadas resoluciones, etc.). El cliente puede decidir no consumir el enlace si considera que el dispositivo en el que se está ejecutando no soporta dichas características. Este tipo de información sólo tiene sentido si el recurso que estamos enlazando tiene un uso relativo a la presentación de información al usuario, tales como imágenes, video, PDFs, CSSs, etc. Por ejemplo podríamos tener varios enlaces con distintos `media`, apuntando a una imagen de un mismo gráfico pero a distintas resoluciones.
- `hreflang`. Cuando se encuentra presente indica el lenguaje soportado por el recurso destino. Pueden existir múltiples pares clave valor de este parámetro indicando que el recurso destino soporta varios lenguajes. De esta forma el cliente pude decidir no consumir el enlace si no soporta ningún lenguaje que el usuario entienda.
- `title`. El título del enlace tal y como hemos visto hasta ahora.
- `title*`. Nos permite codificar el valor del título en caso de que este necesite caracteres que no estén soportados en `UTF-8`. Ejemplo: `title*=UTF-8'de'n%c3%a4chstes%20Kapitel`

Una misma cabecera `Link` puede contener varios enlaces separados por el caracter `","`. Supongamos el ejemplo anterior relativo al carrito de la compra que contien un formulario para realizar el pedido. Si usáramos *web linking* la respuesta quedaría:

```
HTTP/1.1 200 Ok
Content-Type: application/x-shop.basket+json
Link:<./orderform?basket=234255>;
     rel="/rels/form";title="Add a product",
     </shop/products{?pagesize}>;
     rel="/rels/products";template=true;title="Product catalog",
     </shop/books{?pagesize}>;
     rel="/rels/products/search/books";template=true;title="Search books",
     </shop/offers{?pagesize}>;
     rel="/rels/products/search/offers";template=true;title="Offers"

{
  "items": [],
  "total": { currency: "€", amount: 0 },
}
```

En este ejemplo se han eliminado los enlaces del cuerpo de la respuesta y se han definido en la cabecera `Link`. También se ha definido el formularo en un recurso aparte, en vez de tenerlo incrustado en el carrito.

[8] http://www.w3.org/TR/css3-mediaqueries/

La controversia

El enfoque de web linking tiene algunas ventajas obvias:

- Podemos usar tipos mimes que no soportan enlaces
- Si usamos tipos mimes que podrían soportarlo, pero no lo hacen de forma estándar, como es el caso de JSON, no necesitamos inventar una forma propietaria de representarlos.
- Podemos usar el método `HEAD` para saber que enlaces tiene disponible un recurso de forma eficiente. De esta forma no necesitamos transmitir una representación del recurso, ya que las respuestas a peticiones `HEAD` sólo deben contener las cabeceras, y no el cuerpo. Esto nos permite navegar por la web de recursos de forma más eficiente, y hacer un `GET` sólo en aquellos casos que lo necesitemos.

El problema, y de ahí la controversia, es que añade un grado extra de complejidad. Algunos sostienen que este grado de complejidad es innecesaria. Por ejemplo, ¿que JSON no soporta enlaces de forma estándar? Pues simplemente estandaricemos como deben modelarse, no hay necesidad de una nueva cabecera. ¿Qué no podemos poner enlaces en una imagen? Pues simplemente no lo hagamos, nos basta con referenciar a esta imagen desde un recurso de metainformación que use un tipo mime como XML, HTML o JSON. Este es al fin y al cabo el enfoque de ATOM y ha funcionado bien hasta el momento. Otra solución es empotrar el contenido multimedia usando base 64 en un documento XML por ejemplo.

Otra crítica es que añade una nueva forma de crear enlaces, es decir tenemos dos maneras diferentes de hacer la misma cosa. Ahora no sólo debemos mirar el cuerpo de la respuesta en busca de enlaces, sino las cabeceras `Link`. Aparte de complicar la implementación de un posible cliente, debemos lidiar con la posibilidad de que los enlaces en el cuerpo y en la cabecera `Link` se contradigan entre si.

A nivel conceptual también existe un debate sobre si definir los enlaces a nivel de cabecera HTTP es adecuado o no. Algunos piensan que las cabeceras HTTP deben incluir metainformación sobre el mensaje HTTP y aspectos de gestión de las comunicaciones. De esta forma se podría argumentar que no se debe incluir información del recurso en si dentro de las cabeceras HTTP.

El autor considera que en el caso de que el recurso no soporte de ninguna manera enlaces, el uso de web linking es muy apropiado. En otro caso la decisión debería tomarse en función de las circunstancias específicas de la API. En cualquier caso, si se va a usar web linking con un formato que ya soporta enlaces, hay que asegurarse de que dichos enlaces sean coherentes con los presentados en la cabecera `Link`.

6.6 Patrón *Envelope*

Como ya vimos en la sección de *web linking*, el hecho de que existan tipos mime que no soporten ni enlaces ni controles *hypermedia* es un auténtico problema. Ya vimos que actualmente existen dos soluciones "de moda": *web linking* y extender tipos mimes ya existentes para que soporten

hypermedia. Sin embargo, existe una tercera forma de solucionar este problema: usar recursos *envelopes*. Quizás no estén tan de moda pero a veces pueden ser justo lo que necesitamos.

Implementación básica

La idea es sencilla, en vez de modelar en la API un único recurso que no tenga capacidad para *hypermedia*, por ejemplo una imagen, lo que modelaremos en nuestra API sera una pareja de recursos: el *envelope* y el recurso *desnudo*, cada una con su propia URI. El recurso *desnudo* es el recurso que realmente nos interesa, la imagen, video o cualquier otro formato que no tenga *hypermedia*. El *envelope* es un recurso adicional, implementado en un formato *hypermedia* (como los que veremos en posteriores capítulos), y que contiene los enlaces y controles *hypermedia* relativos al recurso desnudo.

Para crear tal recurso, basta con hacer `POST` o `PUT`, tal y como se explicó en capítulos anteriores, y enviar los datos del recurso *desnudo*. Por ejemplo:

```
POST /user/john/pics HTTP/1.1
Host: youpic.com
Content-Type: application/png

...Binary data here....
```

Y el servidor respondería con:

```
HTTP/1.1 201 Created
Content-Type: application/xml
Location: http://youpic.com/user/john/pics/523

```

Obsérvese que la cabecera `Location` apunta a un recurso que no es una imagen, si hacemos `GET` sobre la URI contenida en dicha cabecera obtenemos:

```
<?xml version="1.0" encoding='utf-8'?>
<envelope
  xmlns="http://youpic.com/schemas"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://youpic.com/schemas
                      http://youpic.com/schemas/envelope.xsd">
  <link rel="self"
        href="http://youpic.com/user/john/pics/523"/>
  <link rel="alternate" type="image/png"
```

```
10          href="http://youpic.com/user/john/pics/523.png"/>
11    <link rel="alternate" type="image/jpg"
12          href="http://youpic.com/user/john/pics/523.jpg"/>
13    <link rel="next"
14          href="http://youpic.com/user/john/pics/12d4"/>
15    <link rel="prev"
16          href="http://youpic.com/user/john/pics/X2c22"/>
17  </envelope>
```

Este nuevo documento es el *envelope*. Es decir, cuando creamos un recurso *desnudo*, el servidor crea un *envelope* y nos informa de la URI de este en la cabecera `Location` de la respuesta. ¿Cómo accedemos entonces a los datos del recurso *desnudo*? El *envelope* debe contener al menos un enlace que nos permita acceder y operar con el recurso desnudo. Dicha URI podría admitir las operaciones CRUD sobre el recurso *desnudo*, pero nada más. Si queremos modelar más operaciones, debemos añadir enlaces y controles *hypermedia* adicionales al *envelope*.

En el ejemplo anterior tenemos varios enlaces: `alternate`, `next` y `prev`. En este caso `alternate` es el enlace que apunta al recurso *desnudo*. Obsérvese que podríamos tener varios enlaces `alternate`, cada uno apuntando a una URI diferente, en función del tipo mime que queramos recibir. Los enlaces `next` y `prev` nos permiten navegar al siguiente y al anterior *envelope* respectivamente. Sería un error que `next` y `prev` apunten directamente a un recurso *desnudo*, ya que este no tiene enlaces, y no sabríamos seguir navegando por la API. Por lo tanto sólo se debe referenciar al recurso *desnudo* desde el enlace de contenido de un *envelope*.

En el caso de que decidiéramos borrar el recurso *desnudo* enviando un `DELETE` a la URI de este, el servidor debe borrar el *envelope* también. Después de todo un *envelope* sin recurso *desnudo* no tiene mucho sentido. Sin embargo si borramos el *envelope*, ¿qué debe hacer el servidor? Hay varias opciones:

- Borrar el recurso *desnudo* también, para garantizar la consistencia, pero sólo si es el único *envelope* que apunta a dicho recurso *desnudo*.
- No borrar el recurso *desnudo*, y permitir al cliente la opción de crear un nuevo *envelope* o modificar uno existente para que apunte al recurso *desnudo*.

Recurso *desnudo* incrustado

Una variante de implementación es proveer la capacidad de incrustar la información dentro del envelope. De esta manera no necesitaríamos hacer un segundo `GET` para obtener la información del recurso. Como normalmente el recurso *desnudo* suele ser binario, se codifica la información de este en base 64. Por ejemplo:

```xml
<?xml version="1.0" encoding='utf-8'?>
<envelope
  xmlns="http://youpic.com/schemas"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://youpic.com/schemas
                      http://youpic.com/schemas/envelope.xsd">
  <link rel="self"
        href="http://youpic.com/user/john/pics/523"/>
  <link rel="alternate" type="image/png"
        href="http://youpic.com/user/john/pics/523.png"/>
  <link rel="alternate" type="image/jpg"
        href="http://youpic.com/user/john/pics/523.jpg"/>
  <link rel="next"
        href="http://youpic.com/user/john/pics/12d4"/>
  <link rel="prev"
        href="http://youpic.com/user/john/pics/X2c22"/>
  <content type="image/png">
    ...Base 64 encoded binary data...
  </content>
</envelope>
```

En este caso hemos inventado una etiqueta `content` opcional que nos permite incrustar una copia de los datos del recurso en base 64. El cliente decodifica el base 64 para obtener una imagen binaria del recurso *desnudo* y a continuación usa el tipo mime especificado en `type` para interpretar los datos binarios.

Aprovechando la capacidad multimedia

Si podemos pedir el *envelope* en formato XML, ¿por qué no podriamos pedirlo en cualquier otro formato *hypermedia*? Esto es totalmente factible, si se desea podemos modelar el *envelope* en cualquier formato *hypermedia* que queramos: HAL, SIREN, etc.

Esta idea nos conduce a la otra variante interesante de este patrón, que es hacer que tanto el *envelope* como el recurso *desnudo* tengan exactamente la misma URI. ¿Cómo distingo entonces entre el *envelope* y el recurso *desnudo*? Por el tipo mime que pidamos. Por ejemplo, si hago un `GET` a la URI del recurso y especifico el tipo mime `image/png`, obtendré el recurso *desnudo*. Pero si hago lo mismo especificando el tipo mime `application/xml`, obtendría el *envelope*. Si usamos esta variante el *envelope* sería algo como lo siguiente:

```
<?xml version="1.0" encoding='utf-8'?>
<envelope
  xmlns="http://youpic.com/schemas"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://youpic.com/schemas
                      http://youpic.com/schemas/envelope.xsd">
  <link rel="self"
        href="http://youpic.com/user/john/pics/523"/>
  <link rel="next"
        href="http://youpic.com/user/john/pics/12d4"/>
  <link rel="prev"
        href="http://youpic.com/user/john/pics/X2c22"/>
  <content type="image/png">
    ...Base 64 encoded binary data...
  </content>
</envelope>
```

En este caso, sólo tengo enlace de tipo `self`, `next` y `prev`, no necesito `alternate`. Si quiero el recurso en jpg, pediré `image/jpg`, si pido `application/hal+json`, obtendré el *envelope* en JSON (HAL), y asi sucesivamente.

Esta variante del patrón es la que el autor recomienda, ya que soluciona varios problemas:

- Nos proporciona homogeneidad referencial: usamos siempre una misma URI tanto para el *envelope* como para el recurso *desnudo*.
- Antes la URI que apuntaba al recurso *desnudo* no nos permitía seguir navegando por la API, ya que esta no tenía enlaces.

¿Qué variante es mejor?

El lector puede detectar algunos puntos oscuros en el uso de *envelopes*:

- El recurso *desnudo* sigue sin tener capacidad para enlaces. Esto hace que si el cliente sólo dispone de la URI de dicho recurso no puede navegar ni descubrir el resto de nuestra API. Por lo tanto la URI del recurso desnudo no debería ser memorizada para futura referencia por el cliente de nuestra API, sino sólo la URI del *envelope*. Como consecuencia nuestro sistema no es completamente *hypermedia*, ya que no hay manera de obtener la URI del *envelope* o de cualquier otro recurso a partir del recurso *desnudo*.
- Nuestra API posee un nivel de indirección extra. Nos vemos forzados a manejar únicamente URIs de *envelopes*, si queremos operar sobre un recurso *desnudo*, debemos primero acceder a su *envelope* para obtener la URI.

- La ambiguedad a la hora de borrar un recurso, ¿qué debemos borrar? ¿En *envelope* o el recurso *desnudo*? ¿Qué ocurre si borro únicamente el *envelope*? Las respuestas de todas estas cuestiones son específicas de la implementación, pero el cliente debe conocerlas, con lo cual el nivel de interoperabilidad disminuye.

Sin embargo, si usamos la variante multimedia de este patrón que mostramos anteriormente solucionamos los dos primeros problemas. También podemos mejorar el rendimiento si permitimos que el *envelope* puede tener incrustada una representación de los datos binarios, ya que así evitamos una petición extra al servidor. Por lo tanto el autor recomienda usar únicamente la variante multimedia de este patrón, con una única URI que admita distintas representaciones en distintos formatos.

Hay que hacer notar que el usar este patrón no nos exime de necesitar al menos un tipo mime que tenga capacidad *hypermedia* para ser capaz de representar el *envelope*. Hasta hace relativamente poco, estábamos condenados a usar HTML, o quizás XML con links para modelar los *envelopes*. Pero hoy en día, si queremos un tipo mime más ligero, podemos recurrir a HAL, Collection+JSON o SIREN, por nombrar algunos.

Conclusión

El uso de *envelopes* está justificado en el caso de que no queramos o no podamos usar *web linking*, y que además estemos ya usando un formato *hypermedia* adecuado para modelar los *envelopes*.

Por ejemplo, si ya estamos usando Atom, HTML, HAL, SIREN o Collection+JSON, pero nuestra API necesita gestionar objetos binarias (audio, video, imágenes), podemos usar el patrón *envelope* como una alternativa a *web linking*.

Otro escenario para usar *envelope* se produce cuando realmente nuestro dominio de aplicación tiene *envelopes*. Supongamos que tenemos una imagen, es normal querer añadir información extra como el título, etiquetas, comentarios, etc. En este caso es el propio dominio de aplicación quién necesita modelar un recurso extra para contener dicha información sobre la imagen. En este caso podemos aprovechar dicho recurso para que sea el *envelope* de la imagen, y contenga además todos los enlaces y controles *hypermedia* necesarios.

6.7 Servicios web autodescriptivos

OPTIONS y HEAD

Como se aprecia en el ejemplo anterior cada documento nos va dando los siguientes pasos, y es cuestión de seguir los enlaces usando los métodos adecuados según la semántica HTTP[^8]. Pero, ¿cómo sabemos que información enviar y en qué formatos? Una cosa interesante sería hacer una petición con el método `OPTIONS`. Recordemos el ejemplo que apareció anteriormente:

```
OPTIONS /rest/libro/465 HTTP/1.1
Host: www.server.com

```

Y la respuesta:

```
HTTP/1.1 200 OK
Allow: GET, PUT, POST, OPTIONS, HEAD, DELETE, PATCH
Accept-Patch: application/book-diff+json
Content-Length: 0

```

El servidor nos responde con los métodos admisibles por dicha URI, y si soporta o no el método `PATCH`. Siguiendo la semántica HTTP, y quizás algunas convenciones adicional que habría que documentar, el cliente puede averiguar de forma automática las operaciones disponibles. Además `OPTIONS` nos permite un cuerpo en la respuesta, donde se especificarían detalles adicionales a nivel de aplicación.

Otro método útil es `HEAD`. El método `HEAD` es idéntico a `GET`, excepto que el servidor no devuelve datos y por lo tanto no intenta acceder a información ni realizar ninguna consulta. La utilidad de `HEAD` está en que nos devuelve exactamente las misma cabeceras y respuestas HTTP que como si hubiéramos hecho un `GET`. Por ejemplo:

```
HEAD /rest/libro/465 HTTP/1.1
Host: www.server.com
If-None-Match: W/"686897696a7c876b7e"
If-Modified-Since: Wed, 01 Sep 2012 13:24:52 GMT
Accept: application/json

```

Y la respuesta:

```
HTTP/1.1 200 Ok
Content-Type: application/json;charset=utf-8
Date: Tue, 27 Dec 2012 05:25:19 GMT
Expires: Tue, 27 Dec 2012 11:25:19 GMT
Cache-Control: max-age=21600
Last-Modified: Tue, 27 Dec 2012 03:25:19 GMT
ETag: W/"9082aff9627ab7cb60"

```

Con lo que sabemos si el recurso ha cambiado o no y su nuevo `ETag`. `HEAD` es muy útil para comprobar si un link funciona como el cliente espera. Pero también nos permite averiguar que tipos mime soporta. Si la respuesta hubiera sido:

```
HTTP/1.1 406 Not acceptable

```

Entonces sabríamos si dicho recurso soporta el tipo mime o no por adelantado.

Una ventaja extra de `HEAD` es que podemos usar web linking para añadir un enlace a un documento de descripción del recurso, tal y como veremos en la sección siguiente.

Usando una mezcla de `HEAD` y `OPTIONS` podemos inspeccionar las capacidades de un recurso REST de forma automática, minimizando al máximo el acoplamiento.

Documentos de descripción

Un problema de usar `OPTIONS` es que según el estándar la respuesta no debe ser almacenada en cache, esto incluye almacenarla en memoria dentro del cliente que consuma nuestra API para futura referencia. También puede ocurrir que `OPTIONS` y `HEAD` no estén permitidos para un recurso, o que un *firewall* no nos permita acceder al recurso mediante estos métodos.

Una alternativa a usar `OPTIONS` y `HEAD` es usar un documento de descripción del recurso. Este documento no necesita incluir mucha información. Basta con describir que tipos mime soporta, que enlaces y controlas admite, y que verbos HTTP están implementado. Así puede incluir texto para consumo humano. De hecho es buena idea que este documento descriptivo admita el tipo mime de texto plano o HTML para que sirva de documentación al programador. El documento que describe el recurso debería estar enlazado mediante un enlace de tipo `rel="describedby"`.

Un ejemplo de tal documento podría ser:

```
{
  "allow":["GET", "PUT", "HEAD", "DELETE", "PATCH"],
  "accept": "application/json, application/xml, text/html; q=0.5",
  "accept-patch": "application/book-diff+json"
  "schema": "http://www.server.com/rest/schemas/basket.xsd",
  "link-types":["self", "next", "prev"]
  // ... Cualquier otra información
}
```

En este ejemplo se opta por un enfoque simple, donde básicamente se mezcla la información que hubiera resultado de hacer un `OPTIONS` y un `HEAD`. Adicionalmente se añade información sobre que XML Schema usar (en caso de que se quiera usar XML) y los tipo de enlace que soporta el recurso. Pero por supuesto esto no es más que un mero ejemplo, otros formatos podrían ser más adecuados dependiendo de las circunstancias.

Conclusión

Siguiendo esta filosofía se puede entender por qué en el diseño de APIs REST no se usa nada equivalente al WSDL de los servicios web clásicos. Los documentos WSDL nos proporcionan un medio de automatizar los aspectos sintácticos de interoperabilidad entre servicios. Con un generador de código podemos procesar un WSDL para generar el código necesario para analizar y construir los mensajes SOAP y realizar las llamadas a los correspondientes puntos de servicio.

Sin embargo la dificultad de consumir una API pública no está en la sintaxis de los formatos de datos, sino en la semántica de éstos. El enfoque *hypermedia* nos proporciona unas reglas y patrones mínimos, de bajo acoplamiento, que nos permiten homogeneizar la forma de consumir nuestros servicios. La parte importante queda en la documentación y en la definición de los tipos mime a utilizar. Lo único que se necesita para desarrollar un cliente que pueda consumir una API *hypermedia* es:

- Una primera URL como punto de entrada a nuestro sistema.
- Una documentación describiendo la semántica de los tipos mime específicos de nuestra API. Esto no sólo incluye las entidades, sino también los formularios. Se recomienda basar dichos tipos en otros formatos ya existentes (JSON, XML, ATOM, HTML). La descripción de la estructura sintáctica la podemos dejar en mános de herramientas que consuman XML Schema o Relax NG o algo similar.
- Una documentación describiendo como consumir cada tipo de enlace (métodos a usar, tipos mime aceptables, patrones de uso).

No es de extrañar que los intentos de acercar WSDL a REST[9] o de crear equivalentes a WSDL en REST, como es el caso de WADL[10], no hayan tenido realmente mucho éxito. Al fin y al cabo no aportan mucho a lo que ya te da la filosofía *hypermedia* de diseño de APIs REST.

Sin embargo no debemos caer en la tentación de creer que con el enfoque *hypermedia* no debemos documentar nada, y que podemos lograr un cliente genérico para cualquier tipo de API. Cierto, con *hypermedia* nuestra documentación es mucho más ligera, y podemos ahorrarnos documentar la mayoría de los detalles sintácticos y de HTTP en ella, ya que se consideran "estándares" y por lo tanto cubiertas por posibles librerías y frameworks *hypermedia*. Sin embargo la parte semántica de nuestra API, qué significa cada campo, qué representa cada proceso de negocio y cada mensaje dentro de él, etc. son algo que hay que documentar. Si la API está bien diseñada, simplemente tendremos un esfuerzo de desarrollo a este nivel semántico, y nos ahorraremos todo el detalle de bajo nivel. Pero aun así, seguiremos teniendo que leer la documentación y hacer algo de desarrollo especifíco para consumir cada API *hypermedia*.

[9] WSDL 2.0: http://www.w3.org/2002/ws/desc

[10] WADL: http://wadl.java.net

6.8 ¿Qué modela nuestra API? ¿Aplicaciones o procesos?

Una de las dificultades de definir una API hypermedia consiste en lograr el grado adecuado de abstracción de nuestra API. El objetivo es modelar casos de uso de nuestro sistema, pero de una forma que pueda ser usada en contextos que no hayamos previsto.

Desde este punto de vista un antipatrón muy usual es modelar la API para que mimetice el flujo de uso de la aplicación por parte del usuario humano. Esto no es deseable ya que acoplaría la API a un diseño específico de una interfaz de usuario específica. ¿Qué ocurriría si el diseño de la UI cambia? ¿O si alguna otra aplicación con una UI totalmente distinta quiere usar nuestra API? Al fin y al cabo uno de los escenarios más importantes de uso de una API es que pueda dar servicio a múltiples sistemas y aplicaciones, muchas de ellas no diseñadas por nosotros y no bajo nuestro control. Por lo tanto es importante mantener nuestra API desacoplada de una aplicación específica, y pensar más a nivel de procesos de negocio.

El extremo contrario, hacer nuestra API excesivamente genérica, también es un antipatrón, ya que así terminamos definiendo de forma encubierta una API orientada a datos (CRUD), y cargamos al consumidor con la responsabilidad de implementar la lógica de los proceso de negocio.

El mantener un sano equilibrio entre ambos extremos es complicado, ya que implica definir el proceso de negocio que queremos modelar, en vez de fijarnos en las "pantallas" de una aplicación o en el "esquema" de nuestra base de datos. Antes de dar por bueno un diseño de nuestra API, deberíamos preguntarnos que ocurriría si cambiamos el esquema de datos, o si decidimos en un futuro alterar la UI de nuestra aplicación. Pero sobre todo hay que tener en cuenta si dicha API es sencilla de usar para un tercer equipo de desarrolladores que conozca REST. Al fin y al cabo la API constituye el equivalente a una interfaz de usuario pero para otros desarrolladores. No debemos olvidar que el objetivo de nuestra API es que otros la usen.

6.9 ¿Un tipo mime por API?

Como hemos visto hasta ahora si queremos modelar una API hymermedia necesitamos usar tipos mime que soporten enlaces y controles. Algunos tipos, como HTML lo soportan de forma nativa, otros como XML pueden soportarlo usando extensiones como XLink[11], y algunos, como las imágenes, vídeo o audio, no lo soportan en absoluto. Pero existe un caso intermedio, en el que el tipo mime podría soportarlo, pero no existe ninguna extensión para ello. Este es el caso de JSON, lo que constituye un problema, ya que es el formato más usado en las APIs REST.

En la práctica lo que ha ocurrido es que cada API ha modelado a su manera la forma de definir enlaces y controles sobre JSON. Si observamos los ejemplos atentamente, en ninguno se especifica el tipo `application/json`, sino que todos son de la forma `application/x-$$$$+json`. Esto indica

[11] XLink: http://www.w3.org/TR/xlink

que es un tipo derivado de JSON, de ahí el `+json` final, y por lo tanto se puede procesar como si fuera JSON plano. Pero a la vez estamos indicando mediante el prefijo `x-` que es un tipo propietario, y que necesitamos leer la documentación y codificar lógica adicional para poder consumir los enlaces. Esto tiene sentido ya que JSON no soporta enlaces y por lo tanto ningun parser de JSON va a saber como extraer los enlaces, con lo que necesitamos añadir una capa más de software.

Esto es problemático ya que cada vez que nos enfrentemos a una API hypermedia que soporte JSON el tipo mime va a ser distinto, con lo que tendremos que hacer un nuevo (pero pequeño) desarrollo. Claramente esto es un problema de interoperabilidad: ¿acaso no estamos usando JSON porque es estándar?

Este tipo de problemas ha causado un debate encendido en la comunidad de REST. Básicamente hay tres bandos:

- Usemos *Web Linking*. Como hemos explicado, nos permite incrustar enlaces las cabeceras HTTP, con lo que teóricamente el problema está solucionado.
- No usemos JSON, no soporta ni enlaces ni controles por lo que no es apto para APIs hypermedia. Los miembros de este bando proponen usar otros tipos mime que soporten multimedia, como HTML, XML o Atom.
- Extendamos JSON o creemos un nuevo tipo mime basado en JSON que soporte enlaces y controles de forma estándar.

El autor piensa que el uso de *Web Linking* es valioso, pero lo concibe como algo auxiliar, para casos donde no haya otra alternativa. Además todavía no hay mucha experiencia con este estándar y no sabemos como se van a comportar los nodos intermedios de internet con la nueva cabecera `Link`. Por otro lado el autor cree que los enlaces forman parte del propio estado del recurso y por lo tanto deberían estar en el propio documento.

En cuanto a no usar JSON, el autor no lo ve una alternativa viable. Todo el mundo está usando JSON, y es un formato sencillo y muy práctico. Siendo realista esta opción no tiene mucho futuro. Esto no quiere decir que el autor recomiendo dar soporte únicamente a JSON, ni mucho menos. Cuantos más tipos mimes soporte una API más interoperable será. Es conveniente dar soporte a HTML y a XML también.

La otra alternativa, crear un nuevo tipo mime que extienda JSON con capacidades hypermedia, es muy sensata. Al fin y al cabo JSON tiene la capacidad de soportar hypermedia, ¿por qué simplemente no estandarizarlo? De hecho existen al menos dos propuesta de estándar en este sentido. Algunos podrían pensar que el que existan varios formatos compitiendo es malo. El autor cree justo lo contrario, este tipo de competencia servirá para mejorar ambos formatos, y REST incentiva que se usen varios tipos mime y no restringirse a un único formato.

Finalmente, ¿no encarecerá el desarrollo de clientes hypermedia el hecho de que existan tantos tipos mime? La respuesta es que no, a condición de que no se usen tipos propietarios. Si sólo se usan tipos mime estándar, existirán librerías estándar que consuman dichos formatos. En otras palabras, el desarrollador no tendrá que preocuparse de todo esto, ya que la tarea de parsear un tipo mime, y

extraer datos, enlaces y controles, estará automatizada en la librería, framework o arquitectura que use. Por lo tanto el futuro de REST pasa por la estandarización de tipos mime capaces de soportar hypermedia, y evitar la proliferación de tipos propietarios.

En el resto del libro exploraremos como encajan en una API REST los tipos mime hypermedia que más se usan, ya sean viejos conocidos como XML o HTML, o bien extensiones de JSON.

7 Hypertext Application Language (HAL)

7.1 Introducción

Hypertext Application Language (HAL)[1] es un tipo mime que extiende JSON con la capacidad de poder definir enlaces y controles. De esta manera HAL nos permite no sólo representar los datos de un recurso REST, como hacía JSON, sino las operaciones disponibles en este (controles) y las relaciones con otros recursos (enlaces). Adicionalmente HAL nos permite incrustar recursos unos dentro de otros, lo que es muy útil en REST ya que ahorra llamadas de red. Desde un punto de vista de rendimiento esto es importante ya que en general el factor que produce más coste en las comunicaciones móviles y a través de internet es la latencia.

El tipo mime para HAL es `application/hal+json` y ya existen multitud de librerías en varios lenguajes que permiten parsear documentos HAL.

7.2 Links

HAL no es más que una extensión de JSON, por lo tanto a la hora de representar los datos de un recurso no aporta nada nuevo. Simplemente hay que definir un documento JSON normal con la información pertinente.

A la hora de representar enlaces, HAL nos propone que el documento JSON tenga una propiedad especial llamada `_links`. El valor de esta propiedad es un objeto JSON, donde cada una de sus propiedades es un enlace. El nombre de cada propiedad es el valor del atributo `rel`, que tal y como vimos en el capítulo anterior representa el tipo del enlace. El valor de la propiedad es simplemente otro objeto JSON que representa el resto de los atributos del enlace. En el caso de que existieran varios enlaces del mismo tipo, entonces tendríamos un array con un objeto por enlace. Por ejemplo:

```
{
  "_links": {
    "self": { "href": "/product/Re23SF" },
    "next": {
      "href": "/product/3rq245",
      "title": "Next product"
    },
    "prev": {
      "href": "/product/9wx72d",
```

[1] Hypertext Application Language: http://stateless.co/hal_specification.html y http://bit.ly/TQZwqM

```
      "title": "Previous product"
    },
    "http://www.my.co/rels/stores": [
      {
        "href": "/store/245",
        "title": "Central store"
      },
      {
        "href": "/store/33",
        "title": "iChop Store"
      }
    ]
  },
  "price": 233.0,
  "description": "iJam4000"
}
```

En el campo `_links` tenemos un objeto con cuatro propiedades representando cuatro tipos de enlaces: `self`, `next`, `prev` y `http://www.my.co/rels/stores`. Los tres primeros son tipos de relación estándares correspondientes a un enlace al recurso en si, al siguiente recurso y al anterior recurso. El último tipo de enlace es propietario y representa las tiendas donde se vende ese producto. Este último tiene como valor un array, ya que hay varios enlaces del mismo tipo.

Los atributos soportados por HAL para un enlace son `href`, `title`, `templated`, `name`, `type`, `deprecation` y `hreflang`. En general la semántica coincide con *Web Linking* y con lo explicado en el capítulo anterior. El atributo `name` nos permite referenciar de forma sencilla un enlace desde otra parte del documento, y distinguir entre varios enlaces del mismo tipo. Por otro lado el atributo `deprecation` está presente en el caso de que el enlace vaya a ser eliminado en una futura versión de la API.

Al igual que en *Web Linking* el contexto de navegación de los enlaces es el propio recurso, con lo que cualquier URI relativa debe ser interpretada respecto a éste.

7.3 Recursos incrustados

Como se ha comentado, HAL nos permite incluir recusos incrustados de tal manera que no necesitemos realizar otra llamada de red. La solución es muy simple, el documento HAL puede presentar un atributo opcional llamado `_embedded`.

Dicho atributo se comporta de forma similar a como lo hace `_links`. Consiste en un objeto JSON que contiene varias propiedades. El nombre de cada propiedad corresponde con el valor que tendría `rel` en un hipotético enlace que apuntara al recurso incrustado. El valor es el propio recurso incrustado, que se representa a su vez con otro documento HAL. Es decir, se trataría de un nuevo objeto JSON con los datos del recurso y un atributo `_links`. Opcionalmente podría tener otro atributo `_embedded`, lo que añadiría otro nivel de anidamiento. Veamos un ejemplo:

```
{
  "_links": {
    "self": { "href": "/product/Re23SF" },
    "http://www.my.co/rels/stores": [
      {
        "href": "/store/245",
        "title": "Central store"
      },
      {
        "href": "/store/33",
        "title": "iChop Store"
      }
    ]
  },
  "_embedded": {
    "next": {
      "_links": {
        "self": { "href": "/product/3rq245" },
        "next": {
          "href": "/product/3rq2xx",
          "title": "Next product"
        },
        "prev": {
          "href": "/product/Re23SF",
          "title": "Previous product"
        }
      },
      "price": 45.2,
      "description": "cortador iJam4000"
    },
    "prev": {
      "_links": {
        "self": { "href": "/product/9wx72d" },
        "next": {
          "href": "/product/Re23SF",
          "title": "Next product"
        },
        "prev": {
          "href": "/product/y2d245",
          "title": "Previous product"
        }
      },
```

```
43        "price": 99.99,
44        "description": "Ultra speakers"
45      }
46    },
47    "price": 233.0,
48    "description": "iJam4000"
49  }
```

Dentro de `_embedded` tenemos ahora dos objetos: `next` y `prev`. El contenido de estos objetos es a su vez otro documento HAL representando los datos y enlaces de el producto anterior y el siguiente. Al igual que en los enlaces, si hubiéramos tenido varios recursos incrustados que compartieran el mismo valor de `rel`, hubiéramos usado un array.

Hay que hacer notar que en un recurso incrustado, las URIs relativas que aparecieran en cualquiera de sus enlaces, deben ser resueltas usando la URI del propio recurso como base. Es decir, no hay que tomar la URI del recurso raíz como base, sino la del recurso que directamente contiene el enlace.

7.4 Curie

A veces los valores de `rel` no son estándar, como pudiera ser el caso de `http://www.my.co/rels/stores`. Como ya vimos, en estos casos se suele usar una URI como medio de crear un espacio de nombres, y a la vez apuntar a una posible página de documentación o a un recurso con metainformación sobre el tipo de enlace.

Si vamos a usar muchas veces este tipo de relaciones no estándar podemos simplificar significativamente los documentos usando una técnica llamada "Curie"[2]. Esta técnica es la equivalente a los espacios de nombres de XML pero adaptada a otros tipos mime y estándares. HAL al igual que Web Linking o XHTML la soportan.

La forma de definir estos "espacios de nombres" consiste en añadir un conjunto de enlaces especiales, con tipo tipo de relación `curies`, dentro de la sección `_links` del objeto raíz. En este atributo `curies` definimos un array de enlaces de tipo URI Template. En el atributo `name` de cada uno de estos enlaces se indica la abreviatura del espacio de nombres. En `href` se coloca una URI template, que una vez evaluada, devuelve el valor completo del espacio de nombres.

Veamos un ejemplo:

[2] A syntax for expressing Compact URIs: http://www.w3.org/TR/curie

```
{
  "_links": {
    "curies": [{
      "name": "sh",
      "href": "http://www.my.co/rels/{rel}",
      "templated": true
    }],
    "self": { "href": "/product/Re23SF" },
    "next": {
      "href": "/product/3rq245",
      "title": "Next product"
    },
    "prev": {
      "href": "/product/9wx72d",
      "title": "Previous product"
    },
    "sh:stores": [
      {
        "href": "/store/245",
        "title": "Central store"
      },
      {
        "href": "/store/33",
        "title": "iChop Store"
      }
    ]
  },
  "price": 233.0,
  "description": "iJam4000"
}
```

Se trata de un mecanismo ingenioso. Al igual que en los espacios de nombres de XML, el valor de `rel` se divide en dos partes usando `":"`. La primera parte es la abreviatura del espacio de nombres, que se usa para encontrar un enlace dentro de `curies` cuyo nombre coincida con este. La segunda parte es el valor del parámetro `rel` a usar para evaluar la URI Template y obtener el valor completo del espacio de nombres.

7.5 Un ejemplo

Si recordais el anterior capítulo, para modelar el carrito de la compra, su formulario asociado tuvimos que crear dos tipos mime propietarios: `application/x-shop.basket+json` para el carrito, y

`application/x-shop.form+json` para el formulario. Mediante HAL, podemos representar todo esto usando el mismo tipo mime:

```
{
  "_links": {
    "curies": [{
      "name": "shop",
      "href": "http://www.shop.com/rels/{rel}",
      "templated": true
    }],
    "self": {
      "href": "/shop/basket/234255",
      "title": "My shopping list"
    },
    "shop:products": {
      "href": "/shop/products{?pagesize}",
      "title": "Product catalog",
      "templated": true
    },
    "shop:search/books": {
      "href": "/shop/books{?pagesize}",
      "title": "Search books available in the shop",
      "templated": true
    },
    "shop:search/offers": {
      "href": "/shop/offers{?pagesize}",
      "title": "Special offers",
      "templated": true
    }
  },
  "_embedded": {
    "shop:form": {
      "_links": {
        "self": {
          "href": "/shop/products/43/orderform?basket=234255",
          "title": "Order product form"
        },
        "http://www.shop.com/rels/target": {
          "href": "/shop/basket/234255/items/3",
          "title": "Order product form target"
        }
      },
      "method": "PUT",
```

```
41        "type": "application/hal+json",
42        "class": "add-product"
43      },
44      "shop:item":[
45        {
46          "_links": {
47            "self": {
48              "href": "/shop/basket/234255/items/1"
49            },
50            "shop:basket": {
51              "href": "../..",
52              "title": "My shopping list"
53            },
54            "shop:product": {
55              "href": "/shop/products/43"
56            }
57          },
58          "quantity": 2
59        },
60        {
61          "_links": {
62            "self": {
63              "href": "/shop/basket/234255/items/2"
64            },
65            "shop:basket": {
66              "href": "../..",
67              "title": "My shopping list"
68            },
69            "shop:product": {
70              "href": "/shop/products/233"
71            }
72          },
73          "quantity": 10
74        }
75      ]
76    },
77    "total": { currency: "€", amount: 345.2 }
78  }
```

Obsérvese como hemos conseguido incrustar un formulario para rellenar el carrito de la compra mediante el uso de `_embedded`. El recurso incrustado cuyo `rel` es `shop:form` representa el formulario. Hemos modelado el recurso formulario de forma similar a como lo hicimos en el capítulo dedicado a *hypermedia*. En este caso el campo `method` nos indica que método HTTP usar para ejecutar el

formulario, y el enlace con `rel=/rels/target` la URI sobre la que ejecutar la petición. El campo `type` indica que tipo mime usar para formatear el cuerpo de la petición, es decir, los datos que vamos a enviar. Sin embargo, al contrario que en el capítulo anterior, en este ejemplo no usamos un tipo mime propietario, sino `application/hal+json`. Esto implica que debemos documentar claramente que tipo de datos debemos usar para rellenar este formulario, ya que `application/hal+json` es un contenedor genérico que puede contener cualquier cosa. Esto implica que no es suficiente con usar una supuesta librería estándar que manipulara HAL, sino que hay que hacer un pequeño desarrollo en el cliente, específico para nuestra API, que sepa interpretar este formulario. De ahí la necesidad de documentar este punto claramente. Una pista de esto la indicamos en el campo `class` del formulario, un campo totalmente propietario de nuestra API, que permitiría indicar al cliente que está tratando con un formulario para añadir un producto a un pedido, y seleccionar la lógica correspondiente.

Otro detalle a tener en cuenta es que ahora cada línea de pedido es un recurso independiente, con su propia URI. Este diseño nos permite actualizar y borrar cada línea de pedido de forma independiente, haciendo `PUT` o `DELETE` a cada una de estas URIs. De esta forma evitamos tener que hacer `PUT` sobre el carrito para reescribir el estado completo de éste. También evitamos tener que hacer una actualización parcial mediante `POST` o `PATCH`, que son más complejos de usar y no idempotentes.

Un efecto de este diseño es que el campo `items` que usamos en el anterior capítulo desaparece. Esto es así porque en HAL un recurso no se puede incrustar directamente como datos. Como ahora las líneas de pedido son recursos en si mismos, debemos añadirlas en `_embedded` y usar un nuevo tipo de relación `shop:item`. Alternativamente podríamos haber usado `_links` para almacenar referencias a las líneas de pedido.

Si no nos gusta este diseño podríamos cambiarlo. Podemos decidir por ejemplo que las lineas de pedido no son recursos y que vamos a usar `PATCH` para actualizar cada una de ellas. También decidimos que el formulario no va a ser incrustado en el documento, sino referenciado mediante un enlace. Con ambos cambios el documento HAL quedaría como:

```
{
  "_links": {
    "curies": [{
      "name": "shop",
      "href": "http://www.shop.com/rels/{rel}",
      "templated": true
    }],
    "self": {
      "href": "/shop/basket/234255",
      "title": "My shopping list"
    },
    "shop:products": {
      "href": "/shop/products{?pagesize}",
      "title": "Product catalog",
      "templated": true
```

```
    },
    "shop:search/books": {
      "href": "/shop/books{?pagesize}",
      "title": "Search books available in the shop",
      "templated": true
    },
    "shop:search/offers": {
      "href": "/shop/offers{?pagesize}",
      "title": "Special offers",
      "templated": true
    },
    "shop:form": {
      "href": "/shop/products/43/orderform?basket=234255",
      "title": "Order product form"
    }
  },
  "items": [
    {
      "_links": {
        "http://www.shop.com/rels/product": {
          "href":"/shop/products/43"
        }
       },
      "quantity": 2
    },
    {
      "_links": {
        "http://www.shop.com/rels/product": {
          "href":"/shop/products/4323"
        }
      },
      "quantity": 10
    }
  ]
  "total": { currency: "€", amount: 345.2 }
}
```

En este ejemplo vemos un documento más compacto. Es normal, ya que por un lado no hemos incrustado el formulario, y por el otro las líneas de pedido no son recursos, con lo que eliminamos la necesidad de modelar el enlace `self` o su relación con el carrito.

Sin embargo este diseño no está exento de inconvenientes. El más grave se corresponde con como modelamos cada línea de pedido. El problema está en que la línea de pedido debe indicar no sólo

la cantidad de cada producto que vamos a comprar, sino qué producto en si queremos. La única forma de hacerlo es mediante un enlace al producto, pero como hemos decidido que las líneas de pedido no son recursos entonces no pueden contener enlaces, al menos en teoría. Esto puede dar problemas de interoperabilidad, ya que un cliente que consuma HAL no espera encontrar un enlace fuera de `_links`, y mucho menos dentro del cuerpo de datos del recurso. Si queremos que este diseño funcione debemos documentar este detalle, a fin de que los posibles clientes de nuestra API implementen lógica adicional para extraer el enlace al producto de cada línea de pedido.

Por lo tanto el primer diseño, donde cada línea de pedido es un recurso en si mismo, es un diseño superior. No se necesita lógica adicional para extraer todos los datos, enlaces y recursos del sistema.

Nótese que en HAL la decisión de diseño de representar la línea de pedido como un recurso o no implica un cambio no retrocompatible de nuestra API. Por otro lado la decisión de incrustar o no el formulario es totalmente transparente a nuestros clientes y no implica un cambio de API. Esto es debido a que en HAL los documentos pueden estar enlazados en la sección `_links` o incrustados en la sección `_embedded` y cualquier librería que consuma HAL debe estar preparada para ambos casos. En este sentido la única decisión de diseño que habría que documentar y que realmente impacta nuestra API es definir qué piezas de información son recursos con sus propias URIs y cuales no.

7.6 HAL y XML

Recientemente ha aparecido una representación en XML para HAL. El tipo mime es `application/hal+xml`.

Básicamente se trata de una traducción de los conceptos anteriormente presentados pero en XML. Cada documento HAL tiene un elemento raíz `<resource>`. Dicha etiqueta contiene un atributo `href` que toma el valor el enlace de tipo `self`. Dentro de ese elemento cada campo de datos se modela mediante un elemento hijo. Los enlaces se modelan usando el elemento `<link>` con las propiedades habituales `href`, `rel`, `name`, `templated`, etc. Existe un elemento `<link>` por cada enlace, y no se necesita ningún elemento que los englobe. Finalmente podemos representar recursos incrustado de forma muy simple: anidando elementos `<resource>`. Veamos el ejemplo anterior del carrito pero usando `application/hal+xml` en vez de `application/hal+json`:

```
<resource href="/shop/basket/234255"
          xmlns:shop="http://www.shop.com/rels/">
  <link href="/shop/products{?pagesize}"
        title="Product catalog"
        rel="shop:products"
        templated="true"/>
  <link href="/shop/books{?pagesize}"
        title="Search books available in the shop"
        rel="shop:search/books"
        templated="true"/>
  <link href="/shop/offers{?pagesize}"
```

```
12          title="Special offers"
13          rel="shop:search/offers"
14          templated="true"/>
15    <resource href="/shop/products/43/orderform?basket=234255"
16              rel="shop:form">
17      <link href="/shop/basket/234255/items/3"
18            title="Order product form target"
19            rel="shop:target"/>
20      <method>PUT</method>
21      <type>application/hal+json</type>
22      <class>add-product</class>
23    </resource>
24    <resource href="/shop/basket/234255/items/1"
25              rel="shop:item">
26      <link href="../.."
27            title="My shopping list"
28            rel="shop:basket"/>
29      <link href="/shop/products/43"
30            rel="shop:product"/>
31      <quantity>2</quantity>
32    </resource>
33    <resource href="/shop/basket/234255/items/2"
34              rel="shop:item">
35      <link href="../.."
36            title="My shopping list"
37            rel="shop:basket"/>
38      <link href="/shop/products/233"
39            rel="shop:product"/>
40      <quantity>10</quantity>
41    </resource>
42    <total>
43      <currency>€</currency>
44      <amount>345.2</amount>
45    </total>
46  </resource>
```

Nótese como el atributo `rel` de los elementos `<resource>` anidados representa la relación que tendría un enlace a dicho recurso.

7.7 Conclusión

HAL es un estándar de mínimos, añade lo mínimo necesario a JSON como para poder representar enlaces y recursos incrustados. Esto es una ventaja porque resulta muy sencillo escribir un parser de HAL. Basta con procesar el documento HAL con cualquier parser JSON y hacer un procesamiento adicional para extraer los enlaces y recursos incrustados.

Otra gran ventaja es que en HAL el hecho de que un recurso esté enlazado o incrustado es transparente al consumidor de la API.

Una posible crítica que le podemos hacer a HAL es que no soporta controles, sino sólo enlaces. No existe consenso a este respecto. Muchos ven que los controles no son más que enlaces con más atributos (por ejemplo `type` y tal vez `method`). Lo mismo ocurre con los formularios, que se pueden modelar como recursos (incrustados o no) con un enlace `target` y un atributo `method`. Otros sin embargo optan por modelarlos de forma explícita y darles un estatus de primer ciudadano. ¿Cómo se modela un campo de entrada de texto en un formulario? HAL opta por mantenerse silecioso a este respecto.

En cualquier caso HAL es un estándar muy sencillo de adoptar, y el autor considera que es mucho más efectivo adoptarlo que tratar de crear un tipo propietario.

8 SIREN

8.1 Introducción

En el capítulo anterior hemos visto como HAL se postula como candidato a formato hypermedia basado en JSON. Sin embargo no es el único candidato.

SIREN[1] es una alternativa a HAL. Como este último, SIREN también trata de crear un formato hypermedia basado en JSON. Sin embargo el enfoque de SIREN es ligeramente distinto. SIREN define una estructura más explícita para separar lo que son los datos del recurso en si, de los enlaces y recursos incrustados. Además de los enlaces también permite modelar formularios. En el argot de SIREN los recursos son llamados entidades, y los formularios son acciones.

El tipo mime para SIREN es `application/vnd.siren+json` y también se encuentra incluido en el registro de tipos mime estandarizados[2].

8.2 Datos

SIREN define una sección especial en el documento JSON que nos permite definir los datos que transporta el recurso (o entidad como son llamadas en esta especificación).

Esta sección es muy sencilla, se trata de una propiedad llamada `properties` cuyo valor es un objeto JSON que contiene las propiedades del recurso.

En el ejemplo que pusimos en el capítulo anterior sobre un producto en una tienda online, podríamos tener lo siguiente:

```
{
  "properties": {
    "price": 233.0,
    "description": "iJam4000"
  },
  "links": [
    {
      "rel": [ "self" ],
      "href": "/product/Re23SF"
    }
  ]
}
```

[1] SIREN: https://github.com/kevinswiber/siren

[2] Directorio de tipos MIME estandarizados por IANA: http://www.iana.org/assignments/media-types/index.html

Donde la información sobre el producto está dentro del objeto `properties`. La sección `links` la veremos más adelante.

8.3 Tipos de datos

En SIREN se puede especificar los tipos de datos a los que pertenecen un recurso mediante la propiedad `class`. Dicha propiedad puede contener un array de cadenas. En dicho array se especifican los tipos de datos a los que pertenece el recurso. En SIREN se permite que un recurso pertenezca a varios tipos de datos, de ahí que el valor de `class` sea un array.

El ejemplo anterior se podría haber hecho uso de dicha propiedad para señalar que el recurso representa un producto, con lo que tendríamos:

```
{
  "class": [ "product" ],
  "properties": {
    "price": 233.0,
    "description": "iJam4000"
  },
  "links": [
    {
      "rel": [ "self" ],
      "href": "/product/Re23SF"
    }
  ]
}
```

Esta propiedad es opcional, aunque en muchas APIs puede resultar útil. Los diferentes valores de `class` pueden ser usados por el consumidor del recurso para activar el proceso de validación de datos apropiado. La lógica del consumidor podría tener una colección de validadores en función del tipo de recurso, y seleccionar el adecuado de acuerdo a los valores en `class`. En el caso que el consumidor sea una UI, se podría seleccionar también una vista y/o un controlador específico en función de los valores encontrados en `class`.

Se podría considerar que este es un enfoque fuertemente tipado a las APIs hypermedia. Si se va a usar `class` hay que tener mucho cuidado de definir los tipos de datos de forma extensible. No sería conveniente que si en un futuro el tipo de datos "producto" evolucionara para incorporar nueva información o contener nuevas acciones o enlaces los clientes de la antigua versión dejaran de funcionar, al romperse su algoritmo de validación. Es el mismo problema que se presenta al usar tecnologías como XML Schema para validar documentos XML, si no se ha diseñado el esquema de forma extensible para que pueda evolucionar de forma retrocompatible podemos tener problemas de mantenimiento de nuestra API.

8.4 Links

Los enlaces a otros recursos son representados dentro de una propiedad llamada `links`, que contiene un array con los enlaces presentes en el documento.

Cada enlace es un objeto JSON con dos propiedades. Una de ellas es la consabida propiedad `rel`, que en SIREN toma como valor un array. De esta forma podemos definir varios tipos para un mismo enlace. La otra es la propiedad `href`, que indica la URI a la que apunta el enlace.

En SIREN todo recurso debe poseer al menos un enlace de tipo `self` apuntándo a su URI.

Si incorporamos enlaces a nuestro ejemplo de producto obtenemos:

```
{
  "class": [ "product" ],
  "properties": {
    "price": 233.0,
    "description": "iJam4000"
  },
  "links": [
    {
      "rel": [ "self" ],
      "href": "/product/Re23SF"
    },
    {
      "rel": [ "next" ],
      "href": "/product/3rq245"
    },
    {
      "rel": [ "prev" ],
      "href": "/product/9wx72d"
    },
    {
      "rel": [ "http://www.my.co/rels/stores" ],
      "href": "/store/245"
    },
    {
      "rel": [ "http://www.my.co/rels/stores" ],
      "href": "/store/33"
    }
  ]
}
```

De nuevo tenemos los enlaces de navegación `next` y `prev`, y el enlace de tipo propietario `/rels/stores`, al igual que en el ejemplo del capítulo anterior. Contrástese el diseño de SIREN con el enfoque de

HAL, donde la sección de enlaces es un objeto, y el valor de `rel` es el nombre de cada una de las propiedades.

8.5 Recursos incrustados

Al igual que en HAL podemos modelar recursos incrustados de forma muy sencilla. La ventaja de usar recursos incrustados es que se ahorran llamadas de red, lo que puede ser conveniente en redes donde es la latencia, y no el ancho de banda, el factor principal en el tiempo de respuesta.

En SIREN los recursos incrustados están dentro de un array llamado `entities`. Un recurso incrustado sigue exactamente el formato de cualquier otro recurso SIREN. La única salvedad es que debe contener una propiedad `rel` con el mismo valor que tendría un hipotético enlace que apuntase a él.

Siguiendo con el ejemplo de los productos, si los recursos que representan el siguiente y el anterior producto se incrustaran en el documento en vez de referenciarlos mediante enlaces, tendríamos:

```
{
  "class": [ "product" ],
  "properties": {
    "price": 233.0,
    "description": "iJam4000"
  },
  "entities": [
    {
      "class": [ "product" ],
      "rel": [ "next" ],
      "properties": {
        "price": 45.2,
        "description": "cortador iJam4000"
      },
      "links": [
        {
          "rel": [ "self" ],
          "href": /product/3rq245"
        },
        {
          "rel": [ "next" ],
          "href": "/product/3rq2xx"
        },
        {
          "rel": [ "prev" ],
```

```
          "href": "/product/Re23SF"
        }
      ]
    },
    {
      "class": [ "product" ],
      "rel": [ "prev" ],
      "properties": {
        "price": 99.99,
        "description": "Ultra speakers"
      },
      "links": [
        {
          "rel": [ "self" ],
          "href": "/product/9wx72d"
        },
        {
          "rel": [ "next" ],
          "href": "/product/Re23SF"
        },
        {
          "rel": [ "prev" ],
          "href": "/product/y2d245"
        }
      ]
    }
  ],
  "links": [
    {
      "rel": [ "self" ],
      "href": "/product/Re23SF"
    },
    {
      "rel": [ "http://www.my.co/rels/stores" ],
      "href": "/store/245"
    },
    {
      "rel": [ "http://www.my.co/rels/stores" ],
      "href": "/store/33"
    }
  ]
}
```

La decisión de si un recurso debe aparecer incrustado, o si por el contrario está referenciado mediante un enlace, no implica un cambio que debamos documentar en nuestra API. Todo cliente SIREN debe estar preparado para poder consumir los recursos ya sea navegando los enlaces o descubriendo los recursos incrustados dentro del recurso actual, con lo cual tal cambio no debería romper a los consumidores de nuestra API. Esto nos permite definir una política dinámica en el servidor, que en función de la latencia y el tamaño de los recursos, decida si es más óptimo que estos sean incrustados o no.

8.6 Formularios

SIREN nos ofrece la posibilidad de modelar explícitamente las distintas acciones que podemos realizar sobre un recurso mediante formularios. Para ello podemos añadir una propiedad llamada `actions` que contiene un array de objetos JSON. Cada uno de estos objetos es el equivalente JSON de un formulario HTML5.

Cada acción pude tener los siguientes campos:

- `name` para representar el nombre de la acción o formulario. Es útil si el consumidor quiere localizar una acción por nombre.
- `title` para contener una descripción de la acción orientada al consumo humano.
- `method` indica qué método HTTP se debe usar para ejecutar la acción.
- `href` contiene la URI sobre la que hay que hacer una petición para ejecutar la acción. Se debe usar el método HTTP especificado en `method` para tal petición.
- `type` indica el tipo mime que debe usarse para codificar el cuerpo de la petición. Si no está presente se usa `application/x-www-form-urlencoded` por defecto.

Además de todas estas propiedades, el formulario contiene una propiedad `fields` con un array de objetos JSON. Cada uno de estos objetos es el equivalente en JSON de un control `<input>` de HTML5. Al igual que éstos, pueden contener las propiedades `name`, `type` y `value`, indicando el nombre del campo, su tipo (`text`, `password`, `search`, `tel`, etc.) y un posible valor por defecto.

El uso de controles de entrada basados en `<input>` de HTML5 es polémico. ¿Hasta que punto tiene sentido un campo de tipo `password` en una API orientada a la interconexión de máquinas?

Por otro lado algunos críticos sugieren que el tener que usar el equivalente a formularios HTML5 es bastante rígido, sobre todo en el caso de que los datos a enviar sean más complejos de los que puede soportar un formulario, o bien no exista un tipo de campo de entrada apropiado. Sin embargo no se puede negar

que la sencillez de dicho mecanismo lo hace ideal desde un punto de vista de interoperabilidad.

Finalmente los formularios también pueden estar tipados, y contener un campo de tipo `class` indicando la estructura de datos que soporta de entrada (que inputs y de que tipo contiene el formulario).

Siguiendo este esquema podemos modelar el carrito de la compra usando formularios.

```
{
  "class": [ "basket" ],
  "properties": {
    "total": { currency: "€", amount: 345.2 }
  },
  "entities": [
    {
      "class": [ "item" ],
      "rel": [ "/rels/order-item" ],
      "properties": {
        "quantity": 2
      },
      "links": [
        {
          "rel": [ "self" ],
          "href": "/shop/basket/234255/1"
        },
        {
          "rel": [ "/rels/product" ],
          "href": "/shop/products/43"
        },
        {
          "rel": [ "/rels/basket" ],
          "href": "/shop/basket/234255"
        }
      ]
    },
    {
      "class": [ "item" ],
      "rel": [ "/rels/order-item" ],
      "properties": {
        "quantity": 10
```

```
      },
      "links": [
        {
          "rel": [ "self" ],
          "href": "/shop/basket/234255/2"
        },
        {
          "rel": [ "/rels/product" ],
          "href": "/shop/products/233"
        },
        {
          "rel": [ "/rels/basket" ],
          "href": "/shop/basket/234255"
        }
      ]
    }
  ],
  "actions": [
    {
      "name": "search-products",
      "title": "Search products",
      "method": "GET",
      "href": "/shop/products",
      "type": "application/x-www-form-urlencoded",
      "fields": [
        {
          "name": "pagesize",
          "type": "range",
          "min": 10,
          "max": 100,
          "value": 15
        },
        {
          "name": "type",
          "type": "radio",
          "value": "all"
        },
        {
          "name": "type",
          "type": "radio",
          "value": "books"
        },
```

```
        {
          "name": "type",
          "type": "radio",
          "value": "offers"
        }
      ]
    },
    {
      "class": [ "add-product" ],
      "name": "order-product",
      "title": "Order product form",
      "method": "PUT",
      "href": "/shop/basket/234255/items/3",
      "type": "application/json",
      "fields": [
        {
          "name": "product-href",
          "type": "text"
        },
        {
          "name": "quantity",
          "type": "range",
          "min": 1,
          "max": 20
        }
      ]
    }
  ],
  "links": [
    {
      "rel": [ "self" ],
      "href": "/shop/basket/234255"
    }
  ]
}
```

Lo primero que salta a la vista es cómo se ha modelado la búsqueda de productos. En otros capítulos se ha estado modelando dicha funcionalidad mediante un enlace que usaba una URI Template. Sin embargo SIREN no soporta oficialmente enlaces con URI template, así que el autor ha optado por modelar la búsqueda mediante un formulario llamado `search-products`. El formulario usar el método `GET`, ya que se trata de una operación de lectura, sobre la colección `/shop/product`. Mediante el campo `pagesize` el cliente puede controlar al tamaño de la página de resultados. Además este campo está definido con el tipo `range` con lo que el servidor puede restringir el valor mínimo y

máximo que puede tomar dicho tamaño de página. El campo `type` controla qué tipo de búsqueda se realiza. Al ser de tipo `radio`, el servidor puede controlar en todo momento que sólo se admitan tipos de búsqueda que él implemente.

En este diseño el esquema de URIs cambia con respecto al presentado en anteriores capítulos, antes se tenían las URIs `/shop/product`, `/shop/product/offers` y `/shop/product/books` para modelar las distintas consultas predefinidas. Ahora mediante el formulario se usan disitintas *query strings* para diferenciar las consultas: `/shop/product?type=all`, `/shop/product?type=offers` y `/shop/product?type=books` respectivamente.

Este enfoque en principio no es problemático, ya que mediante el uso de campos tipo `radio` forzamos al cliente a elegir sólo entre los tipos predefinidos de búsqueda y ningún otro. También se restringe la posibilidad de buscar por cualquier criterio.

Usando el siguiente formulario, `order-product`, el cliente puede añadir un producto al carrito. Se especifica que debe hacer `PUT` contra `/shop/basket/234255/items/3` mediante las propiedades `href` y `method`. Este formulario simplemente tiene un campo `quantity` de tipo `range` y otro `product-href` de tipo `text`. El primero permite especificar cuantas unidades de un producto se quiere añadir al carrito. El segundo debe contener la URI del producto elegido. El formato a usar en el cuerpo de la petición es `application/json` tal y como se especifica en la propiedad `type`.

Una posible desventaja del diseño de este último formulario es que el campo `product-href` es de tipo `text`, con lo que cualquier texto sería en principio aceptable por la acción, cuando en realidad lo que se pretende es que dicho texto sea la URI del producto a añadir al carrito. Esto fuerza a hacer un pequeño desarrollo adicional en el consumidor que fuerze esta restricción, al no poderse controlar desde el punto de vista de una posible librería estándar que consumiera SIREN. Este sería uno de los puntos en los que una hipotética documentación de la API debería hacer énfasis. Para ayudar a indicar al consumidor de la API que use esta lógica extra, que fuerce que el valor de `product-href` sea una URI, usamos el valor del campo `class`, que en este caso es `add-product`. De esta manera indicamos que existe una capa de semántica adicional, fuera de los límites de SIREN, que el cliente debe implementar.

Por último, podemos averiguar el contenido actual del carrito mediante la propiedad `entities`. En ella observamos que existen dos recursos cuyo `class` tiene como valor `item`, representando una línea de pedido cada una. Cada uno de estos recursos tiene una única propiedad `quantity`, describiendo cuantas unidades de producto se han añadido al carrito. Para saber que producto referencia esta línea de pedido, basta con mirar en su sección `links` donde se encuentra un enlace al producto seleccionado, cuyo valor de `rel` es `/rels/product`. Finalmente cada una de estas líneas de pedido referencia a su vez al carrito mediante el enlade de tipo `rel=/rels/basket`. Con este diseño cada

línea de pedido puede ser consumida y manipulada tanto como un recurso independiente como un recurso incrustado en el carrito.

8.7 SIREN y XML

El objetivo último de SIREN, al igual que HAL, es definir una estructura de documento estándar donde se puedan representar enlaces, acciones y recursos incrustados, de tal manera que formatos que no soporten estos conceptos de forma nativa puedan ser usados en una API hypermedia. Por lo tanto ambos no se quedan únicamente en JSON sino que proponen ser usados como base para otros formatos de datos, como XML.

Sin embargo en la especificación de SIREN no aparecen aun ejemplos de como sería un supuesto documento XML que siga este formato, ni ningún tipo de guía, con lo cual estamos obligados a improvisar la equivalencia de SIREN en XML. Por lo tanto el autor considera que a día de hoy SIREN no es apto para ser usado en un formato XML.

8.8 SIREN vs. HAL

Aunque tanto SIREN como HAL tienen el mismo objetivo, y se basan en JSON, sus diseños son muy diferentes.

Por un lado HAL permite modelar enlaces ricos, con todos los atributos especificados en *Web Linking* y soportando URI templates. Por el otro SIREN nos ofrece un soporte más básico para éstos. En el diseño de SIREN subyace la idea de que los enlaces deben ser objetos sencillos que sólo pueden ser consumidos mediante `GET`. Si queremos hacer alguna acción más compleja que la mera navegación debemos modelarlo mediante un formulario y como tal incluirla en la sección `actions`.

De manera contrapuesta HAL opta por no modelar en absoluto las acciones y formularios, y permite al diseñador de la API definirlas mediante recursos con vida propia. En SIREN las acciones no son recursos en si mismas, sino que están incrustadas dentro de éstos, además de estar constreñidas por el modelo de formularios de HTML5. Esta falta de flexibilidad puede verse como un defecto de SIREN, pero también como un punto fuerte al permitir consumir los formularios de una forma más sencilla y estandarizada.

Otra gran diferencia es la posibilidad que tiene SIREN de modelar tipos de datos mediante el uso de `class`. Esta es una ventaja dudosa, ya que si bien nos permite ciertas ventajas ya mecionadas anteriormente, nos abre la puerta a cometer el *pitfall* más común en las APIs web: la especificación de tipos de datos sin la suficiente flexibilidad como para evolucionar en el futuro.

Por último el formato de documento propuesto por SIREN es menos compacto que el propuesto por HAL. Por ejemplo, ¿porqué necesitamos una sección de propiedades en vez de definirlas directamente como hace HAL? ¿Por que usar arrays en vez de objetos para los enlaces forzándonos a definir una atributo extra llamado `rel` dentro del enlace? ¿Es realmente útil poder definir varios `rel`?

Hasta cierto punto da la impresión de que SIREN trata de trasladar literalmente la representación HTML de los enlaces a JSON, sin aprovechar al máximo las capacidades de este último.

Algunos pueden pensar que de esta forma podemos extraer más fácilmente los datos al no estar mezclados con los enlaces y recursos embebidos. El autor piensa que en la mayoría de los lenguajes esto no es cierto. En concreto, en JavaScript, extraer los datos de un recurso HAL es relativamente sencillo:

```
function extractData(halDoc) {
  var data={};
  Object.keys(halDoc)
        .filter(function(key) {
          return key != '_embedded' &&
                  key != '_links';
        })
        .forEach(function(key) {
          data[key] = halDoc[key];
        });
  return data;
}
```

Lo que sí es cierto es que SIREN intenta ser lo más parecido a HTML posible dentro de JSON, y muchos programadores pueden estar más cómodos con SIREN que con HAL en este sentido.

En cualquier caso el autor considera que aunque SIREN no sea el formato hypermedia más optimizado, es bastante sencillo de entender y de consumir, con lo cual es una opción muy buena como base de una API hypermedia.

9 Collection+JSON

9.1 Introducción

Actualmente JSON está de moda, y como no hay dos sin tres, no podía faltar un capítulo dedicado al tercer formato *hypermedia* basado en JSON que está teniendo buena acogida: *Collection+JSON*. El tipo mime para este formato es `application/vnd.collection+json`.

Collection+JSON[1] es otra extensión de JSON que ha sido registrada como tipo mime estándar[2]. Este tipo mime ha sido diseñado específicamente para modelar colecciones simples con soporte para queries y formularios. Como ya veremos, implementa estos dos últimos conceptos echando mano de la técnica de templates.

9.2 Datos

Al contrario que SIREN o HAL, Collection+JSON se especializa en modelar colecciones. De esta forma, si tuviéramos una colección de productos, su representación en Collection+JSON podría ser algo como lo que sigue:

```
{
  "collection": {
    "href": "/product",
    "items": [
      {
        "href": "/product/9wx72d",
        "data": [
          {"name": "price", "value": 31.40},
          {"name": "description", "value": "USB Memory"}
        ]
      },
      {
        "href": "/product/Re23SF",
        "data": [
          {"name": "price", "value": 233.0},
          {"name": "description", "value": "iJam4000"}
        ]
      },
      {
```

[1] http://amundsen.com/media-types/collection/

[2] http://www.iana.org/assignments/media-types/application/vnd.collection+json

```
        "href": "/product/3rq245",
        "data": [
          {"name": "price", "value": 18.22},
          {"name": "description", "value": "Fantastic Book"}
        ]
      }
    ]
  }
}
```

En Collection+JSON todos los documentos deben ser objetos JSON con una única propiedad: `collection`. A su vez dicha propiedad es un objeto con diversa información. Por ejemplo, puede tener una propiedad `href` que contiene la URI de la colección. Se trata de un enlace de tipo `self`.

Otra propiedad importante es `items`, que contiene un array con todos los miembros de la colección. Cada uno de los miembros de la colección está modelado por otro objeto, que también posee una propiedad `href`. Los datos del recurso se encuentran dentro del campo `data` que contiene un array con todas las propiedades del objeto. Cada una de estas propiedades es a su vez un objeto, con los campos `name` para el nombre de la propiedad, y `value` para el valor de ésta.

Como se observa la estructura es un poco enrevesada, con varios niveles de anidamiento. Al contrario que SIREN y HAL, se opta por modelar las propiedades de datos como objetos, en vez de como propiedades simples usando JSON. Esto añade complejidad, pero a cambio nos permite una estructura más homogenea y podemos añadir información extra. De hecho Collection+JSON contempla una tercera propiedad, `prompt`, para añadir una descripción a los miembros de un objeto.

Otra cosa curiosa sobre Collection+JSON ocurre cuando hacemos un `GET` sobre la URI de un miembro de la colección. Nosotros esperaríamos que nos devolviera el objeto que describe al recurso que estamos pidiendo, pero no es así. Por ejemplo, si hacemos `GET` sobre `/product/Re23SF`, obtenemos:

```
{
  "collection": {
    "href": "/product",
    "items": [
      {
        "href": "/product/Re23SF",
        "data": [
          {"name": "price", "value": 233.0, "prompt": "Product name:"},
          {"name": "description", "value": "iJam4000"}
        ]
      }
    ]
  }
}
```

¿Qué ha ocurrido? ¿Han sido borrados los otros dos miembros de la colección? La respuesta es no. En Collection+JSON todos los documentos deben tener una propiedad `collection` como raíz, así que devolver el JSON del producto `iJam4000` no sería un documento válido. Por lo tanto no hay más remedio que envolverlo en una `collection`. Esto es bastante contraintuitivo y algunos podrían argumentar que un desperdicio de ancho de banda. Pero tiene como ventaja que es consistente y homogéneo; el cliente no necesita distinguir entre el formato para colecciones y formato para miembros individuales (compárese esto con Atom, como veremos en otro capítulo). Otra ventaja es que al envolver el recurso en una `collection`, disponemos de enlaces a la colección (en `href` de la `collection`), y de todos los controles *hypermedia* que esta tenga.

9.3 Links

Aparte del enlace tipo `self` modelado en la propiedad `href`, tanto las colecciones como los miembros de una colección pueden tener otros enlaces. Para ello se usa el campo `links`. Veamos como quedaría nuestro `iJam4000` con enlaces:

```
{
  "collection": {
    "href": "/product",
    "links": [
      {
        "rel": "http://www.my.co/rels/stores-collection",
        "href": "/store",
        "prompt": "Our stores"
      }
    ],
    "items": [
      {
        "href": "/product/Re23SF",
        "data": [
          {"name": "price", "value": 233.0, "prompt": "Product name:"},
          {"name": "description", "value": "iJam4000"}
        ],
        "links": [
          {
            "rel": "next",
            "href": "/product/3rq245"
          },
          {
            "rel": "prev",
            "href": "/product/9wx72d"
```

```
            },
            {
              "rel": "http://www.my.co/rels/product/pic",
              "href": "/product/Re23SF/pic",
              "render": "image"
            },
            {
              "rel": "http://www.my.co/rels/stores",
              "href": "/store/245"
            },
            {
              "rel": "http://www.my.co/rels/stores",
              "href": "/store/33"
            }
          ]
        }
      ]
    }
}
```

De nuevos vemos que se usa un array de objetos, donde cada objeto modela un enlace. Estos objetos tienen las propiedades `href` y `rel` para indicar la URI y el tipo de relación del enlace respectivamente. También pueden tener otras propiedades, como `name`, `prompt` o `render`. Esta última propiedad puede tomar como valores `link` o `image`. Indica si el enlace, en el caso de ser presentado a un humano, debe interpretarse como un enlace o como una imagen.

9.4 Consultas

Las consultas se especifican mediante el atributo `queries`, que contiene un array de objetos, cada una de ellas representa una consulta diferente. Supongamos que tenemos una colección que modela nuestro carrito de la compra, su representación en Collection+JSON podría ser la siguiente:

```
{
  "collection": {
    "href": "/shop/basket/234255",
    "items": [
      {
        "href": "/shop/basket/234255/1",
        "data": [
          {"name": "quantity", "value": 2},
          {"name": "subtotal", "value": 21.99},
```

```
          {"name": "currency", "value": "eur"}
        ],
        "links": [
          {
            "rel": "/rels/product",
            "href": "/product/43"
          }
        ]
      },
      {
        "href": "/shop/basket/234255/2",
        "data": [
          {"name": "quantity", "value": 10},
          {"name": "subtotal", "value": 1122.2},
          {"name": "currency", "value": "eur"}
        ],
        "links": [
          {
            "rel": "/rels/product",
            "href": "/product/233"
          }
        ]
      }
    ],
    "queries": [
      {
        "href": "/shop/products",
        "rel": "search",
        "prompt": "Search products",
        "data": [
          {"name": "pagesize", "value": 15, "prompt": "Results per page:"},
          {"name": "type", "value": "all", "prompt": "Type of product:"}
        ]
      }
    ]
  }
}
```

Como se observa tenemos una consulta disponible dentro de `queries`. El objeto consulta es igual que un objeto enlace pero tiene un campo extra llamado `data` con un array de criterios de búsqueda. Cada uno de estos criterios de búsqueda tiene la misma estructrua que una propiedad de un objeto: `name`, `value` y `prompt`. Collection+JSON especifica que hay que rellenar los valores de cada uno de estos campos y hacer un `GET` sobre la URI especificada en la propiedad `href` de

la consulta, añadiendo los parámetros de consulta en la *query string* de la URI. Por ejemplo si queremos buscar todos los libros, y obtener 30 resultados por página, debemos hacer un `GET` sobre `/shop/products?pagesize=30&type=book`.

9.5 Formularios

En este aspecto Collection+JSON está muy limitado, sólo nos permite indicar un único formulario para añadir un nuevo miembro a la colección. Aunque limitada, esta funcionalidad es muy útil en el caso del carrito de la compra:

```
{
  "collection": {
    "href": "/shop/basket/234255",
    "items": [
      {
        "href": "/shop/basket/234255/1",
        "data": [
          {"name": "quantity", "value": 2},
          {"name": "subtotal", "value": 21.99},
          {"name": "currency", "value": "eur"}
        ],
        "links": [
          {
            "rel": "/rels/product",
            "href": "/product/43"
          }
        ]
      },
      {
        "href": "/shop/basket/234255/2",
        "data": [
          {"name": "quantity", "value": 10},
          {"name": "subtotal", "value": 1122.2},
          {"name": "currency", "value": "eur"}
        ],
        "links": [
          {
            "rel": "/rels/product",
            "href": "/product/233"
          }
        ]
```

```
      }
    ],
    "queries": [
      {
        "href": "/shop/products",
        "rel": "search",
        "prompt": "Search products",
        "data": [
          {"name": "pagesize", "value": 15, "prompt": "Results per page:"},
          {"name": "type", "value": "all", "prompt": "Type of product:"}
        ]
      }
    ],
    "template": {
      "data": [
        {"name": "quantity", "value": 1, "prompt": "How many?"},
        {"name": "product-href", "value": "", "prompt": "Product to add:"}
      ]
    }
  }
}
```

Dicho formulario aparece bajo el campo `template`, que es un objeto con sólo un único campo `data`. De nuevo, dentro de data se incluyen un array de objetos. Cada uno de estos objetos representa un campo de datos necesario para construir un nuevo miembro de la colección. La idea es muy simple, hacer un `POST` contra la URI de la colección, enviando en el cuerpo un objeto `template` completamente relleno.

Por ejemplo, si quisiéramos añadir 4 memorias USB, haríamos lo siguiente:

```
POST /shop/basket/234255 HTTP/1.1
Host: shop.com
Accept: application/vnd.collection+json
Content-Type: application/vnd.collection+json

{
  "template": {
    "data": [
      {"name": "quantity", "value": 4},
      {"name": "product-href", "value": "/product/9wx72d"}
    ]
  }
}
```

Obsérvese que sólo mandamos un objeto JSON con un único campo `template`, que contiene los parámetros necesarios para crear una nueva línea de pedido. Collection+JSON no permite el uso de `PUT` para crear.

9.6 Conclusiones

Podemos decir que Collection+JSON tiene una aplicación limitada. El hecho de que sólo permita modelar colecciones, y no nos proporcione herramientas para modelar cualquier tipo de acciones, hace que sea difícil de utilizar en un caso de uso genérico.

Sin embargo su sencillez lo hace eficaz cuando queremos modelar colecciones simples, tales como carritos de la compra, lista de tareas, etc.

10 (X)HTML

10.1 Introducción

Tras ver en los capítulos anteriores el estado del arte en formatos *hypermedia*, conviene ahora echar un vistazo al pasado. HTML es el primer formato *hypermedia* que consiguió difusión a escala planetaria. Todos los lectores conocerán HTML en mayor y menor medida, y muchos sabreis que en principio no fue diseñado como parte de un API web, sino como un medio para representar interfaces de usuario rudimentarias accesible por internet.

Sin embargo podemos extrapolar su uso a la transmisión genérica de información entre máquinas, sólo necesitamos un poco de imaginación como veremos en el resto de este capítulo.

10.2 HTML como formato de datos

Si uno lee el estándar HTML rápidamente se dara cuenta de que existen tres tipos de etiquetas: estructurales, semánticas y de presentación. Las etiquetas de presentación, tales como `<b>` o `<i>`, no implican significado alguno y se limitan a indicar al cliente como debe mostrarse la información que contienen. Las etiquetas semánticas sí que aportan significado extra a la información que contienen, por ejemplo `<strong>` indica que el contenido es *importante*, y `<time>` que se trata de un *timestamp*. Finalmente las etiquetas estructurales, como `<p>`, `<div>`, `<li>` o `<section>`, simplementen ayudan a organizar el contenido en grupos lógicos, sin aportar mucho significado.

Por lo tanto para diseñar una API *hypermedia* con HTML debemos usar únicamente las etiquetas semánticas y estructurales, evitando siempre las de presentación. Sin embargo el número de etiquetas semánticas y su significado es muy ambiguo. Esto las hace inútiles si queremos representar conceptos de negocio en general. Lo normal es que queramos representar una entidad de nuestro dominio de aplicación y no exista una etiqueta adecuada para ello; ¿cómo hacemos para representar un producto o un carrito de la compra por ejemplo?

Microformatos

Para solucionar este problema podemos usar los atributos de las etiquetas, tales como `class`, `title` y `rel` para añadir información semántica complementaria. Por ejemplo, para representar a una persona usando HTML podríamos escribir los siguiente:

```
<div class="person">
  <div class="fullname">
    <span class="name">Pepe</span>
    <span class="familyname">Quintano</span>
  </div>
  <div class="age">33</div>
  <div class="phone">635-235-1344</div>
</div>
```

Hemos usado la etiqueta `div` ya que nos permite abrir una estructura compleja de datos, y es neutral desde el punto de vista semántico. El significado se lo hemos asignado mediante el atributo `class`, en este caso se trata de un objeto de tipo `person` con atributos `fullname`, `age` y `phone`. Además hemos descompuesto `fullname` en el nombre común (`name`) y apellido (`familyname`).

Este enfoque podría resolver el problema desde un punto de vista técnico, pero no olvidemos que nuestro objetivo es crear una API web lo más interoperable posible. La elección de los nombres a usar como valores de `class` es totalmente arbitraria, y la estructura de datos usada para representar a una persona también lo es.

Para solucionar este problema de interoperabilidad se creó una especificación llamada Microformatos[1]. La idea es crear un repositorio que incluya tanto un vocabulario común de tipos de datos como patrones de diseño para usar a la hora de crear nuevos tipos. Actualmente se encuentran especificados como definir datos de contactos siguiendo el formato hCard, productos usando hProduct, currículums usando hResume, sindicación de blogs con hAtom, y mucho más. Por cada tipo de datos se especifica que atributos hay que usar (`class`, `rel`, `title`, etc.), qué valores hay que usar para representar cada trozo de información y qué etiquetas se aconsejan.

Actualmente buscadores web, como Google, y numerosos plugins y extensiones para los navegadores, aprovechan el HTML anotado con microformatos para extraer información relevante a sus fines. Es totalmente factible usar esta técnica para exponer información en nuestra API web.

Siguiendo estas líneas, y con los ejemplos de anteriores capítulos, podemos intentar representar un producto de nuestra tienda usando microformatos y HTML de la siguiente forma:

```
<div class="hproduct">
  <span class="brand">HamCraftmen LTD.</span>
  <span class="category">High Tech Food</span>
  <span class="fn">iJam4000</span>
  <span class="description">
   The iJam4000 is the state of the art in Spanish Cured Ham.
   It has been developed with high tech pork.
   Recommended for the avant-garde castizo hipsters
  </span>
</div>
```

[1] http://en.wikipedia.org/wiki/Microformat

En este ejemplo se ha usado el microformato hProduct[2]. Simplemente usamos un `<div>` con `class="hproduct"` para marcar que todo el contenido de dicha etiqueta corresponde a una instancia de producto. Después usamos `class` para marcar los disitintos campos de información del objeto. Es un enfoque simple y bastante interoperable.

Microdata

Sin embargo el enfoque de microformatos está perdiendo fuerza en la actualidad. Por un lado muchos no confían en él, ya que es una iniciativa que no está fuertemente respaldada por ninguna organización o empresa fuerte. Por lo tanto muchos lo ven como una mera recomendación y no como un estándard. Otro problema es que se está extendiendo hasta el límite la semántica original de los atributos HTML como `class` o `title`. Si no se tiene cuidado a la hora de diseñar y/o implementar un microformato podemos terminar teniendo problemas. Quizás el caso más famoso se produjo cuando la BBC dejó de usar el microformato hCalendar[3] debido a problemas de accesibilidad[4].

Con el advenimiento de HTML5 se produjo la oportunidad de arreglar la situación, y dicha oportunidad se aprovechó para crear el estándar de HTML5 llamado Microdata[5]. Los buscadores más populares (Google, Yahoo y Bing) han decidido soportar este estándar y contribuir a él. Ellos y otras organizaciones han creado un registro estandarizado de tipos de datos[6] usando el enfoque de microdata. Actualmente estandariza incluso más tipos de datos que la iniciativa de microformatos, tales como información de contacto y organizacional, eventos, calendario, salud, producto, cocina, negocios, etc. Muchas web que antes usaban microformatos están migrando a este nuevo estándar.

Pero, ¿en qué consisten microdata? Microdata toma el testigo de los microformatos y sigue su misma filosofía: anotar las etiquetas de HTML mediante atributos con información semántica. La diferencia técnica con los microformatos es que no intenta reutilizar los atributos ya existentes para otras tareas, como `class` or `title`, sino que especifica y estandariza nuevos atributos especiales para esta tarea: `itemscope`, `itemtype` e `itemprop`. Desde un punto de vista no técnico la principal diferencia con los microformatos se encuentra en que se trata de un estándar respaldado por W3C y HTML5 y donde están invirtiendo grandes compañías.

Siguiendo estas líneas, y con los ejemplos de anteriores capítulos, podemos intentar representar un producto de nuestra tienda usando microdata y HTML. Para ello usaremos el tipo de dato Product[7], que es el que tiene la semántica adecuada. El HTML para nuestro producto quedaría como sigue:

[2] http://microformats.org/wiki/hproduct
[3] http://www.bbc.co.uk/blogs/radiolabs/2008/06/removing_microformats_from_bbc.shtml
[4] http://www.webstandards.org/2007/04/27/haccessibility/
[5] http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html
[6] http://schema.org/
[7] http://schema.org/Product

```
<div itemscope itemtype="http://schema.org/Product">
  <div itemprop="brand" itemscope itemtype="http://schema.org/Brand">
    <span itemprop="name">HamCraftmen LTD.</span>
  </div>
  <span class="category">High Tech Food</span>
  <span itemprop="name">iJam4000</span>
  <span itemprop="description">
   The iJam4000 is the state of the art in Spanish Cured Ham.
   It has been developed with high tech pork.
   Recommended for the avant-garde castizo hipsters
  </span>
</div>
```

Se observa el uso de la propiedad `itemscope` para abrir un ámbito de datos. Indica simplemente que todo el contenido dentro de la etiqueta a la que pertenece se refiere a un mismo objeto. El tipo de datos se especifica con `itemtype`, indicando la URI de la especificación de éste. Dentro del elemento HTML podemos tener contenido arbitrario, pero cada vez que uno de las etiquetas hijas contenga alguna propiedad del objeto, esta se marca con el atributo `itemprop` y el nombre de la propiedad. Si una propiedad no tiene un valor simple, podemos indicarlo añadiendo un `itemscope` y un `itemtype`, lo que nos permite incrustar un subobjeto, como es el caso de la marca del producto. En este ejemplo sólo queda `category` sin marcar, esto es debido a que la especificación actual no lo contempla como propiedad de un producto, ¿qué hacemos? Afortunadamente la especificación de microdata nos da una solución:

```
<div itemscope itemtype="http://schema.org/Product">
  <div itemprop="brand" itemscope itemtype="http://schema.org/Brand">
    <span itemprop="name">HamCraftmen LTD.</span>
  </div>
  <span itemprop="http://shop.com/product#category">High Tech Food</span>
  <span itemprop="name">iJam4000</span>
  <span itemprop="description">
   The iJam4000 is the state of the art in Spanish Cured Ham.
   It has been developed with high tech pork.
   Recommended for the avant-garde castizo hipsters
  </span>
</div>
```

Aquí ocurre algo curioso, la especificación permite usar URIs como nombres de propiedad. Si un nombre de propiedad es un identificador simple, como por ejemplo `description`, entonces dicha propiedad se interpreta en el contexto del tipo especificado en `itemtype`. Sin embargo si una propiedad es una URI, se interpreta que la semántica de esta no ha de buscarse en el tipo sino en la URI definida. Esto nos permite añadir propiedades no especificadas en el tipo `Product`, es decir,

podemos añadir propiedades propietarias o de otras especificaciones y mezclarlas libremente entre si. De esta manera usando microdata podemos gozar de las ventajas de tener tipos de datos, pero a la vez poder extender las instancias de dichos tipos con nueva información. En el ejemplo de arriba usamos `http://shop.com/product#category` para definir una nueva propiedad que no es parte del estándar pero que nosotros necesitamos.

También podemos anidar tipos de datos, por ejemplo si queremos añadir a nuestro producto información sobre las valoraciones de clientes, podríamos hacer lo siguiente:

```
<div itemscope itemtype="http://schema.org/Product">
  <div itemprop="brand"
    itemscope itemtype="http://schema.org/Brand">
    <span itemprop="name">HamCraftmen LTD.</span>
  </div>
  <span class="category">High Tech Food</span>
  <span itemprop="name">iJam4000</span>
  <span itemprop="description">
   The iJam4000 is the state of the art in Spanish Cured Ham.
   It has been developed with high tech pork.
   Recommended for the avant-garde castizo hipsters
  </span>
  <div itemprop="aggregateRating"
       itemscope itemtype="http://schema.org/AggregateRating">
   Quality: <span itemprop="ratingValue">5</span> Js
   based on <span itemprop="reviewCount">2</span> customer reviews
  </div>
  <ol>
    <li itemprop="review" itemscope itemtype="http://schema.org/Review">
      <span itemprop="name">You must try it</span> -
      by <span itemprop="author">Johny</span>,
      <time itemprop="datePublished"
            datetime="2013-01-02">Feb 1, 2013</time>
      <div itemprop="reviewRating"
           itemscope itemtype="http://schema.org/Rating">
        <meta itemprop="worstRating" content = "1">
        <meta itemprop="bestRating" content = "5">
        <span itemprop="ratingValue">5</span> Js
      </div>
      <span itemprop="description">Best ham I ever tasted!</span>
    </li>
    <li itemprop="review" itemscope itemtype="http://schema.org/Review">
      <span itemprop="name">Happy memories</span> -
      by <span itemprop="author">Beth</span>,
```

```
35        <time itemprop="datePublished"
36              datetime="2013-13-01">Jan 13, 2013</time>
37        <div itemprop="reviewRating"
38             itemscope itemtype="http://schema.org/Rating">
39          <meta itemprop="worstRating" content = "1">
40          <meta itemprop="bestRating" content = "5">
41          <span itemprop="ratingValue">5</span> Js
42        </div>
43        <span itemprop="description">
44          Remembers me of the ham my granma cooked for me.
45        </span>
46      </li>
47    </ol>
48  </div>
```

En este caso estamos incrustando información de tipo Review[8], Rating[9] y AggregateRating[10]. `AggregateRating` nos permite indicar la valoración total del producto por todos los usuarios. `Review` se refiere a la opinión de los consumidores, e incluye dentro de la propiedad `reviewRating` un objeto de tipo `Rating`. Obsérvese el uso de etiquetas `meta` para indicar los valores mínimos y máximos de la escala de puntuación.

Recomendaciones

El autor recomienda las siguientes líneas generales para representar información dentro de HTML:

- Si existe una etiqueta HTML con la semántica adecuada, úsese.
- Si no, es mejor usar el esquema de microdata que más se adapte a la información que queremos transmitir.
- Si no existe un esquema de microdata, podemos usar uno de microformato si lo hubiera
- En caso de que no exista tampoco un microformato que se adapte, es mejor crear un esquema de microdata propietario, y publicar su documentación.

10.3 Enlaces y formularios

HTML es pionero en el uso de enlaces y formularios, por lo tanto es muy sencillo y natural añadir la dimensión *hypermedia* a nuestros documentos. Veamos algunos ejemplos.

[8] http://schema.org/Review
[9] http://schema.org/Rating
[10] http://schema.org/AggregateRating

Enlaces

En HTML existen dos formas de modelar un enlace. Una es usando la etiqueta `<link>` y la otra es mediante la etiqueta `<a>`. La etiqueta `<link>` sólo puede aparecer en la cabecera del documento. Desde el punto de vista de una API web usaremos indistintamente ambas etiquetas. Sin embargo, por homogeneidad, el autor usará preferentemente `<a>`, excepto en los casos en los que la especificación lo prohiba explícitamente, como por ejemplo cuando el tipo de enlace es `rel="icon"` o `rel="stylesheet"`, o si no es necesario incluir el enlace cuando el recurso aparezca embebido dentro de otro recurso.

Siguiendo con el ejemplo de los productos y las tiendas, el siguiente HTML usa links para implementar la navegación:

```
<!DOCTYPE html>
<html lang="en" itemscope itemtype="http://schema.org/Product">
  <head>
    <link rel="self" type="text/html" href="/product/Re23SF"/>
    <link rel="http://www.my.co/rels/stores"
          type="text/html" href="/store/245"/>
    <link rel="http://www.my.co/rels/stores"
          type="text/html" href="/store/33"/>
  </head>
  <body>
    <div itemprop="brand" itemscope itemtype="http://schema.org/Brand">
      <span itemprop="name">HamCraftmen LTD.</span>
    </div>
    <span itemprop="http://shop.com/product#category">High Tech Food</span>
    <span itemprop="name">iJam4000</span>
    <span itemprop="description">
     The iJam4000 is the state of the art in Spanish Cured Ham.
     It has been developed with high tech pork.
     Recommended for the avant-garde castizo hipsters
    </span>
    <a rel="next" href="/product/3rq245">Next product</a>
    <a rel="prev" href="/product/9wx72d">Previous product</a>
    Available at the following stores:
    <ul>
      <li>
        <a rel="http://www.my.co/rels/stores"
           href="/store/245">All about ham</a>
      </li>
      <li>
        <a rel="http://www.my.co/rels/stores"
```

```
            href="/store/33">Big daddy's</a>
      </li>
    <ul>
  </body>
</html>
```

De nuevo tenemos los enlaces de navegación `next` y `prev`, y el enlace de tipo propietario `/rels/stores`, al igual que en el ejemplo de los capítulos anteriores. Es interesante es el uso de `itemscope` e `itemtype` a nivel del elemento raíz `<html>`, esto es perfectamente legal e indica que el documento HTML al completo representa un único producto.

Formularios

Por supuesto también podemos incluir acciones complejas mediante formularios HTML5. Veamos por ejemplo el carrito de la compra:

```
<!DOCTYPE html>
<html lang="en" itemscope itemtype="http://shop.com/basket">
  <head>
    <link rel="self" type="text/html" href="/shop/basket/234255"/>
  </head>
  <body>
    <div itemprop="total"
         itemscope itemtype="http://schema.org/PriceSpecification">
      <span itemprop="price">345.2</span>
      <span itemprop="priceCurrency">eur</span>
    </div>
    <ul>
      <li itemprop="item" itemscope itemtype="http://shop.com/order-item">
        <span itemprop="quantity">2</span>
        <a rel="http://shop.com/rels/product" href="/shop/products/43"/>
      </li>
      <li itemprop="item" itemscope itemtype="http://shop.com/order-item">
        <span itemprop="quantity">10</span>
        <a rel="http://shop.com/rels/product" href="/shop/products/233"/>
      </li>
    </ul>
    <section itemprop="search-form">
      <h1>Search Products</h1>
      <form name="search-products"
            type="application/x-www-form-urlencoded"
            action="/shop/products"
```

```
27              method="get">
28          <input name="pagesize" type="range" min="10" max="100" value="15">
29          <input name="type" type="radio" value="all">
30          <input name="type" type="radio" value="books">
31          <input name="type" type="radio" value="offers">
32        </form>
33      </section>
34      <section itemprop="order-form"
35               itemscope itemtype="http://shop.com/add-product">
36        <h1>Order product form</h1>
37        <form name="order-product"
38              type="application/json"
39              action="/shop/basket/234255/items/3"
40              method="post">
41          <meta itemprop="X-HTTP-Method-Override" content="put">
42          <input itemprop="product-ref" name="product-href" type="text">
43          <input name="quantity" type="range" min="1" max="20">
44        </form>
45      </section>
46    </body>
47  </html>
```

Se ha usado `itemscope` e `itemtype` para indicar que el documento completo HTML describe un carrito de la compra, que hemos modelado con un tipo de datos propietario `http://shop.com/basket`, que contiene las propiedades `total`, `item`, `search-form` y `order-form`.

La propiedad `total` indica el precio total de la compra, y para ello usa el tipo de datos estándar PriceSpecification[11].

La propiedad `item` especifica una linea de pedido, que debe ser de tipo `http://shop.com/order-item`, indicando un enlace al producto y cuantas unidades de dicho producto se van a comprar. Pueden existir varias ocurrencias de `item`, en cuyo caso esta propiedad tomará como valor una lista de lineas de pedido.

La propiedad `search-form` es interesante ya que contiene un formulario. En este se incluyen los controles que necesitamos para indicar todos los parámetros de búsqueda: el tamaño de paginación (por defecto a 15), y que tipo de productos queremos buscar. El tamaño de paginación lo indicamos mediante un control de tipo `range`, y el tipo de productos mediante un grupo de controles `radio`.

El formulario `order-form` es más interesante. Este formulario ha sido marcado como un objeto de tipo `http://shop.com/add-product`. En HTML los fomularios sólo pueden tener en el atributo `method` los valores `get` y `post`, sin embargo nosotros necesitamos usar `PUT` para poder implementar el patrón de "recurso virtual y URI desechable". Otra limitación de los formularios HTML es que no tienen

[11] http://schema.org/PriceSpecification

controles de tipo "URI", lo más parecido es tipo `text`. Debido a estas dos limitaciones necesitamos un tipo propietario que añada una capa extra de semántica. Por un lado nos permite añadir una propiedad `X-HTTP-Method-Override` que nos permite incluir la cabecera `X-HTTP-Method-Override` en la llamada `POST` realizada por el formulario, usando `PUT` como valor para dicha cabecera. Esta cabecera será consumida por el servidor y ejecutará la petición usando el método especificado en la cabecera `X-HTTP-Method-Override`, en vez de usar `POST`. Este "truco" empezó a ser usado por Google en su API[12]. La otra capa extra de semántica se implementa mediante la propiedad `product-ref`, que según la semántica especificada en `http://shop.com/add-product` debe se de tipo `URI`. Esto hace que el cliente deba añadir una capa extra de validación al control HTML para comprobar si el texto es una URI bien formada o no.

10.4 Conclusiones

El uso de HTML como formato hypermedia está vivo y goza de muy buena salud. No hay que olvidar que la gran mayoría de las páginas web están en HTML y contienen información valiosa. Sin embargo existen varios problemas:

- HTML está diseñado para transmitir documentos que van a ser consumidos por humanos. Esto hace que HTML sea excesivamente verboso y muchas veces algo ineficiente. Pero lo más grave es que no es sencillo hacer que una máquina extraiga información de un documento HTML.
- HTML es costoso de parsear, ya que un documento HTML no tiene por qué ser un documento XML bien formado.

Sin embargo estos problemas pueden ser mitigados. Empecemos por el último. La nueva especificación HTML5 asigna dos tipos mime distintos a HTML, el primero es el tradicional `text/html`, y el otro es `application/xhtml+xml`. El primero sí es difícil de consumir, pero el segundo exige que el documento sea un XML bien formado. Por lo tanto a nivel de diseño de APIs web, podemos decidir sólo soportar `application/xhtml+xml`, ya que es más eficiente desde el punto de vista de la comunicación máquina a máquina.

El primer problema, que HTML está diseñado para presentar información a humanos y no a máquinas, puede ser resuelto mediante el estándar de microdata. Mediante el uso de vocabularios estandarizados de microdata, y de tipos de datos propietarios en los casos que realmente se necesite, se pueden generar documentos HTML5 fácilmente consumibles no sólo por humanos, sino también por máquinas. También existen otras iniciativas en la misma línea, tales como Dublic Core Metadata[13], y como vimos anteriormente los microformatos, pero el estándar de microdata es el que se está imponiendo.

Que nuestra API soporte no sólo JSON o XML, sino también HTML, mediante el uso de microformatos y el tipo mime `application/xhtml+xml` tiene algunas ventajas importantes:

[12] https://developers.google.com/gdata/docs/2.0/basics#UpdatingEntry

[13] http://dublincore.org/documents/dc-html/

- Si un usuario humano usa su navegador para consumir nuestra API, no va a recibir un documento JSON, sino una página perfectamente usable. Tal vez no sea una página muy bonita, pero si perfectamente accesible. Quizás con un poco de javascript esa página tan fea pueda ser transformada en una UI potente mediante la técnica de *progressive enhancement o HIJAX*[14]
- En la línea del punto anterior, servir HTML puede ser útil como herramienta de depuración y autodocumentación. ¿Qué mejor forma de que nuestros potenciales clientes aprendan nuestra API que usándola directamente con sus browsers?
- Finalmente, el usar HTML puede activar una posibilidad bastante curiosa, que los buscadores de internet más importantes del mundo (Google, Yahoo, Bing...) puedan indexar el contenido de nuestra API web. ¿No les parece esto una oportunidad interesante para ganar usuarios de sus APIs?

Para finalizar, no deja de ser irónico el hecho de que HTML, un protocolo completamente hipermedia desde sus orígenes, tenga que ser extendido con anotaciones para convertirlo en un contenedor de datos aceptable, y que a su vez, formatos que no tienen problemas para transportar datos, como XML o JSON, tengan que ser extendidos para que soporten hipermedia.

[14] http://en.wikipedia.org/wiki/Progressive_enhancement

11 Atom y AtomPub

11.1 Introducción

Otro de los protocolos pioneros en el mundo de las APIs *hypermedia* es Atom. En realidad Atom tiene dos especificaciones: "The Atom Syndication Format" o RFC4287[1] y "The Atom Publishing Protocol" o RFC5023[2]. La RFC4287 especifica el formato del tipo mime que se usa principalmente en Atom: `application/atom+xml`. Por contra la especificación RFC5023, más comunmente llamada *AtomPub*, se centra en como usar el protocolo HTTP junto con el tipo mime `application/atom+xml`.

AtomPub es una especificación realmente interesante. Si el lector se anima a leerla, se dará cuenta que principalmente define dos cosas:

- Dos tipos mime auxiliares `application/atomcat+xml` y `application/atomsvc+xml`, y cómo se relacionan estos con `application/atom+xml`.
- Las reglas REST para el uso de estos tipos mimes. Cómo se representan los enlaces dentro de éstos, y como usar dichos enlaces para realizar todas las operaciones permitidas por el dominio de aplicación de éste.

Es una lectura interesante porque AtomPub sirve en si mismo como un curso sobre qué es una API *hypermedia* y como usarla con HTTP. De hecho constituye un ejemplo útil de como se debería documentar una API *hypermedia*. Las especificación es concisa, ya que toma por sentado todas las reglas de REST y patrones de diseño básicos que hemos visto en este libro, y se limita a clarificarlas y concretarlas para el caso concreto de Atom.

Atom/AtomPub es otro caso de API *hypermedia* que ha alcanzado una difusión masiva a escala planetaria. Se centra en resolver el problema de como publicar y sindicar contenidos y noticias provenientes de distintos blogs y páginas de noticias, de forma que el usuario reciba notificaciones con los contenidos más recientes y pueda explorar en busca de contenidos más antiguos. Todos los navegadores y aplicaciones de agregación de noticias (o *readers*) lo soportan, y la mayoría de ustedes lo habrán usado cuando se subscriben a un blog o un feed de noticias.

A pesar de que se desarrolló para este dominio concreto de problemas, muchos autores defienden su uso en APIs que no tienen nada que ver con la subscripción a blogs y fuentes de noticias.

11.2 Servicios y autodescubrimiento

AtomPub es un protocolo *hypermedia* y por lo tanto uno de sus principales objetivos es que el consumidor sólo necesite conocer un único punto de entrada al sistema. A partir de ahí el cliente

[1] http://tools.ietf.org/html/rfc4287

[2] http://tools.ietf.org/html/rfc5023

podrá navegar por el sistema, descubriendo por si mismo, sin necesidad de documentación extra, el conjunto de funcionalidades que el servidor le ofrece.

Supongamos que el punto de entrada de nuestra API es `http://shop.com/services`. El cliente sólo debe hacer un `GET` pidiendo el documento de servicios, que debe tener el tipo mime `application/atomsvc+xml`. Veámoslo:

```
GET /services HTTP/1.1
Host: shop.com
Accept: application/atomsvc+xml
Authorization: Basic Y29udHJhc2XxYTpzZWNyZXRh

```

A lo que el servidor contestaría con el siguiente documento:

```
HTTP/1.1 200 Ok
Content-Type: application/atomsvc+xml
Date: Tue, 27 Dec 2012 05:25:19 GMT
Expires: Tue, 27 Dec 2012 11:25:19 GMT
Cache-Control: max-age=21600
Last-Modified: Tue, 27 Dec 2012 03:25:19 GMT

<?xml version="1.0" encoding='utf-8'?>
<service xmlns="http://www.w3.org/2007/app"
         xmlns:atom="http://www.w3.org/2005/Atom">
  <workspace>
    <atom:title>Shop</atom:title>
    <collection href="http://shop.com/product">
      <atom:title>Products in this Shop</atom:title>
      <accept>application/json</accept>
    </collection>
    <collection href="http://shop.com/store">
      <atom:title>Our "real world" stores</atom:title>
    </collection>
  </workspace>
  <workspace>
    <atom:title>Your account</atom:title>
    <collection href="http://shop.com/basket">
      <atom:title>Your shopping baskets</atom:title>
    </collection>
    <collection href="http://shop.com/info">
      <atom:title>Your personal & payment info</atom:title>
    </collection>
```

```
    <collection href="http://shop.com/avatar">
      <atom:title>Avatar pictures</atom:title>
      <accept>image/png</accept>
      <accept>image/jpeg</accept>
      <accept>image/gif</accept>
    </collection>
  </workspace>
</service>
```

En este documento de servicios aparece la lista de todos los `workspaces` disponibles. Un `workspace` no es más que una agrupación lógica de colecciones. Dentro de cada workspace aparecen las colecciones que pertenecen a cada uno de estos workspaces.

A nivel de colección podemos especificar información sobre ésta, tales como una descripción destinada a consumo humano dentro del elemento `atom:title`, o que tipos mime soporta cada colección. Para esto último usaremos la etiqueta `accept` que es opcional. Si no la especificamos se sobreentiende que el tipo mime a usar por defecto es `application/atom+xml;type=entry`.

Pero la información más vital aparece en el atributo `href`, donde se informa obligatoriamente de la URI que hay que usar para operar con la colección. De esta manera, podemos pensar en la etiqueta `collection` como una especie de enlace con capacidad para metainformación. Si queremos acceder a los datos de la colección usaremos la URI disponible en `href`.

11.3 Feeds y entries

Accediendo a los datos de una colección

En el ejemplo anterior descubríamos las URLs de las colecciones a traves del documento de servicios. ¿Cómo podemos usar esta información para acceder, por ejemplo, a la colección de productos? Basta con hacer un `GET` sobre el valor de `href` de la colección de productos y pedir el tipo mime `application/atom+xml;type=feed`. Veámoslo:

```
GET /product HTTP/1.1
Host: shop.com
Accept: application/atom+xml;type=feed

```

Y el servidor responde con un documento de tipo `feed`:

```
HTTP/1.1 200 Ok
Content-Type: application/atom+xml;type=feed

<?xml version="1.0" encoding='utf-8'?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Products in this Shop (1st page)</title>
  <updated>2012-12-17T03:29:29Z</updated>
  <id>urn:uuid:4c9611f7-9397-4ed2-8ca6-d03d3e8c3da6</id>
  <link rel="self" href="http://shop.com/product"/>
  <link rel="first" href="http://shop.com/product"/>
  <link rel="next" href="http://shop.com/product?page=2"/>
  <link rel="last" href="http://shop.com/product?page=32"/>
  <entry>
    <title>iJam4000</title>
    <id>urn:uuid:fcb5bbec-e1f5-4a23-a8d9-554aeeb03d10</id>
    <updated>2012-12-17T03:29:29Z</updated>
    <summary type="text"/>
    <content type="application/json"
             src="http://shop.com/product/43.json"/>
    <link rel="edit-media"
          href="http://shop.com/product/43.json"/>
    <link rel="edit"
          href="http://shop.com/product/43.atom"/>
  </entry>
  <entry>
    <title>A fantastic book!</title>
    <id>urn:uuid:23275f39-7b7a-455f-b665-17203992296c</id>
    <updated>2012-12-13T18:30:02Z</updated>
    <summary type="text"/>
    <content type="application/json"
             src="http://shop.com/product/9wx72d.json"/>
    <link rel="edit-media"
          href="http://shop.com/product/9wx72d.json"/>
    <link rel="edit"
          href="http://shop.com/product/9wx72d.atom"/>
  </entry>
</feed>
```

Básicamente un documento de tipo `feed` consta de metainformación, enlaces y un conjunto de entradas. En el ejemplo se muestra un `feed` donde se especifica el título mediante la etiqueta `title`, la fecha de la última actualización mediante la etiqueta `updated` y un identificador único mediante la etiqueta `id`. Estas tres etiquetas son obligatorias. Llama la atención la etiqueta `id`. En REST el identificador de un recurso es la URI del documento, pero Atom obliga a especificar una URI como

id que no tiene porqué coincidir con la URI del documento. La idea detrás de este diseño es que este `id` sea un identificador lógico que no esté asociado a una dirección física concreta, y que podamos referenciar aunque movamos los recursos de servidor. Sin embargo ya se discutión en anteriores capítulos, que mediante una configuración adecuada de DNS, y si es necesario, el uso de redirecciones, podemos mover los recursos de un servidor a otro de forma transparente sin necesidad de artificios adicionales. Por lo tanto podríamos poner en esta etiqueta `id` la URI del recurso en si. Existen otras muchas etiquetas para añadir a las colecciones, tales como `author`, `rights`, `subtitle`, etc. Estas etiquetas se pensaron para el caso concreto de que nuestra colección represente una fuente de noticias y/o entradas de un blog. En este libro el autor está más interesado en el uso general de Atom para cualquier tipo de API web, con lo que no se centrará en este tipo de detalles.

Cada `feed` puede contener un conjunto de entradas, cada una de ellas representada por una etiqueta `entry`. Por defecto estas entradas se ordenan de la más reciente o recientemente actualizada a la más antigua. Atom no nos propone ningún mecanismo para producir otra ordenación diferente, pero no nos prohíbe inventar alguna forma de hacerlo. Si en la colección existieran demasiadas entradas, Atom nos propone usar enlaces, mediante la etiqueta `link`, con tipos de relación `next`, `previous`, `first` y `last` en exactamente la misma manera que vimos en capítulos anteriores.

11.4 Media Resources VS. Media Entries

En Atom existen dos tipos diferentes de entradas: *media resources* y *media entries*.

Una *media entry* representa una entrada de un blog y/o una noticia. Está modelado expresamente mediante el tipo mime `application/atom+xml;type=entry`. Veamos un ejemplo:

```
<?xml version="1.0" encoding='utf-8'?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>Te lo dije...</title>
  <id>urn:uuid:11c80983-9d41-4ab1-9fff-3f98e444b5bb</id>
  <updated>2010-09-27T15:00:00Z</updated>
  <published>2010-09-27T15:00:00Z</published>
  <author>
    <name>Enrique Amodeo</name>
    <uri>http://eamodeorubio.wordpress.com</uri>
    <email>eamodeorubio@gmail.com</email>
  </author>
  <content type="xhtml" xml:lang="en"
           xml:base="http://eamodeorubio.wordpress.com">
    <div xmlns="http://www.w3.org/1999/xhtml">
      <h1>Servicios web (5): Diseñando servicios REST (3) HATEOAS</h1>
      <p>.....</p>
    </div>
```

```
18    </content>
19    <link rel="edit"
20          href="http://bit.ly/9Q9qFu.atom"/>
21  </entry>
```

En este tipo de entradas podemos especificar información muy útil para un blog o un agregador de noticias, como información sobre el autor, las fechas de publicación o actualización, etc. Incluso podemos, mediante la etiqueta `content` incrustar el contenido de la entrada. En este caso se ha usado formato XHTML para poner el contenido, pero podría usarse HTML, texto plano o XML. Incluso se puede especificar un tipo mime arbitrario (aunque en este último caso habría que codificar en base 64 el contenido de la entrada). Sin embargo lo más interesante es el enlace con tipo de relación `edit`. La URI de este enlace es la que podremos utilizar para actualizar y borrar la entrada usando `PUT` y `DELETE` respectivamente.

Si estamos construyendo una API que exponga noticias, entradas de blog, o algo similar, el usar *media entry* es muy aconsejable ya que se adapta a nuestro domínio y no necesitamos inventar un tipo mime propietario. Pero en general la mayoría de las APIs no se encuentran dentro de este dominio de problema. Si queremos usar Atom para hacer una API de cualquier dominio, entonces *media entry* es demasiado específico.

Para solucionar este problema podemos usar el patrón *envelope* explicado en el capítulo de *hypermedia*. En AtomPub tenemos el concepto de *media link entry*, que no es más que la aplicación del patrón *envelope* para construir cualquier tipo de recursos (o *media resources* en la terminología de AtomPub). Un *media resource* está representado en un sistema Atom mediante dos URIs. La primera es la URI en si de la representación del recurso. La segunda es la URI de un *media entry* especial, llamado *media link entry*, que contiene metainformación sobre el recurso y un enlace directo a este. La idea es que las colecciones unicamente pueden contener entradas de tipo *media entry* (`application/atom+xml;type=entry`). Por ello necesitamos usar estas entradas de tipo *media link entry* para envolver al recurso que realmente queremos. En la sección anterior vimos como la colección de productos contenía dos *media link entry*. Más adelante veremos un ejemplo concreto.

Ya sea para crear un `media entry` o un `media resource`, el tipo mime usado debe ser soportado por el servidor. Esto no es problema para `media entry` ya que un servidor Atom debe soportar siempre `application/atom+xml;type=entry`. Sin embargo, ¿cómo sabemos que tipos mime podemos mandar si queremos crear cualquier tipo de recurso? Es sencillo, en el documento de servicio, dentro de la etiqueta `collection` pertinente, pueden aparecer las etiquetas `accept` indicando los tipos mime que son aceptables. Si no aparecen sólo podremos crear `media entry`.

11.5 Manipulando elementos de una colección

Creando y editando una entrada de blog: *media entry*

Para añadir una nueva entrada a una colección, nos basta con hacer POST a la URI que obtuvimos dentro del documento de servicios. Por ejemplo para añadir una nueva *media entry*:

```
POST /entries HTTP/1.1
Host: blog.com
Content-Type: application/atom+xml;type=entry
Slug: Servicios Web REST HATEOAS

<?xml version="1.0" encoding='utf-8'?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>Te lo dije...</title>
  <id>urn:uuid:11c80983-9d41-4ab1-9fff-3f98e444b5bb</id>
  <updated>2010-09-27T15:00:00Z</updated>
  <author>
    <name>Enrique Amodeo</name>
    <uri>http://blog.com/eamodeo</uri>
    <email>eamodeorubio@gmail.com</email>
  </author>
  <content type="xhtml" xml:lang="en"
           xml:base="http://eamodeorubio.wordpress.com">
    <div xmlns="http://www.w3.org/1999/xhtml">
      <h1>Servicios web (5): Diseñando servicios REST (3) HATEOAS</h1>
      <p>.....</p>
    </div>
  </content>
</entry>
```

Observese que usamos el tipo mime `application/atom+xml;type=entry`. Interesante también la cabecera `Slug` que viene definida en la propia especificación AtomPub. Esta cabecera nos sirve para darle un conjunto de palabras clave al servidor, de forma que éste asigne una URI al nuevo recurso que contenga dichas palabras claves. En este caso el servidor respondería con:

```
HTTP/1.1 201 Created
Content-Type: application/atom+xml;type=entry
Location: http://blog.com/2010/09/27/servicios-web-rest-hateoas-3.atom

<?xml version="1.0" encoding='utf-8'?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>Te lo dije...</title>
  <id>urn:uuid:11c80983-9d41-4ab1-9fff-3f98e444b5bb</id>
  <updated>2010-09-27T15:00:00Z</updated>
  <published>2010-09-27T15:00:00Z</published>
  <author>
    <name>Enrique Amodeo</name>
    <uri>http://blog.com/eamodeo</uri>
```

```
    <email>eamodeorubio@gmail.com</email>
  </author>
  <content type="xhtml" xml:lang="en"
           xml:base="http://eamodeorubio.wordpress.com">
    <div xmlns="http://www.w3.org/1999/xhtml">
      <h1>Servicios web (5): Diseñando servicios REST (3) HATEOAS</h1>
      <p>.....</p>
    </div>
  </content>
  <link rel="edit"
        href="/2010/09/27/servicios-web-rest-hateoas-3.atom"/>
</entry>
```

En la cabecera `Location` aparece la URI de la nueva entrada. Este es exactamente el mismo patrón de creación que vimos en los capítulos anteriores. Si hacemos un `GET` sobre dicha URI obtenemos el documento de la entrada, pero en este caso el servidor lo ha servido incrustado en la propia respuesta. Obsérvese que de nuevo podemos usar la URI del enlace de tipo `edit` para actualizar o borrar la entrada usando los métodos `PUT` y `DELETE` respectivamente.

Creando y editando cualquier tipo de recurso: *media resources*

Es mucho más interesante crear un nuevo *media resource*, es decir un recurso cualquiera, no necesariamente una entrada de un blog. Supongamos que queremos crear un nuevo producto. Para ello haremos un `POST` a la URI de la colección de productos enviando un documento JSON:

```
POST /prduct HTTP/1.1
Host: shop.com
Content-Type: application/json
Slug: iJam 4000

{
  "brand":"HamCraftmen LTD.",
  "price": {
    "quantity":148.3
    "currency":"eur"
  },
  "type":"High Tech Food",
  "name":"iJam4000",
  "description":"The state of the art in Spanish Cured Ham..."
}
```

El servidor responde con:

```
HTTP/1.1 201 Created
Content-Type: application/atom+xml;type=entry
Location: http://shop.com/product/ijam-4000.atom

```

En este caso en la cabecera `Location` aparece una URI que tiene pinta de ser una entrada. Podemos hacer un `GET` al valor de la cabecera `Location` y obtenemos:

```
<?xml version="1.0" encoding='utf-8'?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <id>urn:uuid:fcb5bbec-e1f5-4a23-a8d9-554aeeb03d10</id>
    <updated>2012-12-17T03:29:29Z</updated>
    <summary type="text"/>
    <content type="application/json"
             src="/product/ijam-4000.atom"/>
    <link rel="next"
          href="/product/ijam-3000.json"/>
    <link rel="prev"
          href="/product/book-fantastic.json"/>
    <link rel="order-form"
          href="/basket/24142.atom?product-href=ijam-4000&quantity=1"/>
    <link rel="edit-media"
          type="application/json"
          href="/product/ijam-4000.json"/>
    <link rel="edit-media"
          type="application/xml"
          href="/product/ijam-4000.xml"/>
    <link rel="edit"
          href="/product/ijam-4000.atom"/>
</entry>
```

¿Qué ha pasado? Esto no es el producto que enviamos con la petición `POST`. Lo que ha hecho el servidor ha sido crear dos recursos distintos. Uno es el *media link entry* y el otro es el recurso producto que realmente nos interesa. Sin embargo como respuesta del `POST` nos devuelve la URI del *media link entry* y no del recurso producto.

Podemos usar este *media link entry* para consultar y actualizar la metainformación que nos interese y sea pertinente, como la fecha de actualización, un título descriptivo, quien creó el producto (`author`) en el sistema, etc. Sin embargo lo realmente interesante es que podemos usar los enlaces que están en *media link entry* para seguir haciendo operaciones sobre el producto. Por ejemplo, podemos usar los enlaces de tipo `next` y `prev` para acceder al siguiente y al anterior producto respectivamente. O podríamos seguir el enlace `order-form` para llegar a un formulario que nos permita añadir este producto al carrito. De nuevo estamos ante el concepto *hypermedia*. Mediante un uso inteligente

de los *media link entry*, AtomPub soluciona el problema de que muchos tipos mime no soportan ni controles ni enlaces *hypermedia*. Y por supuesto podemos usar el enlace `edit` para actualizar (`PUT`) dicha metainformación o borrar el recurso (`DELETE`)

Esto está muy bien, pero, ¿cómo accedemos a los datos del producto en si? ¿Cómo los modificamos? Para leer los datos del producto basta con hacer un `GET` a la dirección especificada en el atributo `src` de la etiqueta `content`. Otra opción sería usar la URI especificada en el enlace `edit-media`. En principio ambas URIs no tienen que ser la misma. De hecho la URI que aparece en `content` puede ser de una versión resumida o que está alojada en una cache o CDN, y por lo tanto potencialmente obsoleta (aunque mucho más rápida de acceder). Por lo tanto la práctica segura es usar el enlace `edit-media`. En cualquier caso siempre deberemos usar `edit-media` para modificar (`PUT`) o borrar (`DELETE`) el contenido.

Un punto interesante es que podemos tener varios enlaces `link` de tipo `edit-media`. Si esto ocurriera todos deben ser distinguibles mediante el atributo `type` o `hreflang`. La idea es que podemos tener varios links a distintas versiones del recurso, cada una con distinto tipo mime o recurso.

Un detalle que hay que aclarar, si queremos borrar el producto debemos hacer un `DELETE` contra la URI del enlace `edit-media`. Esto borrará tanto el recurso como el *media link entry*. Sin embargo si hacemos un `DELETE` contra la URI especificada en el enlace `edit`, el servidor borrará el *media link entry*, pero no está obligado a borrar el recurso producto en si.

11.6 Conclusiones

AtomPub es un buen ejemplo de protocolo *hypermedia*. Hace uso de los patrones "CRUD" y *envelope* que vimos en los capítulos anteriores, pero va más allá del enfoque orientado a datos y nos proporciona una formato *hypermedia* en el que podemos seguir los enlaces para descubrir las distintas acciones y documentos que nos ofrece el sistema.

Sin embargo AtomPub tiene algunos problemas.

- Algunos pueden decidir que XML es demasiado pesado de procesar y demasiado verboso, y prefieran formatos más ligeros como JSON.
- Se diseño originalmente para sindicar blogs y noticias. Aunque como hemos visto sí que podemos usarlo para otras aplicaciones más genéricas, a veces se hace demasiado pesado. Si usamos por ejemplo JSON para transportar información, lo único que vamos a necesitar realmente de las *media link entry* es su capacidad para modelar enlaces. Sin embargo AtomPub nos fuerza a incluir las etiquetas `title`, `summary` y `content` que normalmente no nos interesan, ya que toda esa información irá en el propio documento JSON. Además la etiqueta `content` genera una fuente de confusión a la hora de leer el recurso en si, ¿qué usar, `content` o el enlace `edit-media`?
- Usa `POST` para crear nuevas entradas. Como se vio en capítulos anteriores esto puede tener como consecuencia que se dupliquen entradas en las colecciones.

- Al igual que HAL, Atom sólo modela enlaces. No especifica en absoluto como modelar otro tipo de controles, y deja en manos del desarrollador el definir un formato de formularios apropiado.

AtomPub es idóneo para el dominio de aplicación para el que se creó, pero puede ser contraproducente en otro tipo de APIs. De hecho Twitter ha dejado de soportar Atom en su nueva API. Sin embargo, debido a su difusión, es un protocolo muy interesante desde el punto de vista de la interoperabilidad. Existen numerosas aplicaciones y múltiples librerías que consumen Atom.

12 Referencias y bibliografía

Si quieres indagar más sobre REST aquí tienes una lista de recursos que han servido para hacer este libro.

HTTP y REST

- El articulo original sobre REST: http://bit.ly/R2Wi
- Hyper Text Transfer Protocol 1.1: http://tools.ietf.org/html/rfc2616
- Aclaración sobre la semántica de los métodos HTTP: http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-18
- Algunos valores de 'rel' estándarizados: http://www.iana.org/assignments/link-relations/link-relations.xml
- Autenticación "Basic" y "Digest" de HTTP: http://tools.ietf.org/html/rfc2617
- PATCH Method for HTTP: http://tools.ietf.org/html/rfc5789
- Uniform Resource Identifier: http://www.ietf.org/rfc/rfc3986.txt
- Especificación URI Templates: http://tools.ietf.org/html/rfc6570
- Algunas implementaciones de URI Templates: http://bit.ly/Kk7ySu
- Web Linking standard: http://tools.ietf.org/html/rfc5988
- WADL: http://wadl.java.net/

Tipos mime

- Directorio de tipos MIME estandarizados por IANA: http://www.iana.org/assignments/media-types/index.html
- An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors: http://tools.ietf.org/html/rfc5261
- JavaScript Object Notation (JSON):
- JSON Patch: http://tools.ietf.org/html/draft-ietf-appsawg-json-patch-01
- Hypertext Application Language: http://stateless.co/hal_specification.html y http://bit.ly/TQZwqM
- SIREN: https://github.com/kevinswiber/siren
- Collection+JSON: http://amundsen.com/media-types/collection/
- The Atom Syndication Format (RFC4287): http://tools.ietf.org/html/rfc4287
- The Atom Publishing Protocol (RFC5023): http://tools.ietf.org/html/rfc5023
- HTML5: http://www.w3.org/html/wg/drafts/html/master/
- Microformatos HTML: http://en.wikipedia.org/wiki/Microformat
- Microdata: http://www.whatwg.org/specs/web-apps/current-work/multipage/microdata.html
- Tipos de datos estandarizados para microdata: http://schema.org/Product

General

- La pila OSI de protocolos de red: http://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.200-199407-I!!PDF-E&type=items y también http://en.wikipedia.org/wiki/OSI_model
- Concurrencia optimista: http://en.wikipedia.org/wiki/Optimistic_concurrency_control y también http://www.w3.org/1999/04/Editing/
- Hash based Message Authentication Code: http://en.wikipedia.org/wiki/Hash-based_message_authentication_code

SOAP y WS-*

- Basic Profile Version 1.0: http://www.ws-i.org/profiles/BasicProfile-1.0-2004-04-16.html
- WSDL 2.0: http://www.w3.org/2002/ws/desc/