REST

REST es una forma de escribir servicios web apoyándose lo más posible en http como capa de aplicación.
El acrónimo REST responde a "REpresentational State Transfer". El estilo arquitectónico REST fue descrito por primera vez por Roy Thomas Fielding, allá por el año 2000.
Este documento trata de recoger las mejores practicas encontradas hasta el momento relacionadas con el diseño de APIs REST, si bien la mayoría de los apartados se establecen pautas claras a seguir, existen otros aspectos en los que dependerá del contexto en el que se diseña.
Está contada aproximadamente de forma que se van evaluando las distintas opciones para cada aspecto de la API del mismo modo que uno leería una URL que define un recurso REST, desde el protocolo que se usa, hasta el formato de respuesta de la petición.
Dado que REST no es un estándar sino un estilo arquitectónico, existen variantes más o menos definidas que se caracterizan por el objetivo que persiguen, si bien es cierto que en nuestro caso esta API no será pública, será consumido por todo tipo de clientes (web, móvil, gadget y widgets). Por lo que trataremos de seguir las prácticas

Protocolo

Por motivos de seguridad, el protocolo a utilizar deberá ser **https**. Con certificados firmados por una entidad certificada.
Las APIs basados en HTTP sin SSL son susceptibles de ataques basados en "man-in-the-middle".
Dada la sensibilidad de datos de usuario que se manejan, es lo más recomendable.

Dominio

Sería conveniente establecer un subdominio relacionado con el dominio principal para **identificar las peticiones** a la API REST e incluso que fueran a una máquina específica para ello, por ejemplo**api.domain.com**.
En algunas ocasiones, es interesante ofrecer otro subdominio donde alojar **la documentación** para el desarrollo para esta API, como por ejemplo **developers.server.com o dev.server.com** (facebook, twitter y foursquare siguen esta línea ofreciendo ambos subdominios).

Versiones

Never *release an API without a version and make the version mandatory.* Brian Mullo, apigee

Numeración

Las versiones deben numerarse con enteros simples, sin '.' y con el carácter 'v' delante.

```
v1, v1, v2, v2, v14,...
```

No es recomendable utilizar decimales en las versiones ya que el versionado de la API no debería ocurrir a menudo, y los decimales da sensación de que es altamente cambiante y en consecuencia, poco estable.

Política de versionado

Las versiones no indican un estado exacto de la API sino su nivel de retrocompatibilidad, en particular se debe seguir la siguiente política:

- Cambios que no requiere generar versión:
  - Nuevos recursos.
  - Nuevos métodos HTTP en recursos existentes.
  - Nuevos formatos de datos.
  - Nuevos atributos en recursos existentes.
- Cambios que si requieren generar versión:
  - Recursos movidos o eliminados
  - Cambios en las estructuras de datos de retorno.
  - Cese de soporte para cualquier recurso ("marked as deprecated").

Selección de versión

- **Version en la ruta**
- Las versiones deben de estar en el ámbito más alto de la ruta.
  - `===>`
    `GET /v1/dogs/fasd1 HTTP/1.1`
    `Accept: application/json`
-
- Las ventajas de utilizar esta estrategia de versionado son púramente prácticas y son las que mayor adpción tienen.
- Es práctico, fácil de aplicar y de adoptar, y no rompe mucho con el REST dogmático.
- **Versión en la cabecera HTTP**
- Siendo la solución más elegante, la que mayor interoperabilidad ofrece y la más puramente REST, no es la más extendida. Sería deseable adptarlo, pero la opción descrita anteriormente es totalmente válida.
  - `===>`
    `GET /customer/123 HTTP/1.1`
    `Accept: application/vnd.company.myapp.customer-v3+json`

    `<===`
    `HTTP/1.1 200 OK`
    `Content-Type: application/vnd.company.myapp-v3+json`

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

Mantenimiento

Lo normal es mantener al menos una versión anterior.
Las versiones obsoletas, siempre que se pueda, se devolverá el código de estado de redirección 3XX al recurso nuevo.

Recursos

Como bien indica el propio nombre, REST está orientado a recursos, por lo que hay que tener especial cuidado en la selección de recursos accesibles vía API y como se nombran éstos.

Básicamente para cualquier API REST, existen **2 tipos de recursos: entidades y colecciones**.
En particular se recomiendan las siguientes pautas:

- Minimizar el número de "cosas" que expone la API siempre que se pueda, ayuda a tener una API clara, simple y legible.
- Evitar verbos en las URL. REST define recursos, cosas, no operaciones, los verbos se establecen en la cabecera HTTP.
- Utilizar plurales. El uso de nombres en plural enfatiza el significado de que los recursos se distinguen en colecciones y entidades (/dogs y /dogs/1234, respectivamente).
- Evitar nombres demasiado genéricos tipo /items, /assets, /element, ...

Aunque a continuación se expliquen como son las operaciones CRUD en REST, es importante asimilar que normalmente los servicios que se publican son de mayor nivel, o recursos de alto nivel, hay que tener cuidado con no caer en la tentación de modelar operaciones REST de forma análoga a una capa de persistencia.
En caso de tener que modelar procesos de negocio, se puede realizar transformando dichos procesos en recursos, con esa idea, uno en vez de /carrito/comprar, lo que realiza es el acceso a una /carrito/compra.

Operaciones

- Todo recurso tiene un **identificador único y universal** (UUID/GUID), no secuencial.
- **Stateless**. El resultado de la operación es independiente de la conversación establecida entre cliente-servidor.
- **Idempotente**. Las operaciones han de devolver el mismo resultado independientemente del número de veces que se llame.
- **Homogénea**. Todas los recursos aceptan operaciones dentro del conjunto de acciones establecido. Asi mismo, las URL y las respuestas han de ser también homogéneas.
- Los recursos pueden devolevrse en distintos formatos, **son multimedia**, pueden ser, JSON, XML, HTML, JPG, ..., los que proceda.
- Orientado a recursos. Salvo casos excepcionales, sólo deben **exponer recursos y no operaciones**.
- Deben modelar recursos únicamente de dos tipos: **colecciones y entidades**.

Creación

Existen diferentes formas de creación de recursos en REST, pero muchas de ellas no son seguras ni idempotentes, ambas propiedades son importantes para aumentar la interoperabilidad entre diferentes servicios.
La estrategia que seguiremos para la creación de recursos se basa en la creación de **recursos desechables** o virtuales.
Esta estrategia se basa en que antes de realizar la creación del recurso, realizamos antes una petición POST al recurso padre que la contiene (o la colección que la contiene). Puede ser un recurso "factoría" o un recurso responsable de recación de recursos "virtuales".
En este caso, cuando se produce una creación por PUT, es importante devolver el código 201 (Created) y la URL del recurso recién creado.
El servidor entonces devuelve la URL del recurso "virtual" donde el cliente puede realizar la creación del recurso.
El cliente realiza la creación del recurso con el verbo POST al recurso virtual.

===>

```
POST /users HTTP/1.1
```

```
<===
HTTP/1.1 201 OK
Location: /users/kj123

===>
PUT /users/kj123 HTTP/1.1

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

<===
HTTP/1.1 204 OK
```

Esta estrategia se beneficia de las siguientes propiedades:

- Es segura, pues si el cliente no recibe el recurso virtual, al volver a pedirlo, el servidor le podrá devolver el mismo recurso virtual.
- Es idempotente, el cliente puede realizar varias peticiones PUT al mismo recurso virtual obteniendo el mismo resultado.

Por norma general, no permitiremos inserciones desde el cliente sin establecerse primero el recurso virtual.

Modificación

Para las modificaciones es posible utilizar tanto los métodos **POST y PUT**, pero es importante tener claras sus diferencias.

- **PUT**

PUT tiene una semántica UPSERT, esto quiere decir que si se trata de modificar un recurso que no existe, es posible que se cree. **Aunque para nuestro caso, no permitiremos inserciones de elementos ya que han de crearse a través de recursos virtuales**.
Cuando se utiliza PUT **para actualizar los datos, se envían todos ellos**, como un reemplazo, para que la operación siga siendo idempotente. La respuesta puede ser tanto 200 o 204 dependiendo de si el servidor nos devuelve la misma entidad o no.
En caso de que la modficación realize recalcualaciones en la entidad, es interesante devolver 200 con la nueva entidad.

```
===>
PUT /libros/hr123 HTTP/1.1
Host:api.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}
}
```

```
<===
HTTP/1.1 204 NoContent
```

Si fuera necesario modelar actualizaciones masivas, se podría conseguir especificando una query_string en la actualización, dondeel servidor se encargaría de aplicar los filtros indicados.
En caso de colisionar con las reglas de negocio, devolver código HTTP apropiado.

- **POST**
- POST es la única operación que ni es segura ni es idempotente, se utiliza principalmente para modificaciones parciales o creación de recursos (en nuestro caso, virtuales).

Principalmente usaremos **POST tanto para creación de recursos virtuales, como para actualizaciones parciales**, un ejemplo:

```
POST /libros/hr123 HTTP/1.1
Host:api.server.com
Accept:application/json
Content-Type:application/json

{
   "author": "Jorge Pérez"
}
```

Actualmente se está estandarizando el uso del método **PATCH** para modelar las actualizaciones parciales, aunque no todos los servidores la implementan. De momento no lo tendremos en cuenta

Borrado

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

```
===>
DELETE /libros/hr123 HTTP/1.1
Host:api.server.com

<===
HTTP/1.1 204 NoContent
```

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.
Es posible realizar borrados selectivos haciendo uso de la query_string.

Búsquedas

- **Búsquedas simples**
- El acceso a una entidad determinada puede realizarse estableciendo el identificador en la ruta dentro de la colección deseada, por ejemplo:
  - GET /libros/12341-sdawqe

- **Consultas predefinidas**
- Para las consultas muy recurrentes es posible establecer recursos con relaciones implícitas.
- Para el caso anterior, es posible que se quiera acceder únicamente a cierta parte de dicha entidad, por ejemplo el capítulo 3.
    - GET /libros/12341-sdawqe/chapters/3

-
- La idea es que dentro del propio recurso pueda especificar el nivel de detalle de la información deseada en la propia URL, si bien un ejemplo de esto aplicado a nuestras necesidades podría ser:
    - GET /users/ADS12ed1/block/contact //datos de contacto
      GET /users/ADS12ed1/block/physical //datos físicos
      GET /users/ADS12ed1/block/availability //disponibilidad horaria
      GET /session/today
      GET /activity/favorite

-
- Ayudan a la legibilidad reduciendo drásticamente los parámetros y mejoran la interoperabilidad.
- **Búsqueda avanzadas**
- Toda la potencia de las consultas avanzadas se puede modelar a través de la queryString, dando la flexibilidad a los consumidores de obtener únicamente lo que necesitan.
    - **Filtrar campos**
    - Debe ser posible filtrar campos en las búsquedas a través de la query_string. por ejemplo:
        - GET /libros/hr123?fields=author,price.amount

    -
    - Cabe resaltar la capacidad de poder filtrar por un campo anidado, a través de la notación '.'.
    - **Paginación**
    - Es posible modelar paginaciones utilizando parámetros específicos tipo page y limit:
        - /libros/hr123?fields=author,price.amount&page=2&limit=10

    -
    - En este caso estaríamos obteniendo lo que sería la tercera página si obtuviésemos los elementos de 10 en 10.
    - **Consultas predefinidas**
    - Es posible modelar las anteriores consultas predefinidas a través de la query string, usando la técnica llamada *slicing*, quedando de la siguiente forma:
        - GET /users/ADS12ed1?block="contact" //datos de contacto
          GET /users/ADS12ed1?block="physical" //datos físicos
          GET /users/ADS12ed1?block="availability" //disponibilidad horaria

Resumiendo, básicamente se pueden realizar búsquedas complejas a través de la query_string, y es recomendable anidar más en la URL para búsquedas muy recurrentes ya que simplifican su uso.

Peticiones asíncronas

Para las peticiones que puedan llevar su tiempo en completarse, es posible modelarse como petición asíncrona con HTTP.
Una petición asíncrona devuelve el código 202 mientras se está procesando.

- Petición:
  - ===>

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

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

<===
HTTP/1.1 202 Accepted
Location: /jobs/XDC3WD
```

El servidor nos comunicará que la petición ha sido aceptada pero que se está procesando (202), además nos devuelve el recurso exacto donde podemos ver el progreso de la operación /jobs/XDC3WD, mientras se está procesando la operación asíncrona.
El resultado de la petición a @@:

```
===>
GET /jobs/XDC3WD

<===
HTTP/1.1 200 Ok
Content-Type: application/json

{
   "progress":"working"
}
```

Si el servidor es consciente de progreso exacto de la tarea, es conveniente que devuelva un valor que indique el porcentaje del progreso con el fin de dar un feedback más detallado al consumidor.
Cuando la tarea se haya realizado, la respuesta podría ser la siguiente:

```
===>
GET /jobs/XDC3WD

<===
HTTP/1.1 200 Ok
Content-Type:application/json

{
```

```
  "progress":"ok",
  "result":"/pago/DEF245SW"
}
```

Es importante que al finalizar una tarea se le indique el recurso donde puede acceder al resultado de la tarea asíncrona.

Sobreescritura de métodos

Existen **situaciones en las que el cliente no soporta todos los métodos HTTP**, por lo que es recomendable darles opción a definir el método HTTP que quieren sobreescribir.
Por su naturaleza HTTP tunneling funciona únicamente con GET y el método se establece en la query.

- Crear
  - `GET /dogs?method=post`
- Leer
  - `GET /dogs`
- Actualizar
  - `GET /dogs/1234?method=put&location=park`
- Eliminar
- No es recomendable dar soporte al método DELETE en HTTP tunneling por las implicaciones que tendría, por ejemplo, que un Googlebot conociera dicho recurso.

Este mecanismo también es conocido como HTTP tunneling.

Recursos especiales

Para **peticiones en las que no se ve involucrado ningún recurso**, y sólo en este caso, es posible definir dicho recurso con un método. Como por ejemplo obtener el resultado de aplicar cierto algoritmo en base a unos parámetros.

```
GET /convert?from=EUR&to=CNY&amount=100
```

Este tipo de recurso han de estar claramente documentadas (en que contexto se usa, motivos, etc), y separadas del resto de operaciones típicas.

Resumen

- Esquema de operaciones para colecciones

Método HTTP	Operación
GET	Búsqueda de entidades dentro de la colección
PUT	Actualización múltiple y/o masiva (no se permite creación directa)
DELETE	Borrar la colección y todas sus entidades
POST	Crear una nueva entidad (virtual) dentro de la colección

- Esquema de operaciones para entidades

Método HTTP	Operación
GET	Leer los datos de una entidad en concreto
PUT	Actualizar una entidad existente (no se permite creación directa)
DELETE	Borrar una entidad en concreto
POST	Modificaciones parciales de una entidad ya existente

- Esquema general de métodos HTTP

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

Respuesta

En la respuesta de los servicios REST aparte de proveer toda la información que el cliente solicita, es importante que tenga una estructura bien definida y previsible y que sea extensible para permitir la buena evolución de la API.
En un principio la respuesta de las llamadas a la API REST deberían devolver la información en nivel más alto, ya sea JSON o XML...

```
{
  "id": "/libro/BlVX4BmJJFch7sF_C4DUtaWmJLOf+Cz",
  "author": "Enrique Gómez Salas",
  "title": "Desventuras de un informático en Paris",
```

```
  "genre": "scifi",
  "price": { "currency":"€", "amount":50 }
}
```

En un futuro es posible ampliar el formato de la respuesta aplicando HAL (Hypertext Application Language) como formato de respuesta (MIME type: hal+json). Un ejemplo de respuesta en formato HAL podría ser:

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

Las ventajas de adoptar HAL son numerosas, las más importantes:

- Mayor adopción de la API ya que es autodescriptivo y autodescubrible
- Facilita a los servidores la evolucion de la API de forma independiente a sus consumidores gracias al uso de controles-URL y plantillas-URL

Formato de atributos

El formato de nombres de atributos deberá ser el siguiente:

```
{
  "price": 233.0,
  "description": "iJam4000",
  "createdAt": "12314324243",
  "_links": {...}
}
```

- Los nombres de los atributos deberá ir en minúsculas, sin espacios ni delante, detrás, ni dentro.
- Los atributos con nombres compuestos irá en formato lowerCamelCase
- Los atributos especiales de la API comenzarán con "_".

- Existe quien prefiere establecer el formato fecha siguiendo el estándar ISO-8601 antes que un timestamp de UNIX, pero eso ya es más opcional.

Formatos de respuesta y formatos multimedia (tipos MIME)

Todo recurso REST tiene que soportar diferentes formas de representarlo (y siempre que sea oportuno), **los formatos básicos que debemos soporte deberían ser al menos JSON y XML**.
El formato por defecto en caso de no especificar ninguno debería ser JSON
Lo normal es que el formato venga indicado en las cabeceras Content-Type y Accept. Un ejemplo de Accept con diferentes niveles de preferencia:

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

Con la cabecera Accept se negocia el tipo MIME entre el cliente y servidor, enviando el listado de tipos MIME que el cliente acepte para que el servidor seleccione el formato que más le interese al cliente y que el propio servidor soporte. (en otro caso devuelve un 406).
Aunque es posible especificar el tipo MIME en la propia URL, como en el siguiente ejemplo:

```
http://www.server.com/reserva/22.json
http://www.server.com/reserva/22.pdf
```

Este mecanismo lo utilizaremos únicamente para indicar explícitamente el formato deseado y cuando no exista otra posibilidad. Como cuando se abre un recurso en otra ventana, el navegador no sabe añadir la cabecera Accept con el tipo MIME pdf.
También es posible especificar el tipo como atributo en la propia query, este valor puede aparecer a la vez que la cabecera Accept pero el valor de la query sobreescribirá al de la cabecera.

Tratamiento de errores

Es importante mostrar errores con información de utilidad para los desarrolladores y además ayuda a minimizar el coste de arreglarlos ya que es posible acotarlos rápidamente.
Un ejemplo completo de un formato de error podría ser el siguiente:

```
{
  "status": "401",
  "message": "Bad credentials",
  "type": "Authenticate",
  "code": 20003,
  "info": "http://www.twilio.com/docs/errors/20003",
  "devinfo": "Verbose, plain language description of the problem for the app developer with hints about how to fix it."
}
```

Muy resumidamente se podría recomendar las siguientes prácticas en el tratamiento de errores:

- Apoyarse en los códigos de error que ofrece HTTP (ver detalles más adelante).
- Es recomendable incluir el código HTTP en la propia respuesta pues para algunos consumidores les es más cómodo, si se usa modo suppress_response_codes, este campo es obligatorio.
- Mensajes de error claros y concisos, en lugar de "Error grave en el servidor" y cosas similares.

Este mensaje es un mensaje que debe entenderse independientemente del nivel técnico del leyente. Es un mensaje potencialmente mostrable directamente por el consumidor.
- Devolver un código de error interno con el significado específico del error.
- URL o recurso donde está documentado/explicado dicho error.
- Es muy recomendable mandar información técnica adicional para que los desarrolladores puedan arreglarlo lo antes posible.

suppress_response_codes

Existen situaciones en las que el cliente no tiene posibilidad de recibir la respuesta en caso de producirse un error, como por ejemplo los clientes con flash player.

Por ello es recomendable ofrecer una alternativa en la que el servidor tratará de retornar siempre 200 OK, pero en el mensaje se devolverá la estructura de error anteriormente mencionada, es en este caso cuando el atributo status cobra importancia y pasa a ser obligatorio.

Ejemplo:

```
GET /owners/5678/dogs?suppress_response_codes=true
```

Propuesta de formato de salida

Teniendo en cuenta todos los aspectos relacionados con la respuesta de las peticiones, se propone el siguiente formato de respuesta REST en JSON que engloba todas las buenas prácticas anteriormente mencionadas:

```
{
   "_response": {
      "status": "401",
      "message": "Bad credentials",
      "type": "Authenticate",
      "code": 20003,
      "info": "http://www.twilio.com/docs/errors/20003",
      "devinfo": "Verbose, plain language description of the problem for the app developer with hints about how to fix it."
   },
   "_links": {
      "self": { "href": "/product/Re23SF" },    // Como mínimo se devuelve self
   },
   "_payload": {
      "author": "Enrique Gómez Salas",
      "title": "Desventuras de un informático en Paris",
      "genre": "scifi",
      "price": { "currency":"€", "amount":50 }
   }
}
```

- Toda la información acerca de la respuesta se engloba en el campo específico de la API **_response**. Además seguimos siendo capaces de soportar **suppress_response_codes**.
- Toda la potencia del formato HAL la encapsulamos en el campo específico de la API **_links**. De momento sólo contemplaremos el control self. Pero con el tiempo podemos extenderlo todo lo que queramos.
- Todos los clientes más básicos pueden seguir consumiendo la API como de costumbre, ignorando los campos específicos de la API "_".

- Si fuera un listado, en el _playload irá un array de entidades.

Códigos de estado

Cada respuesta al servicio REST deberá devolver un código de estado que indicará el resultado de la operación, siempre que se pueda, será un código de error contemplado en el protocolo HTTP.
El uso de los códigos de estado de HTTP es vital para mantener la semántica de HTTP.
En la práctica, el resultado de una llamada REST podría clasificarse en los siguientes tipos:

- Todo fué bien.
- La aplicación hizo algo mal, error en cliente.
- La API hizo algo mal, error en servidor.
- Recurso sin cambios, obsoleto o movido (gestión de versiones).

Esto traducido a códigos HTTP sería:

- 2XX Todo fué bien.
- 4XX La aplicación hizo algo mal, error en cliente.
- 5XX La API hizo algo mal, error en servidor.
- 3XX Recurso sin cambios, obsoleto o movido (gestión de versiones).

Los códigos de estado más comunes en APIs REST son los siguientes:

- **200**: OK: Mensaje genérico de éxito. se debe especificar en el cuerpo del mensaje los detalles de la respuesta.
- **201**: Created: respuesta típica a una operación de creación a través de POST (o PUT para creaciónes idempotentes).
- En la respuesta se devuelve la URI del nuevo recurso creado dentro de la cabecera Location.
- **202**: Accepted: Petición aceptada pero aún no se ha terminado de procesarse. Normalmente aparecen en peticiones que requieren tiempo para ejecutarse o cuando el servidor está congestionado y se consulta periódicamente para conocer su progreso. Pricipalmente se usa para modelar operaciones de larga duración.
- **204**: OK: Mensaje genérico de éxito, sin cuerpo de mensaje.
- **301**: Indica que el recurso se ha movido a otra URI de forma permanente. La nueva URI se indica en la cabecera Location de la respuesta. También puede usars para redirigir servicios entre diferentes versiones.
- **304**: Not Modified: Sin datos nuevos que enviar, probablemente esté en caché.
- **400**: Bad Request: Petición inválida. Se adjunta mensaje de error con detalles.
- **401**: Unauthorized: Los datos de autenticación no se enviaron o son incorrectos.
- **403**: Forbidden: Aun estando autenticado, la petición fué rechazada, se intentó acceder a un recurso sin permisos suficientes. Se adjunta mensaje con más detalles.
- **404**: Not Found: El recurso es inválido o no existe.
- **405**: Método o verbo HTTP no soportado o no implementado.
- **406**: Not Acceptable: Normalmente se devuelve este error cuando los parametros pasados , aun siendo correctos de forma individual, no son correctos, principalmente porque se contradicen o no tienen sentido. Se envia mensaje con más detalles si es posible.
- **409**: Conflicto de versiones. Normalmente sucede en modificaciones con concurrencia optimista.
- **412**: Fallo en la precondición.
- **500**: Internal Server Error: Sucedió un error inesperado en el servidor. Este tipo de respuesta tienen que minimizarse.
- **503**: Service Unavailable: Servidor inacesible o en mantenimiento.

La lista de los códigos de estado pueden verse en los siguientes enlaces:

- W3C
- Wikipedia

Autenticación y Seguridad

Información más actualizada sobre el tema (leer primero): Seguridad y Autenticación en una API REST
La autenticación en los servicios REST es un tema que aún se debate y del que aún no existe consenso claro.
Uno de los aspectos importantes de las APIs REST es la autenticación, que dada la naturaleza REST, es importante indicar las credenciales en cada petición para cada recurso que necesite de privilegios.
Los recursos públicos no se ven afectados por los mecanismos de autenticación.
La autenticación debería basarse en el recurso y no en la URL.
Entre las diferentes estrategias de autenticación

Basado en HTTP

Solución fácilmente implementable, basado en HTTP, tiene el problema de que requiere de interacción del usuario, cosa que en REST puede no suceder (puede ser una aplicación la que interactúe con la API, no con un usuario).
Dependiendo del método, las claves pueden ir en claro y por último no existe el concepto de "log-out".

Basado en sesiones (Cookies)

El contexto del cliente (su sesión) se gestiona y almacena en el servidor, a través del uso de cookies y el concepto de sesiones HTTP.
Esta solución es fácil de implementar pero viola una de las propiedades fundamentales de las APIs REST, donde toda petición REST tiene que llevar consigo todo el contexto necesario para realizar la petición independientemente de la conversación realizada entre cliente y servidor.
El contexto del cliente debería pertenecer a éste, no al servidor, que al tener que gestionar las sesiones de usuario, complican su escalabilidad.

Peticiones firmadas

Este método consiste en firmar todas las peticiones añadiendo parámetros a la petición REST.
La firma de peticiones se realiza a través de un conjunto de API keys (pública y privada) que comparten tanto el cliente como el servidor.
El cliente al realizar cualquier petición, firmará la petición realizando un HASH de la parte específica de la URL que define al recurso, junto a la query_string con los parámetros ordenados alfabéticamente y su private_key
Así pues, para la siguiente llamada

```
GET /object?apiKey=Qwerty2010
```

se convierte en esta otra:

```
GET /object?apiKey=Qwerty2010&timestamp=1261496500&signature=abcdef0123456789
```

Lo que se está firmando es "/object?apikey=Qwerty2010&timestamp=1261496500". Nótese que el orden de los parámetros sigue un orden alfabético, no es trivial. Se ha firmado usando un HASH SHA256 junto a la clave privada.
El uso del timestamp es para evitar que una misma llamada firmada pueda ser reutilizada seguidas veces y evitar que un tercero con dicha url pueda acceder a los recursos mientras la api_key privada sea válida.
Existen muchos detractores preocupados por la capacidad de cachear este tipo de URLs y los que la adoptan realizan la **caché a nivel de consulta y no a nivel de petición**.

También es posible montar una fachada que se responsabilice únicamente de la autenticación y luego realice la petición del recurso al subsistema adecuado eliminando la firma y el timestamp. La petición entre cliente y fachada dese ser rápida, con fachadas geolocalizadas, y la comunicación entre fachada y servidor se cachea fácilmente ya que la URL va sin firma ni timestamp. La seguridad entre fachada y servidor se realiza con al menos, un firewall que identifique a las fachadas geolocalizadas.
El cifrado debería realizarse con un hash HMAC-SHA256.
El sistema debe permitir a un mismo usuario tener diferentes api_keys con distinta expiración y la posibilidad de desactivarlas individualmente de forma simple en caso de que cualquiera de ellas se viera comprometida.
Existe debate sobre cuándo firmar las peticiones, si antes o despues de codificarlas. Realmente da igual cualo de ellas escoger, lo importante es ser siempre consistente.

Seguridad

Let's just be blunt: if you aren't encrypting your API calls, you aren't even pretending to be secure - George Reese, enStratus CTO
Se asume SSL en la conexión.
Consideraciones sobre los identificadores:

- Evitar identificadores incrementales, pues permite a un atacante localizar recursos válidos del sistema.
- Evitar el uso de claves primarias en la URL, o al menos protegerlas contra inyeccicones de SQL o construir la clave primaria a partir de la URI mediante una clave secreta. Se trataría de hacer una hash criptográfica de la URI con una clave secreta.

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

Caché

La caché es uno de los elementos claves que hacen a las APIs REST escalables.

ETag

Es un mecanismo para identificar si la información que tiene el cliente sigue siendo válida en el servidor. Esta clave es un hash del recurso que el cliente solicita.
Ejemplo de cabeceras usando ETag:

- Servidor envía la siguiente cabecera tras solicitar un recurso:
  - `ETag: "21314123132134as12dd2"`
- El cliente vuelve a solicitar el mismo recurso más tarde con la siguiente cabecera adjunta:
  - `If-None-Match: "21314123132134as12dd2"`
- Respuesta del servidor:
  - 304 Not Modified

If-Modified-Since / Last-Modified

Es posible utilizar las cabeceras If-Modified-Since y Last-Modified cuando la probabilidadde modificación datos es baja y es aceptable que el usaurio pueda ver contenido antiguo.

Cache-Control y Expires

Los mecanimos antes mencionados no impiden que se tenga que hacer una petición al servidor por muy poco coste computacional que tenga ya que se ve penalizado igualmente por la latencia.
Las cabeceras Cache-Control y Expires son útiles en este caso, pero sólo cuando es asumible que el cliente tenga información obsoleta durante un periodo corto de tiempo. Por ejemplo, un servidor de noticias, donde es posible tener noticias de hace cinco minutos...

Internacionalización

Las cabeceras Accept-Charset y Accept-Language permiten seleccionar el juego de caracteres y el idioma deseado para la respuesta. Un ejemplo:

```
GET/libros/465 HTTP/1.1
Accept-Charset:iso-8859-5,unicode-1-1;q=0.8
Accept-Language:es-es,en-gb;q=0.8,es-ca;q=0.7
```

Es posible establecer varios idiomas con niveles de preferencia.

Fuente: https://code.tscompany.es/redmine/projects/desarrollo/wiki/REST