**EBook Gratis**

# APRENDIZAJE

# rest

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#rest

# Tabla de contenido

# Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: rest

It is an unofficial and free rest ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official rest.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# Capítulo 1: Empezando con el descanso

## Observaciones

Esta sección proporciona una descripción general de qué es el resto y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema importante dentro del resto, y vincular a los temas relacionados. Dado que la Documentación para el descanso es nueva, es posible que deba crear versiones iniciales de los temas relacionados.

## Examples

### Resumen de REST

RESTO significa **RE** de presentación **S** tate **T** ransferencia y fue acuñado por **Roy Fielding** en su tesis doctoral de estilos arquitectónicos y el diseño de arquitecturas de software basado en la red . En él identifica principios arquitectónicos específicos como:

- Recursos direccionables: la abstracción clave de información y datos en REST es un recurso y cada recurso debe ser direccionable a través de un URI.
- Una interfaz uniforme y restringida: uso de un pequeño conjunto de métodos bien definidos para manipular nuestros recursos.
- Orientado a la representación: un recurso al que hace referencia un URI puede tener diferentes formatos y diferentes plataformas necesitan diferentes formatos, por ejemplo, los navegadores necesitan HTML, JavaScript necesita JSON y las aplicaciones Java pueden necesitar XML, JSON, CSV, texto, etc. Así que interactuamos con los servicios. utilizando la representación de ese servicio.
- Comuníquese sin estado: las aplicaciones sin estado son más fáciles de escalar.
- Hipermedia como el motor del estado de la aplicación: permita que nuestros formatos de datos conduzcan transiciones de estado en nuestras aplicaciones.

El conjunto de estos principios arquitectónicos se llama REST. Los conceptos de REST están inspirados en el de HTTP. Roy Fielding, quien nos dio REST, también es uno de los autores de las especificaciones HTTP.

Los servicios web y los servicios web RESTful son servicios que están expuestos a Internet para el acceso programático. Son api en línea a las que podemos llamar desde nuestro código. Hay dos tipos de servicios web "Grandes" de servicios web SOAP y REST.

Servicios web RESTful : los servicios web que se escriben aplicando los conceptos arquitectónicos de REST se denominan servicios web RESTful, que se centran en los recursos

del sistema y en cómo el estado de un recurso se puede transferir a través de un protocolo HTTP a diferentes clientes.

Este documento se centra exclusivamente en los servicios web RESTful, por lo que no entraremos en los detalles de SOAP WS.

No hay reglas estrictas al diseñar servicios web RESTful como

- Sin protocolo estándar
- Sin canal de comunicación estándar.
- Sin definición de servicio estándar

Pero SOAP tiene reglas estrictas para todo esto. Todos los servicios web de SOAP siguen la especificación de SOAP, que determina qué deben ser los servicios web de SOAP. Esta especificación es desarrollada y administrada por un comité y si SOAP WS no sigue una sola regla, entonces, por definición, no es SOAP.

***Conceptos de Servicios Web RESTful***

Hay algunas pautas que deben tenerse en cuenta al diseñar / desarrollar la API RESTful:

1. Ubicaciones basadas en recursos / URI
2. Uso adecuado de los métodos HTTP
3. HATEOAS (hipermedia como el motor del estado de la aplicación)

El enfoque principal al desarrollar las API RESTful debe ser hacer que la API sea "lo más RESTful posible".

## REST sobre HTTP

REST es una arquitectura sin **protocolo** propuesta por Roy Fielding en su disertación (el capítulo 5 es la presentación de REST), que generaliza el concepto comprobado de los navegadores web como clientes para separar a los clientes en un sistema distribuido de los servidores.

Para que un servicio o API sea RESTful, debe cumplir con restricciones dadas como:

- Servidor de cliente
- Apátrida
- Cacheable
- Sistema de capas
- Interfaz uniforme
    - Identificación de recursos
    - Representación de recursos
    - Mensajes auto-descriptivos
    - Hipermedia

Además de las restricciones mencionadas en la disertación de Fielding, en su blog, las *API REST deben ser controladas por hipertexto* , Fielding aclaró que **solo invocar un servicio a través de HTTP no lo hace REST** . Por lo tanto, un servicio también debe respetar otras reglas que se

resumen a continuación:

- La API debe cumplir y no violar el protocolo subyacente. Aunque la REST se usa a través de HTTP la mayor parte del tiempo, no está restringido a este protocolo.
- Fuerte enfoque en los recursos y su presentación a través de los tipos de medios.
- Los clientes no deben tener conocimiento o suposiciones iniciales sobre los recursos disponibles o su estado devuelto ( recurso "escrito" ) en una API, sino aprenderlos sobre la marcha a través de solicitudes emitidas y respuestas analizadas. Esto le da al servidor la oportunidad de moverse o cambiar el nombre de los recursos fácilmente sin interrumpir la implementación de un cliente.

## Modelo de Madurez de Richardson

El modelo de madurez de Richardson es una forma de aplicar restricciones REST a través de HTTP para obtener servicios web RESTful.

Leonard Richardson dividió las aplicaciones en estas 4 capas:

- Nivel 0: uso de HTTP para el transporte.
- Nivel 1: uso de URL para identificar recursos
- Nivel 2: uso de verbos y estados HTTP para las interacciones
- Nivel 3: uso de HATEOAS

Como la atención se centra en la representación del estado de un recurso, se recomienda el uso de múltiples representaciones para el mismo recurso. Por lo tanto, una representación podría presentar una visión general del estado del recurso, mientras que otra muestra los detalles completos del mismo recurso.

Tenga en cuenta también que, dadas las restricciones de Fielding, **una API es efectivamente REST completa solo una vez que se implementa el tercer nivel de RMM** .

---

# Solicitudes y respuestas HTTP

Una solicitud HTTP es:

- Un verbo (método aka), la mayoría de las veces uno de GET , POST , PUT , DELETE o PATCH
- Una URL
- Encabezados (pares clave-valor)
- Opcionalmente un cuerpo (también conocido como carga útil, datos)

Una respuesta HTTP es:

- Un estado, la mayoría de las veces uno de 2xx (correcto) , 4xx (error del cliente) o 5xx (error

del servidor)
- Encabezados (pares clave-valor)
- Un cuerpo (también conocido como carga útil, datos)

Características de los verbos HTTP:

- Verbos que tienen un cuerpo: `POST` , `PUT` , `PATCH`
- Verbos que deben ser seguros (es decir, que no deben modificar los recursos): `GET`
- Verbos que deben ser idempotentes (es decir, que no deben volver a afectar los recursos cuando se ejecutan varias veces): `GET` (nullipotente), `PUT` , `DELETE`

```
        body  safe  idempotent
GET      ✗  ✔  ✔
POST  ✔  ✗  ✗
PUT   ✔  ✗  ✔
DELETE ✗  ✗  ✔
PATCH  ✔  ✗  ✗
```

**En consecuencia** , los verbos HTTP se pueden comparar con las funciones CRUD :

- **C** reate : `POST`
- **R** EAD: `GET`
- **U** PDATE : `PUT` , `PATCH`
- **D** ELETE : `DELETE`

Tenga en cuenta que **una solicitud `PUT` solicita a los clientes que envíen todo el recurso** con los valores actualizados. Para actualizar parcialmente un recurso, se podría usar un verbo `PATCH` (consulte *¿Cómo actualizar parcialmente un recurso?* ).

# Estados usuales de respuesta HTTP

## Éxito

- 201 (CREADO) : el recurso ha sido creado
- 202 (ACEPTADO) : solicitud aceptada, pero proceso aún en curso
- 204 (SIN CONTENIDO) : solicitud cumplida y sin contenido adicional
- De lo contrario: 200 (OK)

## Redireccion

- 304 (NO MODIFICADO) : el cliente puede usar la versión en caché que tiene del recurso solicitado

## Errores del cliente

- 401 (NO AUTORIZADO) : una solicitud anónima accede a una API protegida
- 403 (FORBIDDEN) : una solicitud autenticada no tiene suficientes derechos para acceder a una API protegida

- 404 (NO ENCONTRADO) : recurso no encontrado
- 409 (CONFLICTO) : estado de recurso en conflicto (por ejemplo, un usuario que intenta crear una cuenta con un correo electrónico ya registrado)
- 410 (GONE) : igual que 404, pero el recurso existía
- 412 (PRECONDITION FAILED) : la solicitud intenta modificar un recurso que se encuentra en un estado inesperado
- 422 (ENTIDAD IMPRECESABLE) : la carga útil de solicitud es sintácticamente válida, pero es semánticamente errónea (por ejemplo, un campo obligatorio que no se ha valorado)
- 423 (BLOQUEADO) : el recurso está bloqueado
- 424 (DEPENDENCIA FALLIDA) : la acción solicitada dependía de otra acción que falló
- 429 (DEMASIADAS SOLICITUDES) : el usuario envió demasiadas solicitudes en un tiempo determinado
- De lo contrario: 400 (SOLICITUD MALA)

### Errores del servidor

- 501 (NO IMPLEMENTADO) : el servidor no admite la funcionalidad requerida para completar la solicitud
- 503 (SERVICIO NO DISPONIBLE) : el servidor actualmente no puede manejar la solicitud debido a una sobrecarga temporal o mantenimiento programado
- 507 (ALMACENAMIENTO INSUFICIENTE) : el servidor no puede almacenar la representación necesaria para completar con éxito la solicitud
- De lo contrario: 500 (ERROR DE SERVIDOR INTERNO)

### Notas

Nada le impide agregar un cuerpo a respuestas erróneas, para que el rechazo sea más claro para los clientes. Por ejemplo, el 422 (ENTIDAD IMPROCESABLE) es un tanto vago: el cuerpo de la respuesta debe proporcionar la razón por la cual la entidad no pudo ser procesada.

---

# Odio

Cada recurso debe proporcionar hipermedia a los recursos a los que está vinculado. Un enlace está compuesto *al menos* por:

- Una `rel` (para **rel** ación, nombre aka): describe la relación entre el recurso principal y el (los) enlace (s)
- A `href` : la URL dirigida a los recursos vinculados

También se pueden usar atributos adicionales para ayudar con la eliminación, la negociación de contenido, etc.

Cormac Mulhall explica que *el cliente debe decidir qué verbo HTTP usar en función de lo que está tratando de hacer* . En caso de duda, la documentación de la API debería ayudarlo a comprender las interacciones disponibles con todos los hipermedia.

---

# Tipos de medios

Los tipos de medios ayudan a tener mensajes auto-descriptivos. Juegan la parte del contrato entre clientes y servidores, de modo que puedan intercambiar recursos e hipermedias.

Aunque `application/json` y `application/xml` son tipos de medios muy populares, no contienen mucha semántica. Solo describen la sintaxis general utilizada en el documento. Se deben usar tipos de medios más especializados que admitan los requisitos de HATEOAS (o se extiendan a través de los tipos de medios de los proveedores ), como:

- Átomo
- RSS 2.0
- HAL
- colección + json
- JSON-LD
- Sirena

Un cliente le dice a un servidor qué tipo de medios entiende al agregar el encabezado `Accept` a su solicitud, por ejemplo:

```
Accept: application/hal+json
```

Si el servidor no puede producir el recurso solicitado en dicha representación, devuelve un 406 (NO ACEPTABLE) . De lo contrario, agrega el tipo de medio en el encabezado `Content-Type` de la respuesta que contiene el recurso representado, por ejemplo:

```
Content-Type: application/hal+json
```

---

# Sin estado> con estado

## ¿Por qué?

Un servidor con estado implica que las sesiones de los clientes se almacenan en un almacenamiento local de instancia de servidor (casi siempre en sesiones de servidor web). Esto comienza a ser un problema cuando se trata de escalar horizontalmente : si oculta varias instancias de servidor detrás de un equilibrador de carga, si un cliente se envía primero a la *instancia # 1* al iniciar sesión, pero luego a la *instancia # 2* cuando busca un recurso protegido, por ejemplo , luego, la segunda instancia manejará la solicitud como anónima, ya que **la sesión del cliente se ha almacenado localmente en la *instancia # 1*** .

Se han encontrado soluciones para abordar este problema (por ejemplo, mediante la configuración de la replicación de la sesión y / o la sesión persistente ), pero la arquitectura REST propone otro enfoque: simplemente no haga que su servidor esté lleno de estados, *deje el estado*

*sin estado* . Según Fielding :

> Cada solicitud del cliente al servidor debe contener toda la información necesaria para comprender la solicitud y no puede aprovechar ningún contexto almacenado en el servidor. **Por lo tanto, el estado de sesión se mantiene enteramente en el cliente.**

En otras palabras, una solicitud debe manejarse exactamente de la misma manera, independientemente de si se envía a la *instancia # 1* o la *instancia # 2* . Es por esto que las aplicaciones sin estado se consideran *más fáciles de escalar* .

## ¿Cómo?

Un enfoque común es una autenticación basada en token , especialmente con los tokens Web Jens de JSON . Tenga en cuenta que JWT todavía tiene algunos problemas, en particular con respecto a la invalidación y la prolongación automática de la caducidad (es decir, la función *Recordarme* ).

## Notas laterales

El uso de cookies o encabezados (o cualquier otra cosa) no tiene nada que ver con si el servidor tiene o no estados: estos son solo medios que se usan aquí para transportar tokens (identificador de sesión para servidores con estado, JWT, etc.), nada más.

Cuando los navegadores solo usan una API REST, las cookies ( HttpOnly y seguras ) pueden ser bastante convenientes ya que los navegadores las adjuntarán automáticamente a las solicitudes salientes. Vale la pena mencionar que **si opta por las cookies, tenga en cuenta CSRF** (una buena forma de evitarlo es hacer que los clientes generen y envíen el mismo valor secreto único tanto en una cookie como en un encabezado HTTP personalizado ).

---

# API cacheable con peticiones condicionales

## Con el encabezado `Last-Modified`

El servidor puede proporcionar un encabezado de fecha de `Last-Modified` a las respuestas que contienen recursos que se pueden almacenar en caché. Los clientes deben almacenar esta fecha junto con el recurso.

Ahora, cada vez que los clientes solicitan la API para leer el recurso, pueden agregar a sus solicitudes un encabezado `If-Modified-Since` contiene la última fecha de `Last-Modified` que recibieron y almacenaron. El servidor tiene entonces que comparar el encabezado de la solicitud y la *última* fecha de *modificación* real del recurso. Si son iguales, el servidor devuelve un 304 (NO MODIFICADO) con un cuerpo vacío: el cliente solicitante debe usar el recurso actualmente en caché que tiene.

Además, cuando los clientes solicitan la API para actualizar el recurso (es decir, con un verbo inseguro), pueden agregar un encabezado `If-Unmodified-Since` . Esto ayuda a lidiar con las condiciones de carrera: si el encabezado y la fecha de la *última modificación* actual son diferentes, el servidor devuelve un 412 (PRECONDITION FAILED) . El cliente debe leer el nuevo estado del recurso antes de volver a intentarlo para modificar el recurso.

## Con la cabecera `ETag`

Un ETag (etiqueta de entidad) es un identificador para un estado específico de un recurso. Puede ser un hash MD5 del recurso para una validación fuerte o un identificador específico del dominio para una validación débil .

Básicamente, el proceso es el mismo que con el encabezado `Last-Modified` : el servidor proporciona un encabezado `ETag` para las respuestas que contienen recursos que se pueden almacenar en caché, y los clientes deben almacenar este identificador junto con el recurso.

Luego, los clientes proporcionan un encabezado `If-None-Match` cuando quieren leer el recurso, que contiene la última etiqueta de ETag que recibieron y almacenaron. El servidor ahora puede devolver un 304 (NO MODIFICADO) si el encabezado coincide con el ETag real del recurso.

Nuevamente, los clientes pueden proporcionar un encabezado `If-Match` cuando quieren modificar el recurso, y el servidor tiene que devolver un 412 (PRECONDITION FAILED) si el ETag proporcionado no coincide con el real.

## Notas adicionales

### ETag> fecha

Si los clientes proporcionan tanto fecha como ETag en sus solicitudes, la fecha debe ignorarse. Desde RFC 7232 ( aquí y aquí ):

> Un destinatario DEBE ignorar `If-Modified-Since` / `If-Unmodified-Since` si la solicitud contiene un campo de encabezado `If-None-Match` / `If-Match` ; la condición en `If-None-Match` / `If-Match` se considera un reemplazo más preciso de la condición en `If-Modified-Since` / `If-Unmodified-Since` , y las dos solo se combinan por el bien de la interoperación con intermediarios más antiguos eso podría no implementar `If-None-Match` / `If-Match` .

### Poco profundas

Además, aunque es bastante obvio que las *últimas fechas modificadas* se conservan junto con los recursos del lado del servidor, hay varios enfoques disponibles con ETag.

Un enfoque habitual es implementar ETags poco profundos: el servidor procesa la solicitud como si no se dieran encabezados condicionales, pero solo al final, genera el ETag de la respuesta que está a punto de devolver (p. Ej., Mediante hashing) y compara con el proporcionado. Esto es relativamente fácil de implementar ya que solo se necesita un interceptor HTTP (y ya existen

muchas implementaciones, según el servidor). Dicho esto, vale la pena mencionar que este enfoque **ahorrará ancho de banda pero no el rendimiento del servidor** :

> Una **implementación más profunda** del mecanismo ETag podría proporcionar beneficios mucho mayores, como atender algunas solicitudes de la caché y no tener que realizar el cálculo en absoluto, pero la implementación definitivamente no sería tan simple ni tan fácil de conectar como el enfoque superficial. descrito aquí.

---

# Errores comunes

## ¿Por qué no debería poner verbos en una URL?

HTTP no es RPC : *lo que hace a HTTP significativamente diferente de RPC es que **las solicitudes se dirigen a los recursos*** . Después de todo, la URL significa Uniform *Resource* Locator, y una URL es un URI : un Uniform *Resource* Idenfitier. **La URL se dirige *al recurso con el que desea tratar* , el método HTTP indica *qué desea hacer con él* .** Los métodos HTTP también se conocen como *verbos* : los verbos en las URL no tienen sentido. Tenga en cuenta que las relaciones HATEOAS no deberían contener verbos, ya que los enlaces también están dirigidos a los recursos.

## ¿Cómo actualizar parcialmente un recurso?

Como las solicitudes `PUT` solicitan a los clientes que envíen **el recurso completo** con los valores actualizados, `PUT /users/123` no se puede usar para actualizar simplemente el correo electrónico de un usuario, por ejemplo. Según lo explicado por William Durand en Por favor. No parches como un idiota. , varias soluciones compatibles con REST están disponibles:

- Exponga las propiedades del recurso y use el método `PUT` para enviar un valor actualizado, ya que la especificación `PUT` indica que *las actualizaciones parciales de contenido son posibles al dirigir un recurso identificado por separado con un estado que se superpone a una parte del recurso más grande* :

```
PUT https://example.com/api/v1.2/users/123/email
body:
  new.email@example.com
```

- Utilice una solicitud de `PATCH` que contiene un conjunto de instrucciones que describen cómo se debe modificar el recurso (por ejemplo, siguiendo el parche JSON ):

```
PATCH https://example.com/api/v1.2/users/123
body:
  [
    { "op": "replace", "path": "/email", "value": "new.email@example.com" }
  ]
```

- Utilice una solicitud de `PATCH` contenga una representación parcial del recurso, tal como se

propone en el comentario de Matt Chapman :

```
PATCH https://example.com/api/v1.2/users/123
body:
  {
    "email": "new.email@example.com"
  }
```

## ¿Qué pasa con las acciones que no encajan en el mundo de las operaciones de CRUD?

Citando a Vinay Sahni en Mejores prácticas para diseñar una API RESTful pragmática :

> Aquí es donde las cosas pueden ponerse borrosas. Hay una serie de enfoques:
>
> 1. Reestructurar la acción para que aparezca como un campo de un recurso. Esto funciona si la acción no toma parámetros. Por ejemplo, una acción de *activación* podría asignarse a un campo `activated` booleano y actualizarse mediante un PATCH al recurso.
> 2. Trátelo como un sub-recurso con principios RESTful. Por ejemplo, la API de GitHub te permite iniciar una esencia con `PUT /gists/:id/star` y unstar con `DELETE /gists/:id/star` .
> 3. A veces, realmente no tiene forma de asignar la acción a una estructura REST completa. Por ejemplo, una búsqueda de múltiples recursos realmente no tiene sentido que se aplique a un punto final de un recurso específico. En este caso, `/search` tendría más sentido aunque no sea un recurso. Esto está bien, solo haga lo correcto desde la perspectiva del consumidor de API y asegúrese de que esté documentado claramente para evitar confusiones.

---

# Practicas comunes

- API está documentada. Las herramientas están disponibles para ayudarlo a construir su documentación, por ejemplo, Swagger o Spring REST Docs .
- La API está versionada , ya sea a través de encabezados o a través de la URL:

```
https://example.com/api/v1.2/blogs/123/articles
                        ^^^^
```

- Los recursos tienen nombres en plural :

```
https://example.com/api/v1.2/blogs/123/articles
                             ^^^^^     ^^^^^^^^
```

- Las URL utilizan kebab-case (las palabras están en minúsculas y separadas por guiones):

```
https://example.com/api/v1.2/quotation-requests
                             ^^^^^^^^^^^^^^^^^^
```

- HATEOAS proporciona un enlace "propio" a los recursos, dirigiéndose a sí mismos:

```
{
  ...,
  _links: {
    ...,
    self: { href: "https://example.com/api/v1.2/blogs/123/articles/789" }
    ^^^^
  }
}
```

- Las relaciones de HATEOAS utilizan lowerCamelCase (las palabras se escriben en minúsculas, luego se escriben con mayúscula, excepto la primera, y se omiten los espacios), para permitir que los clientes de JavaScript utilicen la notación de puntos respetando las convenciones de nomenclatura de JavaScript al acceder a los enlaces:

```
{
  ...,
  _links: {
    ...,
    firstPage: { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=1&pageSize=25" }
    ^^^^^^^^^
  }
}
```

### Gestión de blogs a través de una API HTTP REST.

Los siguientes ejemplos utilizan HAL para expresar HATEOAS y utilizan:

- CURIE (URI compacto): se utiliza para proporcionar enlaces a la documentación de la API
- Plantillas de URI : URI que incluye parámetros que deben sustituirse antes de que se resuelva el URI

# Obtener blog 123

## Solicitud

```
GET https://example.com/api/v1.2/blogs/123
headers:
  Accept: application/hal+json
```

## Respuesta

```
status: 200 (OK)
headers:
  Content-Type: application/hal+json
body:
  {
    "id": 123,
    "title": "The blog title",
    "description": "The blog description",
    "_links": {
      "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated": true
}],
      "self": { "href": "https://example.com/api/v1.2/blogs/123" },
      "doc:articles": { "href":
"https://example.com/api/v1.2/blogs/123/articles{?pageIndex,pageSize}", "templated": true }
    }
  }
```

# Crea un nuevo artículo en el blog 123.

## Solicitud

```
POST https://example.com/api/v1.2/blogs/123/articles
headers:
  Content-Type: application/json
  Accept: application/hal+json
  X-Access-Token: XYZ
body:
  {
    "title": "The title 2",
    "content": "The content 2"
  }
```

## Respuesta

```
status: 201 (CREATED)
headers:
  Content-Type: application/hal+json
body:
  {
    "id": 789,
    "title": "The title 2",
    "content": "The content 2",
    "_links": {
      "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated": true
}],
      "self": { "href": "https://example.com/api/v1.2/blogs/123/articles/789" },
      "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" },
      "doc:comments": { "href":
"https://example.com/api/v1.2/blogs/123/articles/789/comments{?pageIndex,pageSize}",
"templated": true }
    }
  }
```

# Consigue el artículo 789 del blog 123.

## Solicitud

```
GET https://example.com/api/v1.2/blogs/123/articles/789
headers:
  Accept: application/hal+json
```

## Respuesta

```
status: 200 (OK)
headers:
  Content-Type: application/hal+json
body:
  {
    "id": 789,
    "title": "The title 2",
    "content": "The content 2",
    "_links": {
      "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated": true
}],
      "self": { "href": "https://example.com/api/v1.2/blogs/123/articles/789" },
      "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" },
      "doc:comments": { "href":
"https://example.com/api/v1.2/blogs/123/articles/789/comments{?pageIndex,pageSize}",
"templated": true }
    }
  }
```

# Consigue la 4ª página de 25 artículos del blog 123.

## Solicitud

```
GET https://example.com/api/v1.2/blogs/123/articles?pageIndex=4&pageSize=25
headers:
  Accept: application/hal+json
```

## Respuesta

```
status: 200 (OK)
headers:
  Content-Type: application/hal+json
body:
  {
```

```
    "pageIndex": 4,
    "pageSize": 25,
    "totalPages": 26,
    "totalArticles": 648,
    "_link": {
      "firstPage": { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=1&pageSize=25" },
      "previousPage": { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=3&pageSize=25" },
      "self": { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=4&pageSize=25" },
      "nextPage": { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=5&pageSize=25" },
      "lastPage": { "href":
"https://example.com/api/v1.2/blogs/123/articles?pageIndex=26&pageSize=25" }
    },
    "_embedded": [
      {
        ...
      }, {
        "id": 456,
        "title": "The title 1",
        "content": "The content 1",
        "_links": {
          "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated":
true }],
          "self": { "href": "https://example.com/api/v1.2/blogs/123/articles/456" },
          "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" },
          "doc:comments": { "href":
"https://example.com/api/v1.2/blogs/123/articles/456/comments{?pageIndex,pageSize}",
"templated": true }
        }
      }, {
        "id": 789,
        "title": "The title 2",
        "content": "The content 2",
        "_links": {
          "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated":
true }],
          "self": { "href": "https://example.com/api/v1.2/blogs/123/articles/789" },
          "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" },
          "doc:comments": { "href":
"https://example.com/api/v1.2/blogs/123/articles/789/comments{?pageIndex,pageSize}",
"templated": true }
        }
      }, {
        ...
      }
    ]
  }
```

# Actualizar artículo 789 del blog 123.

## Solicitud

```
PUT https://example.com/api/v1.2/blogs/123/articles/789
headers:
  Content-Type: application/json
  Accept: application/hal+json
  X-Access-Token: XYZ
body:
  {
    "id": 789,
    "title": "The title 2 updated",
    "content": "The content 2 updated"
  }
```

## Respuesta

```
status: 200 (OK)
headers:
  Content-Type: application/hal+json
body:
  {
    "id": 789,
    "title": "The title 2 updated",
    "content": "The content 2 updated",
    "_links": {
      "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated": true
}],
      "self": { "href": "https://example.com/api/v1.2/blogs/123/articles/789" },
      "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" },
      "doc:comments": { "href":
"https://example.com/api/v1.2/blogs/123/articles/789/comments{?pageIndex,pageSize}",
"templated": true }
    }
  }
```

## Notas

- El identificador que se usa para identificar el recurso a actualizar es **el que está en la URL** : el que está en el cuerpo (si lo hay) debe ignorarse silenciosamente.
- Como una solicitud `PUT` actualiza todo el recurso, si no se hubiera enviado ningún `content` , debería haberse eliminado del recurso persistido.

# Eliminar artículo 789 del blog 123.

## Solicitud

```
DELETE https://example.com/api/v1.2/blogs/123/articles/789
headers:
  Accept: application/hal+json
  X-Access-Token: XYZ
```

## Respuesta

```
status: 204 (NO CONTENT)
headers:
  Content-Type: application/hal+json
body:
  {
    "_links": {
      "curies": [{ "name": "doc", "href": "https://example.com/docs/{rel}", "templated": true
}],
      "doc:blog": { "href": "https://example.com/api/v1.2/blogs/123", "title": "The blog
title" }
    }
  }
```

### Violación REST

```
<stock>
    <add>
        <item>
            <name>Milk</name>
            <quantity>2</quantity>
        </item>
    </add>
</stock>
```

Poner este cuerpo a un recurso como `/stocks/123` viola la idea detrás de REST. Si bien se `put` este cuerpo y contiene toda la información necesaria, también viene junto con un método llamado a `add` algún lugar cuando se procesa el cuerpo. Después de REST, se publicaría el `item` en `/stocks/123/items/` .

Lea Empezando con el descanso en línea: https://riptutorial.com/es/rest/topic/1664/empezando-con-el-descanso

# Creditos

| S. No | Capítulos | Contributors |
|---|---|---|
| 1 | Empezando con el descanso | Community, Jason, Najeeb Arif, Roman Vottner, Shog9, slartidan, sp00m, ssanrao, sschrass, zretscen |