Gobierno de APIs. (API Owner)

Filosofía REST

Índice del capítulo

- ¿Qué es REST?
- Protocolo HTTP.
- Hipermedia.
- Servicios CRUD.
- HATEOAS.
- Buenas prácticas para el diseño de una API RESTful.

¿Qué es REST?

- Introducción a los servicios REST.
- Características de REST.
- Ventajas que ofrece REST para el desarrollo.
- API RESTful.

Introducción a los servicios REST

- **REST** cambió por completo la ingeniería de software **a partir del 2000**.
- Este nuevo enfoque de desarrollo de proyectos y servicios web fue definido por **Roy Fielding**, el padre de la **especificación HTTP** y uno los referentes internacionales en todo lo relacionado con la Arquitectura de Redes, en su disertación 'Architectural Styles and the Design of Network-based Software Architectures'.
- En el campo de las APIs, REST (**Representational State Transfer** - Transferencia de Estado Representacional) es, a día de hoy, el **alfa y omega del desarrollo** de **servicios** de aplicaciones.
- En la actualidad, no existe proyecto o aplicación que no disponga de una API REST para la creación de servicios profesionales a partir de ese software.
- Twitter, YouTube, los sistemas de identificación con Facebook, etc, hay cientos de empresas que generan negocio gracias a **REST y las APIs REST**. Sin ellas, todo el crecimiento en horizontal sería prácticamente imposible. Esto es así porque REST es el estándar más lógico, eficiente y habitual en la creación de APIs para servicios de Internet.

Introducción a los servicios REST

- Buscando una definición sencilla, REST es **cualquier interfaz** entre sistemas que use **HTTP** para **obtener datos o generar operaciones** sobre esos datos en todos los formatos posibles, como **XML y JSON**.

- Es una **alternativa en auge** a otros protocolos estándar de intercambio de datos como **SOAP** (Simple Object Access Protocol), que disponen de una **gran capacidad** pero también **mucha complejidad**.

- A veces es preferible una **solución más sencilla** de manipulación de datos como REST.

Características de REST

- Introducción.
- Cliente-Servidor.
- Sin estado.
- Información cacheable.
- Interfaz uniforme.
- Acceso a recursos por nombre.
- Recursos relacionados.
- Respuesta en un formato conocido.
- Uso de hipermedia.

Características de REST

- REST **no es un estándar**, simplemente define unos principios de arquitectura a seguir para implementar aplicaciones o servicios en red. Sin embargo, **REST se basa en estándares para su implementación**: HTTP, XML, etc.
- Los servicios REST tienen las siguientes características:
  - Cliente-Servidor.
  - Sin estado.
  - Información cacheable.
  - Interfaz uniforme.
  - Acceso a recursos por nombre.
  - Recursos relacionados.
  - Respuesta en un formato conocido.

Cliente-Servidor

- Los **servicios REST** deben estar basados en una **arquitectura Cliente-Servidor**.
- Un servidor que contiene los **recursos** y **estados** de los mismos, y unos clientes que acceden a ellos.
- Cada **petición HTTP** contiene **toda la información necesaria para ejecutarla**, lo que permite que ni cliente ni servidor necesiten recordar **ningún estado previo** para satisfacerla.
- Aunque esto es así, algunas aplicaciones HTTP incorporan memoria caché. Se configura lo que se conoce como protocolo cliente-caché-servidor sin estado: existe la posibilidad de definir algunas respuestas a peticiones HTTP concretas como cacheables, con el objetivo de que el cliente pueda ejecutar en un futuro la misma respuesta para peticiones idénticas. De todas formas, **que exista la posibilidad no significa que sea lo más recomendable**.
- Las **operaciones más importantes** relacionadas con los datos en cualquier sistema REST y la especificación HTTP son cuatro: **POST** (crear), **GET** (leer y consultar), **PUT** (editar) y **DELETE** (eliminar).

Sin estado

- Los **Servicios REST** pueden ser **escalados** hasta alcanzar **grandes rendimientos** para abarcar la demanda de todos los posibles clientes.
- Esto implica que sea necesario crear **granjas de servidores con balanceo de cargas y failover** o diferentes niveles de **servidores** para minimizar el tiempo de respuesta a los clientes.
- Al utilizar servidores intermedios, es necesario que los clientes REST envíen la **información completa e independiente** en cada **solicitud** de estado de un recurso.
- De esta manera, los servidores intermedios pueden reenviar, enrutar, balancear sin necesidad de que los servidores intercambien información de sesiones de clientes.

Sin estado

- Una **solicitud completa e independiente** no requiere que el servidor, mientras procesa la solicitud, tenga que almacenar **ningún tipo de contexto o sesión**.
- Un **cliente REST** debe incluir dentro de la cabecera y cuerpo HTTP todos los **parámetros**, **contexto** y **datos necesarios** para que el servidor genere la respuesta.
- Esto **aumenta el rendimiento del servicio REST** y **simplifica** el diseño e implementación del servidor ya que la ausencia de sesiones de clientes elimina la necesidad de sincronizar datos de sesión con aplicaciones externas.

Información cacheable

- Para mejorar la **eficiencia** en el tráfico de red, las **respuestas del servidor** deben tener la posibilidad de ser marcadas como **cacheables**.
- Esta información es utilizada por los **clientes REST** para decidir **si hacer una copia local del recurso** con la fecha y hora del último cambio de estado del recurso.
- En tal caso, el **cliente** realiza **solicitudes del estado del recurso** de forma que, si el estado no ha cambiado, el **servidor solo le responde** con un **mensaje muy pequeño** indicando que no ha cambiado.
- Esto permite **optimizar el tráfico de red**.

Interfaz uniforme

- Una de las **características principales** de los servicios Web REST es el **uso explícito de métodos HTTP** (HyperText Transfer Protocol). Estos métodos son indicados en la cabecera HTTP por parte del cliente y son los **siguientes**:
  - GET: recoge información de un recurso.
  - PUT: modifica o actualiza el estado de un recurso.
  - POST: crea un nuevo recurso en el servidor.
  - DELETE: elimina un recurso del servidor.
- Habitualmente una URL en una **solicitud HTTP GET identifica un recurso**. Por ejemplo, en la siguiente solicitud se estaría solicitando el estado de un recurso llamado “SensorHumedad_001" dentro del catálogo de sensores de MiEmpresa:
  - http://www.MiEmpresa.com/Sensores/SensorHumedad_001

Interfaz uniforme

- **Otra manera** de solicitar información a un servidor es **construyendo una URL** que incluya **parámetros** que definan los **criterios de búsqueda** en el servidor para que pueda encontrar los recursos solicitados:
  - http://www.MiEmpresa.com/Sensores?Tipo=humedad&id=001
- Un **recurso** es una **URL lógica**, no física. De este modo **no es necesario** que en el servidor exista una **página HTML para cada uno de los sensores**.
- Con un **método POST** se generaría el nuevo recurso en el servidor, el cual sería accesible como URL lógica.

Interfaz uniforme

- El **modo** en el que el servidor maneja las solicitudes de los clientes debe estar oculto para los éstos.
- Un **cliente** no debe saber el **lenguaje de programación** utilizado en el servidor ni cómo éste está generando la información; simplemente debe saber el **modo de acceder a la información**.
- Esto permite **migrar** de un servidor a otro, de un lenguaje a otro **sin necesidad** de hacer cambios en los **clientes existentes**.
- En los principio definidos por REST, se diferencia el **método GET** de los demás ya que éste no debe **tener efectos de cambio en la información contenida en el servidor**.
- Los métodos **PUT, POST y DELETE** modifican un recurso, pero **GET solo debe recoger información**, nunca modificar el recurso.

Acceso a recursos por nombre

- Un **sistema REST** está compuesto por recursos a los que se accede mediante URL y éstas deben ser **intuitivas, predecibles** y **fáciles** de entender y componer.
- Una manera de conseguirlo es mediante una **estructura jerárquica**, similar a directorios. Puede existir un **nodo raíz único**, a partir del cual se crean los **subdirectorios** que expongan las **áreas principales de los servicios**, hasta formar un árbol con la información de los recursos.
- El acceso a los recursos se realiza como **composición de una URL** que identifique el recurso, y debe ser la **misma aunque la implementación en el servidor se modifique**.
- La URL compuesta **no debe contener llamadas a funciones** que ejecuten código en el servidor, por lo que no deberían ser direcciones de archivos que ejecuten acciones (páginas con extensión .jsp, .php, .asp).

Recursos relacionados

- Los **recursos accesibles** en el servidor suelen estar **relacionados unos con otros**.
- Por tanto, la **información de estado de un recurso** debería permitir **acceder** a **otros** recursos.
- Esto se consigue añadiendo en el estado de los **recursos links o URL** de otros recursos.
- Por ejemplo, se podría pedir un recurso cuyo estado identifique los sensores de un determinado tipo:
  - http://www.MiEmpresa.com/Sensores?Tipo=humedad
- El servidor podría devolver **el listado** de todos los **sensores de ese tipo**, y dentro de la información del estado se puede incluir un **link a cada uno de los sensores**, añadiendo una capacidad de Drill-down para acceder al máximo detalle.

Recursos relacionados

- La respuesta del servidor a la solicitud anterior podría ser similar a esta:
  - {Id}SensorHumedad_001 {Detalle}http://www.MiEmpresa.com/Sensores/SensorHumedad_001
  - {Id}SensorHumedad_002 {Detalle}http://www.MiEmpresa.com/Sensores/SensorHumedad_002
  - {Id}SensorHumedad_003 {Detalle}http://www.MiEmpresa.com/Sensores/SensorHumedad_003
  - {Id}SensorHumedad_004 {Detalle}http://www.MiEmpresa.com/Sensores/SensorHumedad_004

Recursos relacionados

- Una **buena práctica** en los servicios REST es **no mostrar toda la información** del estado de un recurso en una solicitud, sino mostrar la información de **manera gradual**.
- Esto permite **minimizar** el **uso de la red** si el cliente no necesita muchos detalles de un determinado recurso.
- Esto se puede conseguir **incluyendo links en el estado del recurso**, siendo estos **links** otros recursos con **mayor nivel de detalle**.

Respuesta en un formato conocido

- La **representación** de un recurso refleja el **estado actual del mismo** y sus atributos en el instante en el que el cliente ha realizado la solicitud.
- Este resultado puede representar simplemente el **valor de una variable** en un instante de tiempo, un **registro** de una Base de Datos o cualquier otro tipo de información.
- En cualquiera de los casos, la **información** debe ser entregada al cliente en un **formato comprensible** para **ambas partes** y contenida dentro del **cuerpo HTTP**.
- Este **contenido** debe ser **simple y compresible** por un humano y, por supuesto, **interpretable** por una aplicación.
- Esto permite **utilizar** el servicio REST por **diferentes clientes** independientemente del lenguaje con el que se hayan programado.

Respuesta en un formato conocido

- Los **formatos** más habituales son **JSON** (JavaScript Object Notation) y **XML** (Extensible Markup Language), aunque se aceptan otros, como **CSV** (Comma Separated Values).
- Cada formato tiene sus **ventajas y desventajas:**
  - Puesto que **JSON** fue diseñado para JavaScript, su interpretación es muy directa en este entorno.
  - **XML** es fácil de expandir y contraer ya que la información está anidada; además, es un formato muy conocido.
  - **CSV**, por su parte, es un formato compacto.

Respuesta en un formato conocido

- En los servicios REST, puesto que pueden **existir diferentes tipos de clientes**, **no** se aconseja implementar un servidor que devuelva el estado de un recurso **en un solo formato**.
- Esto se puede conseguir de diferentes maneras, como por ejemplo, crear en el servidor una URL diferente para cada formato, o escribiendo un parámetro en la **misma URL** que el servidor interprete.
- A continuación se muestran ambas opciones:
  - http://www.MiEmpresa.com/Sensores/xml/Sensor001
  - http://www.MiEmpresa.com/Sensores/Sensor001?Output=xml

Uso de hipermedia

- **Hipermedia** es un término acuñado por **Ted Nelson en 1965** y que es una extensión del concepto de hipertexto.
- Ese concepto llevado al desarrollo de páginas web es lo que permite que el usuario puede navegar por el conjunto de objetos a través de enlaces HTML.
- En el caso de una API REST, el **concepto de hipermedia** explica la capacidad de una interfaz de desarrollo de aplicaciones de proporcionar al cliente y al usuario los enlaces adecuados para ejecutar acciones concretas sobre los datos.

Ventajas que ofrece REST para el desarrollo

- **Separación entre el cliente y el servidor:**
  - El protocolo REST separa totalmente la **interfaz de usuario del servidor** y el **almacenamiento** de datos.
  - Eso tiene algunas ventajas cuando se hacen desarrollos. Por ejemplo, mejora la portabilidad de la interfaz a otro tipo de plataformas, aumenta la escalabilidad de los proyectos y permite que los distintos componentes de los desarrollos se puedan evolucionar de forma independiente.

Ventajas que ofrece REST para el desarrollo

- **Visibilidad, fiabilidad y escalabilidad**:
  - La separación entre **cliente y servidor** tiene una ventaja evidente y es que cualquier equipo de desarrollo puede **escalar el producto sin excesivos problemas**.
  - Se puede **migrar** a otros servidores o realizar todo **tipo de cambios en la base de datos**, siempre y cuando los **datos de cada una de las peticiones se envíen de forma correcta**.
  - Esta separación facilita tener en **servidores distintos el front y el back** y eso convierte a las aplicaciones en productos más flexibles a la hora de trabajar.

Ventajas que ofrece REST para el desarrollo

- La API REST siempre es **independiente** del **tipo de plataformas o lenguajes**:
  - La API REST siempre se adapta al tipo de sintaxis o plataformas con las que se estén trabajando, lo que ofrece una **gran libertad a la hora de cambiar** o probar nuevos entornos dentro del desarrollo.
  - Con una API REST se pueden tener servidores **PHP, Java, Python o Node.js**.
  - Lo único que es **indispensable** es que las respuestas a las peticiones se hagan siempre en el **lenguaje de intercambio de información** usado, normalmente **XML o JSON**.

API RESTful

- Un **servicio web** es un programa, diseñado para el **intercambio de información máquina a máquina**, sobre una red. Entonces esto hace que una computadora o programa pueda solicitar y recibir información de otra computadora o programa. A quien solicita la información se le llama **cliente y a quien envía la información se le llama servidor**.

- Una **API (Application Programming Interface)** no es más que un programa que permite que otros programas se comuniquen con un programa en específico. A diferencia de los web services, las **API no necesariamente deben comunicarse entre una red**, pueden usarse entre dos aplicaciones en una misma computadora.

- Una API RESTful es un programa basado en REST.

API RESTful

- Ejemplo: API REST de Activiti. Los métodos a utilizar se muestran a continuación:

Method	Operations
GET	Get a single resource or get a collection of resources.
POST	Create a new resource. Also used for executing resource-queries which have a too complex request-structure to fit in the query-URL of a GET-request.
PUT	Update properties of an existing resource. Also used for invoking actions on an existing resource.
DELETE	Delete an existing resource.

API RESTful

- Ejemplo: API REST de Activiti. Código de respuesta Http

Response	Description
`200 - Ok`	The operation was successful and a response has been returned (`GET` and `PUT` requests).
`201 - Created`	The operation was successful and the entity has been created and is returned in the response-body (`POST` request).
`204 - No content`	The operation was successful and entity has been deleted and therefore there is no response-body returned (`DELETE` request).
`401 - Unauthorized`	The operation failed. The operation requires an Authentication header to be set. If this was present in the request, the supplied credentials are not valid or the user is not authorized to perform this operation.
`403 - Forbidden`	The operation is forbidden and should not be re-attempted. This does not imply an issue with authentication not authorization, it's an operation that is not allowed. Example: deleting a task that is part of a running process is not allowed and will never be allowed, regardless of the user or process/task state.
`404 - Not found`	The operation failed.The requested resource was not found.
`405 - Method not allowed`	The operation failed. The used method is not allowed for this resource. Eg. trying to update (PUT) a deployment-resource will result in a `405` status.
`409 - Conflict`	The operation failed. The operation causes an update of a resource that has been updated by another operation, which makes the update no longer valid. Can also indicate a resource that is being created in a collection where a resource with that identifier already exists.
`415 - Unsupported Media Type`	The operation failed. The request body contains an unsupported media type. Also occurs when the request-body JSON contains an unknown attribute or value that doesn't have the right format/type to be accepted.
`500 - Internal server error`	The operation failed. An unexpected exception occurred while executing the operation. The response-body contains details about the error.

API RESTful

- Ejemplo: API REST de Activiti. Parámetros de la petición.
  - Los parámetros de la petición forman parte de la url, son lo que se denominan fragmentos url:

```
http://host/actviti-rest/service/repository/deployments/{deploymentId})
```

  - Los parámetros añadidos en la url como consulta (http://host/activiti-rest/service/deployments?name=Deployment) pueden tener los siguientes tipos:

Type	Format
String	Plain text parameters. Can contain any valid characters that are allowed in URL's. In case of a `XXXLike` parameter, the string should contain the wildcard characted `%` (properly url-encoded). This allows to specify the intent of the like-search. Eg. '`Tas%`' matches all values, starting with 'Tas'.
Integer	Parameter representing an integer value. Can only contain numeric non-decmimal values, between -2.147.483.648 and 2.147.483.647.
Long	Parameter representing a long value. Can only contain numeric non-decmimal values, between -9.223.372.036.854.775.808 and 9.223.372.036.854.775.807.
Boolean	Parameter representing a boolean value. Can be eiter `true` or `false`. All other values other than these two, will cause a '`405 - Bad request`' response.
Date	Parameter representing a date value. Use the ISO-8601 date-format (see ISO-8601 on wikipedia) using both time and date-components (eg. `2013-04-03T23:45Z`).

Protocolo HTTP

- Introducción.
- El funcionamiento de HTTP.
- Peticiones HTTP.
- Métodos de HTTP.
- Respuestas HTTP.
- Cabeceras HTTP.
- Códigos de estado.

Introducción

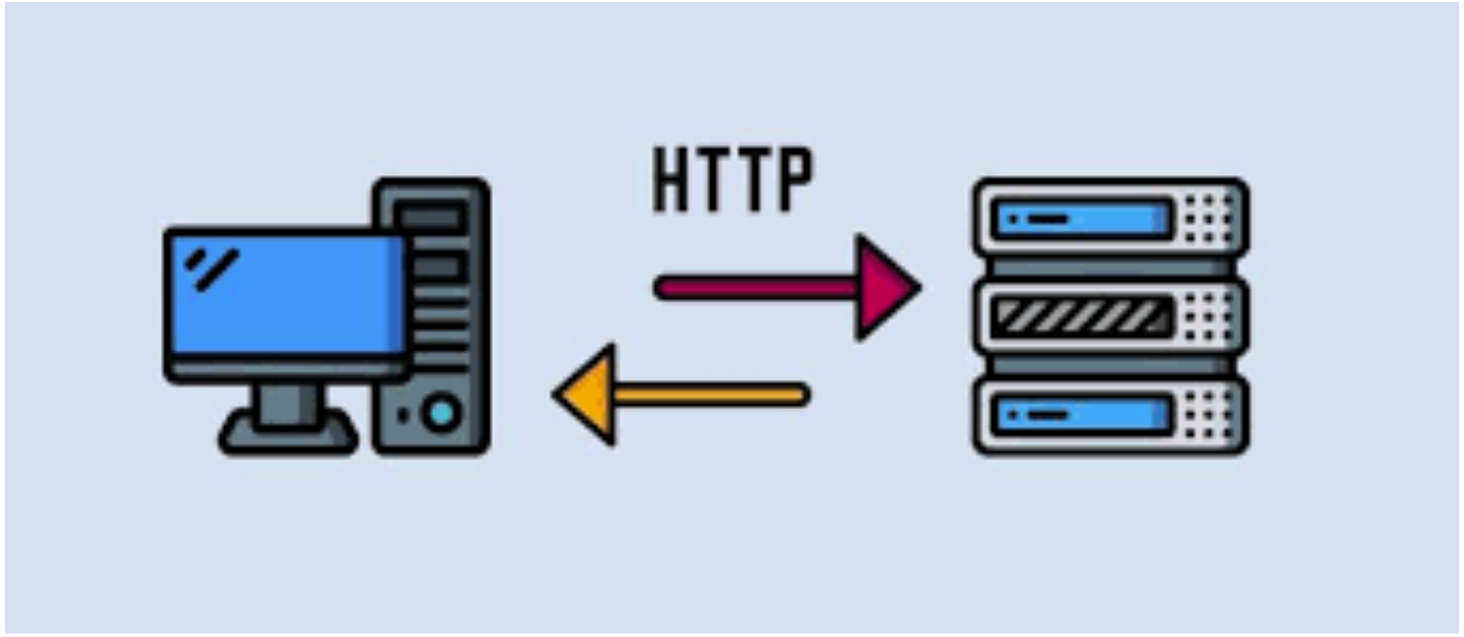
- HTTP o **HyperText Transfer Protocol** es un **protocolo de transferencia** sobre el que se basa la red informática mundial (WWW).

- Funciona como **base** para los **intercambios de datos** realizados en la web y mantiene una estructura basadas en los **clientes y servidores** y orientada a **transacciones**.

Introducción

- La **arquitectura del protocolo HTTP** implica que **programas clientes** como Firefox, Chrome, Opera y Robots, establezcan conexión y realicen **peticiones** de datos a programas servidores como Apache, Nginx, entre otros.

- Estas peticiones son **gestionadas y contestadas** por los servidores a través de intermediarios denominados proxies.



El funcionamiento de HTTP

- Se trata de un **protocolo que trabaja a nivel de aplicación** y que está soportado por los servicios de conexión **TCP/IP**.
- Su funcionamiento es idéntico al del resto de servicios de su entorno:
  - El **servidor** está continuamente **escuchando el puerto de comunicaciones TCP** a la espera de que llegue una solicitud de conexión por parte del cliente HTTP (el navegador, por ejemplo).
  - En cuanto **llega dicha solicitud** el mismo protocolo TCP se encarga de mantener la comunicación y garantizar que el intercambio de datos se realiza sin errores. Es un proceso de **solicitud/ respuesta**.
  - El **cliente** establece una **conexión** con un servidor por medio de una **solicitud**.
  - El **servidor** responde con un mensaje que contiene el **estado de la operación** (si se encuentra ese recurso, si no aparece…) y el **resultado** que el cliente procesa, interpreta y muestra al usuario.

El funcionamiento de HTTP

- Para que este **protocolo funcione correctamente** necesita de un mecanismo que permita **identificar los recursos**, con el fin de que el cliente los pueda llamar de la forma correcta, y el servidor sepa en todo momento y de manera unívoca qué es lo que se busca.

- Ese mecanismo es la **URL** (Uniform Resource Locator). Es una **forma estándar** de hacer referencia a cualquier **recurso** (servicio) de Internet.

- Si el **servidor HTTP** no puede mostrar el recurso solicitado, envía un código de respuesta formado por tres dígitos. El primero indica el **estado**, mientras que los otros dos explican la naturaleza exacta del error. En otras palabras, es el origen del famoso **Error 404: URL not found**.

- Para entender correctamente el funcionamiento de HTTP vamos a analizar la estructura de la **petición** y la **respuesta** HTTP.

Peticiones HTTP

- Una petición HTTP está formado por los siguientes elementos:
  - **Método**: GET, POST, PUT, etc. Indica qué tipo de request es.
  - **Path**: la URL que se solicita, donde se encuentra el resource/recurso.
  - **Protocolo**: contiene HTTP y su versión, actualmente 1.1.
  - **Headers:** Son esquemas de **key: value** que contienen información sobre **el HTTP request** y el **navegador**. Aquí también se encuentran los datos de las **cookies**. La mayoría de los headers son opcionales.
  - **Body:** Si se envía información al servidor a través de **POST o PUT** va en el body.

Peticiones HTTP

- Ejemplo:

```
GET php.net HTTP/1.1 **Accept**: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
**Accept-Encoding**: gzip, deflate, sdch
**Accept-Language**: es-ES,es;q=0.8,en;q=0.6
**Cache-Control**: max-age=0
**Connection**: keep-alive
**Cookie**: COUNTRY=NA%2C122.16.430.651; LAST_LANG=es; LAST_NEWS=3847110839
**Host**: php.net
**If-Modified-Since**: Mon, 09 Nov 2015 11:50:11 GMT
**Upgrade-Insecure-Requests**: 1
**User-Agent**: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
46.0.2490.80 Safari/537.36
```

Métodos de HTTP

- Este protocolo implementa varios **métodos de peticiones** que tienen diferentes funciones.
- **GET**:
  - Este método solicita la **representación de un recurso determinado**. Las peticiones que utilicen el método GET **solo** deben ser usadas para **la recuperación de información**, por lo que no pueden incluir datos.
- **POST**:
  - Tiene la función de **enviar datos** para que sean procesados en un recurso especificado, lo que usualmente trae como consecuencia efectos secundarios en el servidor, como por ejemplos, **cambios en su estado**.
- **PUT**:
  - Se encarga de **reemplazar las representaciones** que tenga el recurso de destino con la **carga útil** de la petición realizada.

Métodos de HTTP

- **CONNECT**:
  - Tiene la función de establecer un **túnel hacia el servidor** que haya sido identificado con el recurso.
- **DELETE**:
  - Esta a cargo de **eliminar un recurso** especificado por el usuario.
- **OPTIONS**:
  - Este modo es usado con el fin de **describir las opciones de comunicación** que tiene el recurso de destino.

Respuestas HTTP

- Una vez que el navegador envía una petición HTTP, el servidor responde con una respuesta HTTP formada por los siguientes elementos:
  - **Protocolo**: Contiene HTTP y su versión, actualmente 1.1.
  - **Status code:** El código de respuesta, por ejemplo: 200 OK, que significa que el GET request ha sido satisfactorio y el servidor devolverá los contenidos del documento solicitado. Otro ejemplo es 404 Not Found, el servidor no ha encontrado el resource solicitado.
  - **Headers:** Contienen información sobre el software del servidor, cuando se modificó por última vez el resource solicitado, el mime type, etc. De nuevo la mayoría son opcionales.
  - **Body**: Si el servidor devuelve información que no sean headers va en el body.

Cabeceras HTTP

- Introducción.
- Cabeceras Accept.
- Authorization.
- Cookie.
- Referer.
- User-Agent.
- Más información.

Introducción

- Las **cabeceras o headers HTTP**, como su nombre lo indica, son parte del Protocolo de Transferencia de Hipertexto (HTTP) y se encargan de transmitir **información adicional** durante una **solicitud** o una **respuesta** HTTP.

- Asimismo, además de compartir datos, estas cabeceras facilitan el **intercambio de metainformación** acerca de un documento entre un navegador y un servidor.

- En este sentido, una **solicitud HTTP** contiene un **área de headers** con información tal como la **fecha** de solicitud, el referente o el **idioma preferido**, mientras que las **respuestas HTTP** también contienen un campo de cabecera donde el servidor envía la información pertinente al navegador de la usuaria o usuario.

- Nota: cabe destacar que todo este intercambio de información, por lo general, es totalmente **invisible** para los internautas.

Introducción

- Básicamente, la estructura de las **cabeceras HTTP** incluyen campos que constan de una línea que a su vez contienen un **par de nombre/valor** llamado **par clave/valor**, que están separados por **dos puntos** y terminan con un **salto de línea**.

- Estos valores a emplear para el header HTTP, están ya definidos en el **RFC** (“Request for Comments", una serie de documentos donde se describen las reglas y el funcionamiento de internet y otras redes).

- Además de los campos especificados, existen **otras cabeceras** que **no son estándares** y que se emplean para añadir **información definida** por el usuario.

- A continuación, se muestran algunos ejemplos de posibles campos de cabeceras de una solicitud.

Cabeceras Accept

- Son varios los campos "accept" que podemos encontrar en los headers y, aunque su funcionalidad es en principio parecida, cada campo se encarga de especificar aún más el tipo de documento admitido.
- **Accept**:
  - Este campo informa al servidor sobre qué tipo de datos se pueden retornar.
- **Accept-Charset**:
  - El campo de la cabecera HTTP “Accept Charset" se utiliza para especificar cuál es la codificación de caracteres que acepta la aplicación de cliente para responder a la solicitud. Su sintaxis es:

```
Accept-Charset: character-set
```

```
Accept-Charset: iso-8859-5, Unicode-1-1; q = 0,8
```

Cabeceras Accept

- **Accept-Encoding**:
  - El campo Accept-Encoding limita los algoritmos de codificación que son aceptables en la respuesta. Esta es su sintaxis:

```
Accept-Encoding: encodings
```

  - Ejemplos:

```
Accept-Encoding: gzip
```

```
Accept-Encoding: *
```

```
Accept-Encoding: gzip;q=0.7
```

Cabeceras Accept

- **Accept-Language**:
  - Si el header incluye el campo “Accept-Language" le comunica al servidor qué lenguaje humano legible espera que retorne.
  - Nota: esto es solo una indicación y no puede ser controlada al 100%. En estos casos el servidor siempre debe evitar sobrescribir una selección explícita de la usuaria/o.
  - Esta es su sintaxis:

```
Accept-Language: <language>; q=qvalue
```

  - Además, si se añaden diferentes idiomas, se pueden separar entre comas. Por ejemplo:

```
Accept-Language: en-US; q=0.9
```

Authorization

- El campo de “Authorizatión" se emplea en las cabeceras HTTP para autenticar a un “user agent" (la app que funciona como “cliente”) con el servidor.
- Su sintaxis es la que sigue:

```
Authorization: <type> <credentials>
```

Cookie

- Esta cabecera de solicitud HTTP contiene **las cookies del HTTP** almacenadas en pares de nombre/valores enviados previamente por el servidor usando el campo “set cookie".
- Sin embargo, los navegadores pueden bloquear este comportamiento y dejar de transmitir las cookies al servidor.
- Por ejemplo:

```
Cookie: nombre1=valor1; nombre2=valor2; nombre3=valor3
```

Referer

- El header **Referer** le permite a la app cliente o al navegador indicar la dirección URL de la fuente desde donde se está emitiendo la petición.
- Su sintaxis general se ve así:

```
Referer: URL
```

- Ejemplo:

```
Referer: https://www.ematiz.com
```

User-Agent

- La cabecera HTTP que incluye este campo envía información sobre la app cliente a un servidor.
- Su sintaxis puede ser como la siguiente:

```
User-Agent: <product> / <product-version> <comment>
```

- Ejemplo:

```
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
110.0.0.0 Safari/537.36
```

Más información

- Para consultar todos los campos de headers para solicitudes y respuestas, se recomienda visitar:
  - https://hmong.es/wiki/List_of_HTTP_header_fields

Códigos de estado

- Introducción.
- Respuestas informativas.
- Peticiones correctas.
- Redirecciones.
- Errores del cliente.
- Errores del servidor.

Introducción

- Los **códigos de estado HTTP** describen de forma abreviada la respuesta HTTP. Estos códigos están especificados por el **RFC 2616**.
- El primer dígito del código de estado especifica uno de los 5 tipos de respuesta, el mínimo para que un cliente pueda trabajar con HTTP es que reconozca estas 5 clases. La Internet Assigned Numbers Authority (IANA) mantiene el registro oficial de códigos de estado HTTP.
- Así pues, puedes obtener códigos divididos por centenas como:
  - 1XX: Respuestas informativas.
  - 2XX: Peticiones correctas.
  - 3XX: Redirecciones.
  - 4XX: Errores del cliente.
  - 5XX: Errores de servidor.

Respuestas informativas

- Este tipo de código de estado indica una **respuesta provisional**.
- HTTP/1.0 no definió ningún estado 1XX, por lo que los servidores no deberían enviar una respuesta 1XX a un cliente HTTP/1.0 salvo para experimentos.
  - **100 Continue**. El servidor ha recibido los headers del request y el cliente debería proceder a enviar el cuerpo de la respuesta.
  - **101 Switching Protocols**. El requester ha solicitado al servidor conmutar protocolos.
  - **102 Processing (WebDAV; RFC 2518)**. Usado en requests para reanudar peticiones PUT o POST abortadas.

Peticiones correctas

- Este **código** de estado indica que la acción solicitada por el cliente ha sido recibida, entendida, aceptada y procesada **correctamente**.

Peticiones correctas

- **200 OK**. El request es correcto. Esta es la respuesta estándar para respuestas correctas.
- **201 Created**. El request se ha completado y se ha creado un nuevo recurso.
- **202 Aceptada**. El request se ha aceptado para procesarlo, pero el proceso aún no ha terminado.
- **203 Non-Authoritative Information**. El request se ha procesado correctamente, pero devuelve información que podría venir de otra fuente.
- **204 No Content**. El request se ha procesado correctamente, pero no devuelve ningún contenido.
- **205 Reset Content**. El request se ha procesado correctamente, pero no devuelve ningún contenido y se requiere que el requester recargue el contenido.
- **206 Partial Content**. El servidor devuelve sólo parte del recurso debido a una limitación que ha configurado el cliente (se usa en herramientas de descarga como wget).
- **207 Multi-Status** (WebDAV; RFC 4918). El cuerpo del mensaje es XML y puede contener un número de códigos de estado diferentes dependiendo del número de sub-requests.

Redirecciones

- El **cliente** ha de tomar una acción adicional para **completar el request**.
- Muchos de estos estados se utilizan para **redirecciones**.
- El **user-agent** puede llevar a cabo la **acción adicional sin necesidad de que actúe el usuario** sólo si el método utilizado en la segunda petición es **GET o HEAD**.
- Un **user-agent** no debería redireccionar automáticamente un request más de cinco veces, ya que esas redirecciones suelen indicar un infinite loop.

Redirecciones

- **300 Multiple Choices**. Es una lista de enlaces. El usuario puede seleccionar un enlace e ir a esa dirección. Hay un máximo de cinco direcciones.
- **301 Moved Permanently**. La página solicitada se ha movido permanentemente a una nueva URI.
- **302 Found**. La página solicitada se ha movido temporalmente a una nueva URI.
- **303 See Other**. La página solicitada se puede encontrar en una URI diferente.
- **304 Not Modified**. Indica que la página solicitada no se ha modificado desde la última petición.
- **305 Use Proxy** (desde HTTP/1.1). El recurso solicitado sólo está disponible a través de proxy, cuya dirección se proporciona en la respuesta. Muchos clientes HTTP como Mozilla o Internet Explorer no manejan bien estas respuestas con estos códigos de estado, sobre todo por seguridad.

Redirecciones

- **307 Temporary Redirect** (desde HTTP/1.1). La página solicitada se ha movido temporalmente a otra URL. En este caso el recurso debería de repetirse con otra URI, sin embargo, futuros requests deberán usar la URI original. Al contrario que con la 302, el método request no puede cambiar cuando se reubique el request original.

- **308 Permanent Redirect** (RFC 7538). El request y futuros requests deberían repetirse usando otro URI. Este también es similar al 301, pero no permite al método HTTP que cambie.

Errores del cliente

- Excepto cuando se responde a un HEAD request, el servidor debe incluir una entidad que contiene una explicación del error, y si es temporal o permanente.
- Son aplicables a cualquier método de solicitud (GET, POST...).
- Los user agents deben mostrar cualquier entidad al usuario.

Errores del cliente

- **400 Bad Request**. El servidor no puede o no va a procesar el request por un error de sintaxis del cliente.
- **401 Unauthorized** (RFC 7235). Similar al error 403, pero se usa cuando se requiere una autentificación y ha fallado o todavía no se a facilitado.
- **402 Payment Required**. Reservado para futuro uso. La intención original fue para pago con tarjeta o micropago, pero eso no ha ocurrido, y este código apenas se usa.
- **403 Forbidden**. El request fue válido pero el servidor se niega a responder.
- **404 Not Found**. El recurso del request no se ha podido encontrar pero podría estar disponible en el futuro. Se permiten requests subsecuentes por parte del cliente.
- **405 Method Not Allowed**. Se ha hecho un request con un recurso usando un método request no soportado por ese recurso (por ejemplo usando GET en un formulario que requiere POST).
- **406 Not Acceptable**. El recurso solicitado solo genera contenido no aceptado de acuerdo con los headers Accept enviados en el request.

Errores del cliente

- **407 Proxy Authentication Required** (RFC 7235). El cliente se debe identificar primero con el proxy.
- **408 Request Timeout**. El cliente no ha enviado un request con el tiempo necesario con el que el servidor estaba preparado para esperar. El cliente podría repetir el request sin modificaciones más tarde.
- **409 Conflict**. Conflicto en el request, como cuando se actualizan al mismo tiempo dos recursos.
- **410 Gone**. El recurso solicitado no está disponible ni lo estará en el futuro. Un buscador eliminará antes una página 410 que una 404.
- **411 Length Required**. El request no especificó la longitud del contenido, la cual es requerida por el recurso solicitado.
- **412 Precondition Failed** (RFC 7232). El servidor no cumple una de las precondiciones que el requester añade en el request.

Errores del cliente

- **413 Request Entity Too Large**. El request es más largo que el que está dispuesto a aceptar el servidor.
- **414 Request-URI Too Long**. El URI es muy largo para que el servidor lo procese.
- **415 Unsupported Media Type**. La entidad request tiene un media type que el servidor o recurso no soportan.
- **416 Requested Range Not Satisfiable** (RFC 7233). El cliente ha solicitado una porción de archivo, pero el servidor no puede ofrecer esa porción.
- **417 Expectation Failed**. El servidor no puede cumplir los requerimientos del header del request Expect.
- **418 I'm a teapot (RFC 2324)**. Fue parte de un April Fool's day, y no se espera que se implemente en servidores HTTP. La RFC especifica que este código debería ser devuelto por teteras para servir té.

Errores del servidor

- El **servidor** ha fallado al completar una solicitud aparentemente válida.
- Cuando los **códigos de estado** empiezan por **5** indica casos en los que el **servidor** sabe que tiene un **error** o realmente es **incapaz de procesar el request**.
- Salvo cuando se trata de un **request HEAD**, el servidor debe incluir una entidad conteniendo una explicación del error, y de si es temporal o permanente.
- Igualmente los **user-agents** deberán mostrar cualquier entidad al usuario.
- Estos códigos de respuesta se aplican **a cualquier método request**.

Errores del servidor

- **500 Internal Server Error**. Error genérico, cuando se ha dado una condición no esperada y no se puede concretar el mensaje.
- **501 Not Implemented**. El servidor o no reconoce el método del request o carece de la capacidad para completarlo. Normalmente es algo que se ofrecerá en el futuro, como un nuevo servicio de una API.
- **502 Bad Gateway**. El server actuaba como puerta de entrada o proxy y recibió una respuesta inválida del servidor upstream.
- **503 Service Unavailable**. El servidor está actualmente no disponible, ya sea por mantenimiento o por sobrecarga.
- **504 Gateway Timeout**. El servidor estaba actuando como puerta de entrada o proxy y no recibió una respuesta oportuna por parte del servidor upstream.
- **505 HTTP Version Not Supported**. El servidor no soporta la versión del protocolo HTTP usada en el request.
- **511 Network Authentication Required** (RFC 6585). El cliente necesita autentificarse para poder acceder a la red.

Hipermedia

- **Hipertexto, multimedia e hipermedia** son conceptos **interrelacionados**.
- Son herramientas interactivas que sirven como un **medio de comunicación** y relacionan varias áreas del conocimiento humano tales como Ciencias de la Comunicación, Ergonomía y factores humanos, Informática, Psicología y otros.
- La **multimedia** consiste en integrar **diferentes medios** (texto, imágenes estáticas, imágenes en movimiento, audio, vídeo) controlados por un usuario (interactividad), lo que proporciona una gran riqueza en los tipos de datos, dotando de **mayor flexibilidad a la expresión de la información**.
- **Textos, imágenes y otros tipos de contenidos** se van secuenciando de una forma dinámica, mejorando notablemente la atención, la comprensión y el aprendizaje.
- El entorno con que se diseñan los **programas multimedia** mantienen **cautivos a los usuarios**, ya que la información solicitada puede presentarse en forma **audible, visible o ambas**.

Hipermedia

- **Hipermedia** es el término con el que se designa al **conjunto de métodos o procedimientos** para escribir, diseñar o componer contenidos que integren soportes tales como **texto, imagen, vídeo, audio, mapas** y otros soportes de información, de tal modo que el resultado obtenido, además, tenga la posibilidad de **interactuar con los usuarios**.

- Por lo tanto, la **hipermedia** conjuga la tecnología **hipertextual** con la multimedia.

- Si la **multimedia** proporciona una **gran riqueza en los tipos de datos**, el **hipertexto** aporta una estructura que permite que los datos puedan **presentarse y explorarse** siguiendo distintas secuencias, de acuerdo a las necesidades y preferencias del **usuario**.

Servicios CRUD

- El **concepto CRUD** está estrechamente vinculado a la **gestión de datos digitales**.
- **CRUD** hace referencia a un **acrónimo** en el que se reúnen las **primeras letras** de las **cuatro operaciones fundamentales** de aplicaciones persistentes en sistemas de bases de datos:
  - **C**reate (Crear registros).
  - **R**ead (Leer registros).
  - **U**pdate (Actualizar registros).
  - **D**elete. Destroy (Borrar registros).

Servicios CRUD

- En pocas palabras, **CRUD** resume las **funciones requeridas** por un usuario para **crear y gestionar** datos.
- Varios procesos de gestión de datos están basados en **CRUD**, en los que dichas operaciones están específicamente adaptadas a los requisitos del sistema y de usuario, ya sea para la gestión de bases de datos o para el uso de aplicaciones.
- Para los expertos, las operaciones son las **herramientas de acceso típicas e indispensables** para comprobar, por ejemplo, los problemas de la base de datos, mientras que para los usuarios, CRUD significa crear una cuenta (créate) y utilizarla (read), actualizarla (update) o borrarla (delete) en cualquier momento.

Servicios CRUD

- Dependiendo de la configuración regional, las operaciones CRUD pueden implementarse de diferentes maneras, como lo muestra la siguiente tabla:

CRUD-Operation	SQL	RESTful HTTP	XQuery
Create	INSERT	POST, PUT	insert
Read	SELECT	GET, HEAD	copy/modify/return
Update	UPDATE	PUT, PATCH	replace, rename
Delete	DELETE	DELETE	delete

HATEOAS

- Introducción.
- ¿Qué significa HATEOAS?
- HATEOAS y REST: el paradigma hipermedia.
- HATEOAS aplicado a una tienda online.
- Ejemplo para la comunicación servidor-cliente.
- Spring HATEOAS.

Introducción

- La transferencia de estado representacional, más conocida por **REST** (siglas del inglés Representational State Transfer), forma parte de los **paradigmas de programación más importantes** en el desarrollo de aplicaciones.

- Este principio de arquitectura, presentado en el año 2000 por Roy Fielding en su disertación, tiene como **objetivo principal** adaptar las **aplicaciones web** lo mejor posible a los requisitos de la Web moderna y, como recientemente ha ganado en popularidad y relevancia, cada vez son más los webmasters que se esfuerzan en aplicar este concepto.

- Los resultados de este esfuerzo, sin embargo, no son siempre realmente REST (RESTful), porque a menudo **ciertos principios o propiedades como HATEOAS** no se implementan correctamente.

¿Qué significa HATEOAS?

- El término **HATEOAS**, introducido por Fielding en su definición de REST, es el acrónimo de **Hypermedia as the engine of applicacion state** (en castellano, hipermedia como el motor del estado de la aplicación) y describe una de las **propiedades más significativas de REST**: como este enfoque de diseño de aplicaciones ha de ofrecer **una interfaz universal**, lo que postula HATEOAS es que el **cliente** pueda **moverse por la aplicación web** únicamente siguiendo a los **identificadores** únicos URI en **formato hipermedia**.

- Cuando se aplica este principio, el **cliente**, aparte de una **comprensión básica de los hipermedia**, **no** necesita **más información** para poder interactuar con la aplicación y el servidor.

¿Qué significa HATEOAS?

- Estas URIs pueden presentarse:
  - En forma de **atributos href** y **src** si hacen referencia a **documentos HTML** o a snippets,
  - Con **elementos o atributos** JSON o XML que los clientes pueden reconocer de **forma automática.**
- La aplicación del principio HATEOAS permite que la **interfaz de un servicio REST** pueda **modificarse siempre que se requiera**, lo que constituye una ventaja fundamental de esta arquitectura frente a otras, como las basadas en **SOAP** (Simple Object Access Protocol).

HATEOAS y REST: el paradigma hipermedia

- Para comprender el significado de **HATEOAS** para las aplicaciones REST, puede ayudar entender primero el marco general. Tal como Fielding define su concepto, son **5 los principios fundamentales** que ha de cumplir un servicio para que sea considerado REST.

- En **primer lugar**, ha de basarse siempre en una **estructura cliente-servidor** que permita la **comunicación sin estado** entre el servidor y los clientes, lo que significa que todas las peticiones de los clientes al servidor se tratan **de forma independiente** a peticiones anteriores.

- El servicio debe estar **estructurado en capas** y utilizar las **ventajas del caching HTTP** (almacenamiento en caché) para que la utilización del servicio sea lo más sencilla posible para el cliente que realiza la petición.

- Por último, ha de tener una **interfaz unitaria**.

HATEOAS y REST: el paradigma hipermedia

- Para la conexión entre el **servidor y el cliente** Fielding define estas **cuatro características**:
  - **Identificación inequívoca de todos los recursos**: todos los recursos han de poder identificarse con una URI (Unique Resource Identifier).
  - **Interacción con los recursos por medio de representaciones**: si un cliente necesita un recurso, el servidor le envía una representación (p. ej., HTML, JSON o XML) para que el cliente pueda modificar o borrar el recurso original.
  - **Mensajes explícitos**: cada mensaje intercambiado por el servidor y el cliente ha de contener **todos los datos necesarios** para entenderse.
  - **HATEOAS**: este principio también integra una API REST. Esta estructura basada en hipermedia facilita a los clientes el **acceso a la aplicación**, puesto que de este modo no necesitan saber nada más de la interfaz para poder acceder y navegar por ella.
- **HATEOAS** es, en definitiva, **una de las propiedades más elementales de las API REST** y como tal, imprescindible en cualquier servicio REST.

HATEOAS aplicado a una tienda online

- Para hacer más **comprensible** el concepto **HATEOAS**, aclararemos a continuación **cómo se lleva a la práctica** con el ejemplo de una **tienda online**.

- Veremos cómo HATEOAS se realiza por medio de hiperenlaces, como es habitual en el diseño de aplicaciones web.

- Estos enlaces, entre dos documentos o a otro punto dentro del mismo documento, se marcan en HTML de esta forma:

```
<a href="URI">Texto</a>
```

- Los medios (texto, gráficos, audio, video) enlazados de esta forma se denominan **hypermedia**. En el caso de las **aplicaciones web** suele tratarse de **documentos de texto simples en formato HTML**, que en este contexto también se entienden como los diversos “estados” (del inglés “states”) de una aplicación. Dentro de la filosofía REST, cada uno de estos documentos tiene un **identificador único URI** que permite la comunicación con ellos.

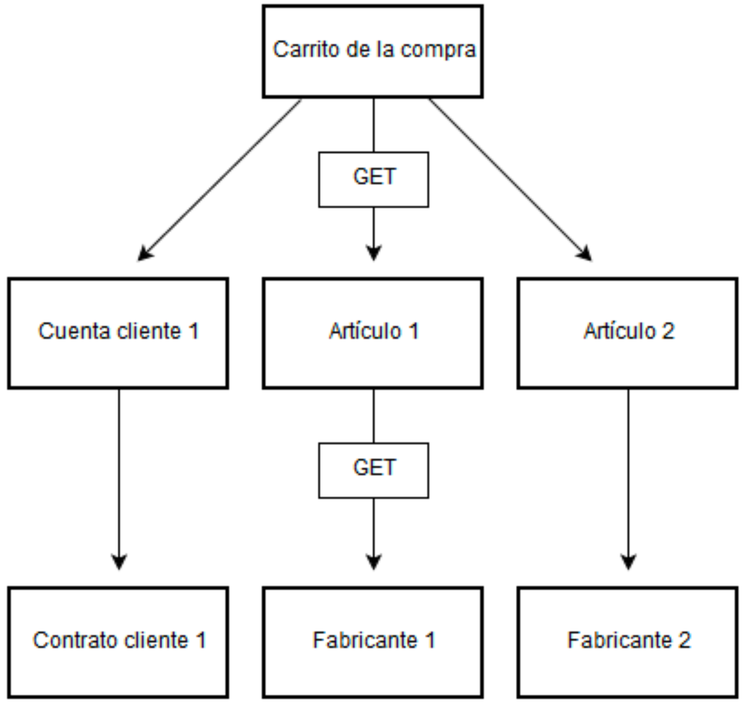
HATEOAS aplicado a una tienda online

- Si aplicamos este concepto a una tienda online, este es el resultado:
- Al documento que describe el estado “carrito de la compra" se le ha asignado una URI fija (como, p. ej., 'https://ejemplo.es/carritodelacompra').
- Las URIs para cada uno de los artículos que pueden depositarse en la cesta de la compra siguen el mismo esquema:
  - 'https://ejemplo.es/articulo/1',
  - 'https://ejemplo.es/articulo/2',
  - etc.
- Otro posible estado es la “cuenta de usuario", a la que puede accederse desde el carrito de la compra y que podría tener este URI: 'https://ejemplo.es/cliente/1'. A su vez, cada uno de estos documentos/ estados contiene los hiperenlaces a las acciones que el usuario podría llevar a cabo a continuación.

HATEOAS aplicado a una tienda online

- Para el **documento de la cesta de la compra** esto significa que ha de contener enlaces a las URIs de los artículos y los clientes; estos por su parte, a los fabricantes o a los contratos.

- El **cliente navega** de este modo por la **tienda o por el carrito de la compra** gracias a los diversos **hiperenlaces vía GET request**, como se ve en el siguiente gráfico:



HATEOAS aplicado a una tienda online

- En un **servicio REST** que siga el principio HATEOAS tal y como lo definió Fielding el usuario solo ha de conocer **la URI inicial** para poder interactuar con la oferta según el plan del desarrollador.

Ejemplo para la comunicación servidor-cliente

- Esta **estructura** sirve también para explicar **cómo** tiene lugar la comunicación entre el cliente y el servidor.
- Con el siguiente código la aplicación cliente hace **una petición para obtener** una representación XML del estado “carrito de la compra”:

```
GET https://ejemplo.es/carritodelacompra HTTP/1.1
Host: https://ejemplo.es
Accept: application/xml
```

Ejemplo para la comunicación servidor-cliente

- El servidor podría responder entonces:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: 256

<?xml version="1.0"?>
<carritodelacompra>
 <carritodelacompra_numero>1</carritodelacompra_numero>
 <link rel="cuenta" href="https://ejemplo.es/cliente/1" />
 <link rel="articulo1" href="https://ejemplo.es/articulo/1" />
 <link rel="articulo2" href="https://ejemplo.es/articulo/2" />
</carritodelacompra>
```

Otro ejemplo

- Supongamos que tenemos **una API REST** para gestionar entidades de tipo **cliente** y sus diferentes **pedidos**. Cada cliente tendrá asociado una **lista de pedidos**.
- Cuando se solicite información de un **cliente** mediante la siguiente URI:

```
URI/cliente/:id
```

- Devolvemos los datos del **cliente** con **su lista de pedidos**.
- Podemos encontrarnos con el problema de que tenga **muchos pedidos** y, por otro lado, puede que la aplicación solo necesite **datos del cliente, no de los pedidos**.
- Por lo tanto, estoy **recuperando más información** de lo que debiera.

Otro ejemplo

- Una solución sería tener **otro servicio REST** que me devuelva los pedidos. Por ejemplo, esta URI me devolverá los **datos del cliente 2**:

```
URI/cliente/2
```

- Esta URI me devolverá los datos del cliente 2 que podría ser:

```
{ id:2, nombre:"Pepe", apellido:"Gomez" }
```

- Y podríamos hacer **otro servicio** que nos devuelva los pedidos por medio de la siguiente URI:

```
URI/cliente/2/pedidos
```

- Supongamos que por algún problema debemos **cambiar la URI de pedidos**, para hacer esto debemos **avisar a todos los clientes** y ellos deberán **consumir la nueva URI**.

Otro ejemplo

- **HATEOAS** nos dice que **debemos mostrar las URIs** de forma que el **cliente esté menos acoplado** al servidor y no sea necesario hacer cambios en estos casos.

- Por ejemplo, el **servicio de consulta de clientes** puede devolver la **URI del pedido** de la siguiente manera:

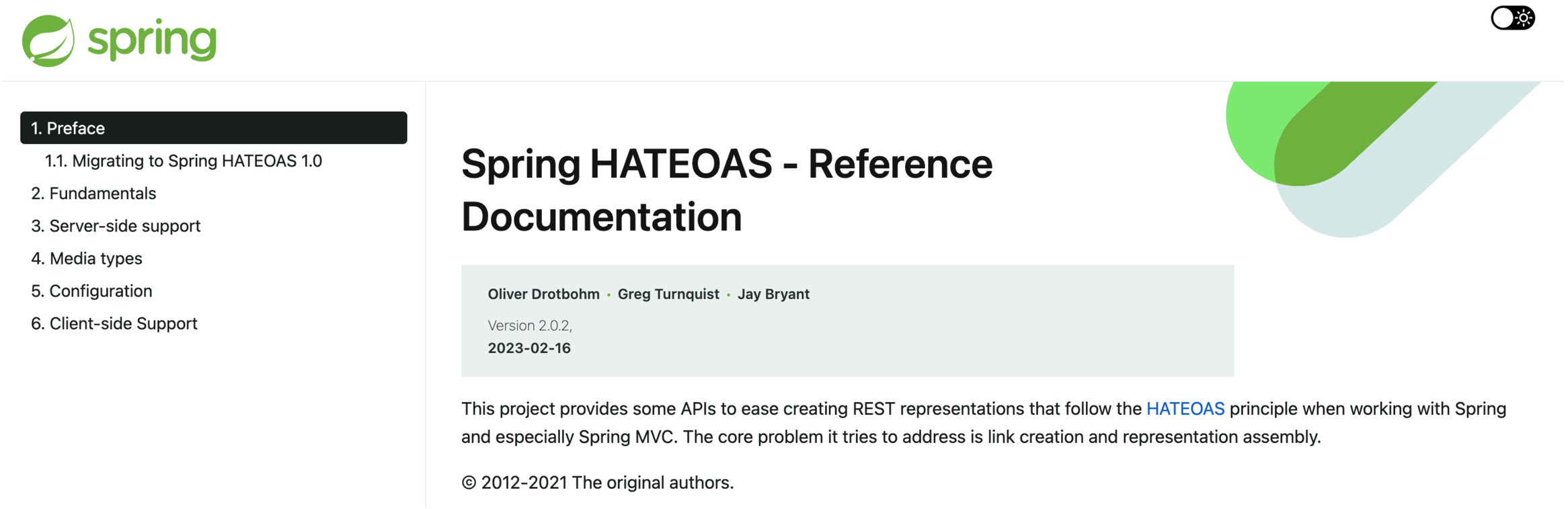
```
{
  id:2,
  nombre:"Pepe",
  apellido:"Gomez" ,
  pedidos:"URL/cliente/2/pedidos "
}
```

Otro ejemplo

- Si **no mostramos la URI** de los pedidos el **cliente REST** tiene que ir a la **documentación de la API** por cada servicio que desee utilizar.
- Con **HATEOAS** los **servicios descubren nuevos servicios** lo que permite navegar por la información y un cliente con **una URI principal** puede ir **descubriendo los demás servicios**.
- La restricción HATEOAS **desacopla al cliente** y al **servidor** de una manera que permite que la funcionalidad del servidor evolucione de forma independiente.

Spring HATEOAS

- https://docs.spring.io/spring-hateoas/docs/current/reference/html/



Buenas prácticas para el diseño de una API RESTful

- Introducción.
- Usar nombres pero no verbos.
- Métodos GET y los parámetros de consulta no deben alterar el estado.
- Usar nombres en plural.
- Usar subrecursos para establecer relaciones.
- Usar cabeceras HTTP para la serialización de formatos.
- Filtrado, Ordenación, Paginación y Selección de campos.
- Versionar la API.
- Manejar errores con código de estado HTTP.

Introducción

- A medida que pasa el tiempo y **se van diseñando servicios** que tienen que escalarse e integrarse con otros servicios, nos vamos dando **cuenta de lo necesario que es un buen diseño** o, por lo menos, de un diseño mínimamente bien pensando y ejecutado que cumpla una serie de requisitos mínimos.

- Procedemos a presentar **una serie de recomendaciones**, que pueden considerarse buenas prácticas, y que debemos tener en cuenta para asegurar un **buen diseño de APIs RESTful**.

- El concepto subyacente de una API RESTful es la de **dividir la estructura de la API en recursos lógicos**, montados sobre una URL, que nos permitan acceder a información o datos concretos usando **métodos HTTP como POST, GET, PUT y DELETE** para operar con los recursos.

- Estos métodos son equiparables a las **operaciones CRUD** (Create, Read, Update, Delete) de las bases de datos.

Usar nombres pero no verbos

- Para mejorar la **comprensión y legibilidad de la semántica de la API**, usa sólo sustantivos para toda su estructura.

RECURSO	POST CREATE	GET READ	PUT UPDATE	DELETE
/cars	Crea un nuevo coche	Devuelve la lista de coches	Actualización en bloque de coches	Borrar todos los coches
/cars/711	Método no permitido (405)	Devuelve un coche específico	Actualiza un coche específico	Borra un coche específico

- No usar verbos como:
  - /getAllCars
  - /createNewCar
  - /deleteAllRedCar

Usar nombres pero no verbos

- La **propia semántica** que compone la URL, junto con el método HTTP usado (POST, GET, PUT, DELETE), permite **prescindir de verbos** como «create», «get», «update» y «delete».

Métodos GET y los parámetros de consulta no deben alterar el estado

- Usa **los métodos POST, PUT o DELETE** para cambiar los estados de los recursos, en vez de hacerlo con el método GET o un petición con parámetros de consulta.
- Esto es lo que no se debería hacer:

```
GET /users/711?activate
GET /users/711/activate
```

- Es mucho mejor elegir una de estas opciones:

```
PUT /users/711/activate
```

```
POST /users/711/activate
```

Usar nombres en plural

- No debemos mezclar **nombres en singular** con **nombres en plural**. Debemos hacerlo simple y mantener los nombres en plural. En vez de usar:

```
/car
/user
/product
/setting
```

- Debemos usar el plural:

```
/cars
/users
/products
/settings
```

Usar nombres en plural

- El **uso del plural** nos permitirá posteriormente **hacer operaciones del tipo**:

```
/cars/<id>
/users/<id>
/products/<id>
/settings/<name>
```

Usar subrecursos para establecer relaciones

- Si un recurso está relacionado con otro, debemos usar subrecursos montados sobre la estructura de la URL.
- Ejemplo: Devuelve una lista de conductores para el coche 711.

```
GET /cars/711/drivers/
```

- Ejemplo: Devuelve el conductor #4 para el coche 711.

```
GET /cars/711/drivers/4
```

Usar cabeceras HTTP para la serialización de formatos

- Tanto la parte cliente como la del servidor, necesitan saber en **qué formato** se están pasando los datos para poder comunicarse.
- Lo más sencillo es especificarlo en la cabecera HTTP.
  - **Content-Type**: Define el formato de la petición.
  - **Accept**: Define la lista de formatos aceptados en la respuesta.

Filtrado

- Utilizar un parámetro de consulta único para todos los campos o un lenguaje de consulta formalizado para filtrar.
- Ejemplo: Devuelve una lista de coches rojos.

```
GET /cars?color=red
```

- Ejemplo: Devuelve una lista de coches con un máximo de 2 plazas.

```
GET /cars?seats<=2
```

Ordenación

- Permitir ordenación ascendente o descendente sobre varios campos.
- Ejemplo:

```
GET /cars?sort=-manufactorer,+model
```

- En el ejemplo se devuelve una lista de coches ordenada con los fabricantes de manera descendiente y los modelos de manera ascendente.

Paginación

- Uso de **«offset»** para establecer la posición de partida de una colección y **«limit»** para establecer el número de elementos de la colección a devolver desde el offset. Es un **sistema flexible** para el que consume la API y bastante extendido entre los sistemas de bases de datos.

- Ejemplo:

```
GET /cars?offset=10&limit=5
```

- En el ejemplo se devuelve una lista de coches correspondiente a los coches comprendidos de la posición 10 a la 15 de la colección.

- Para devolver al consumidor de la API el número total de entradas se puede usar la cabecera HTTP «X-Total-Count».

Selección de campos

- Permitir la **selección de unos pocos campos de un recurso** si no se precisan todos.
- Dar al consumidor de la API la posibilidad de **qué campos quiere que se devuelvan**, reduce el volumen de tráfico en la comunicación y aumenta la velocidad de uso de la API.
- Ejemplo:

```
GET /cars?fields=manufacturer,model,color
```

- En el ejemplo se pide la lista de coches, pero pidiendo sólo los **campos de fabricante, modelo y color**.

Versionar la API

- Es muy recomendable poner la versión de API y no liberar APIs sin versión. Usa un simple número para especificar la versión.
- Si se usa la URL para marcar la versión, pon una «v» precediendo el número de versión:

```
/blog/api/v1
```

- Es la forma **más sencilla y eficaz** de asegurar la compatibilidad en el versionado de la API.
- Especificarla como **parámetro o en la petición y/o en la respuesta**, conlleva el aumento del tráfico y el coste de computación, al tener que existir lógica para discriminar las distintas versiones soportadas sobre una misma URL.

Manejar errores con código de estado HTTP

- Introducción.
- Usar códigos de estado HTTP.
- Usar el payload para los errores.

Introducción

- Es difícil trabajar con una API que **ignora el manejo de errores**, por no decir imposible.
- La devolución de un **código HTTP 500** para cualquier tipo de error o acompañado de una traza de error del código del servidor **no es muy útil**, además de que puede ponernos en peligro ante una vulnerabilidad que un atacante quisiera explotar.

Usar códigos de estado HTTP

- El estándar HTTP proporciona más de **70 códigos de estado para describir los valores de retorno**. En general no los utilizaremos todos, pero se debe utilizar por lo menos un mínimo de 10, que suelen ser los más comunes:

CÓDIGO	SIGNIFICADO	EXPLICACIÓN
200	OK	Todo está funcionado.
201	OK	Nuevo recurso ha sido creado.
204	OK	El recurso ha sido borrado satisfactoriamente.
304	No modificado	El cliente puede usar los datos cacheados.
400	Bad Request	La petición es inválida o no puede ser servida. El error exacto debería ser explicado en el payload de respuesta.
401	Unauthorized	La petición requiere de la autenticación del usuario.
403	Forbidden	El servidor entiende la petición pero la rechaza o el acceso no está permitido.
404	Not found	No hay un recurso tras la URI.
422	Unprocessable Entity	Debe utilizarse si el servidor no puede procesar la entidad. Por ejemplo, faltan campos obligatorios en el payload.
500	Internal Server Error	Los desarrolladores de APIs deben evitar este error. Si se produce un error global, el stracktrace se debe registrar y no devolverlo como respuesta.

Usar el payload para los errores

- Todas las excepciones deben ser asignadas en un payload de error. A continuación se muestra un ejemplo de cómo podría verse un payload de error en formato JSON:

```
{
  "errors": [
   {
    "userMessage": "Sorry, the requested resource does not exist",
    "internalMessage": "No car found in the database",
    "code": 34,
    "more info": “http://empresa.com/blog/api/v1/errors/12345”
   }
  ]
}
```