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 **i**

Agradecimientos **ii**

Érase una vez... **iii**

1 APIs orientadas a datos: CRUD **1**

1.1 Introducción 1

1.2 Leyendo 2

1.3 Actualizando 9

1.4 Borrando 11

1.5 Creando 12

1.6 Seguramente CRUD no sea lo mejor para tu API... 15

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.

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 APIs orientadas a datos: CRUD

1.1 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.

1.2 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.

1.3 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

```

1.4 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.

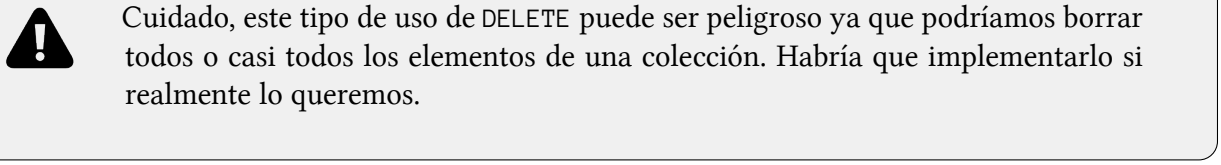
Hay que tener en cuenta que borrar una entidad, debe involucrar un borrado en cascada en todas las entidades hijas. De la misma forma, si borramos una colección se deben borrar todas las entidades que pertenezcan a ella.

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*.



1.5 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.

1.6 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.