¿Qué es una API REST?

API REST

El término REST (Representational State Transfer) se originó en el año 2000, descrito en la tesis de Roy Fielding, padre de la especificación HTTP. Un servicio REST no es una arquitectura software, sino un conjunto de restricciones que tener en cuenta en la arquitectura software que usaremos para crear aplicaciones web respetando HTTP.

Según Fielding las restricciones que definen a un sistema RESTful serían:

- **Cliente-servidor**: El servidor se encarga de controlar los datos mientras que el cliente se encarga de manejar las interacciones del usuario. Esta restricción mantiene al cliente y al servidor débilmente acoplados (el cliente no necesita conocer los detalles de implementación del servidor y el servidor se "despreocupa" de cómo son usados los datos que envía al cliente).
- **Sin estado**: aquí decimos que cada petición que recibe el servidor debería ser independiente y contener todo lo necesario para ser procesada.
- **Cacheable**: debe admitir un sistema de almacenamiento en caché. Este almacenamiento evitará repetir varias conexiones entre el servidor y el cliente para recuperar un mismo recurso.
- **Interfaz uniforme**: define una interfaz genérica para administrar cada interacción que se produzca entre el cliente y el servidor de manera uniforme, lo cual simplifica y separa la arquitectura. Esta restricción indica que cada recurso del servicio REST debe tener una única dirección o "URI".
- **Sistema de capas**: el servidor puede disponer de varias capas para su implementación. Esto ayuda a mejorar la escalabilidad, el rendimiento y la seguridad.

Hoy en día la mayoría de las empresas utilizan API REST para crear servicios web. Esto se debe a que es un estándar lógico y eficiente. Por poner algún ejemplo tenemos los sistemas de identificación de Facebook o también la autenticación en los servicios de Google (hojas de cálculo, Google Analytics, Google Maps, …).

Métodos en una API REST

Las operaciones más importantes que nos permitirán manipular los recursos son:

- **GET** es usado para recuperar un recurso.

- **POST** se usa la mayoría de las veces para crear un nuevo recurso. También puede usarse para enviar datos a un recurso que ya existe para su procesamiento. En este segundo caso, no se crearía ningún recurso nuevo.
- **PUT** es útil para crear o editar un recurso. En el cuerpo de la petición irá la representación completa del recurso. En caso de existir, se reemplaza, de lo contrario se crea el nuevo recurso.
- **PATCH** realiza actualizaciones parciales. En el cuerpo de la petición se incluirán los cambios a realizar en el recurso. Puede ser más eficiente en el uso de la red que PUT ya que no envía el recurso completo.
- **DELETE** se usa para eliminar un recurso.

Otras operaciones menos comunes pero también destacables son:

- **HEAD** funciona igual que GET pero no recupera el recurso. Se usa sobre todo para testear si existe el recurso antes de hacer la petición GET para obtenerlo (un ejemplo de su utilidad sería comprobar si existe un fichero o recurso de gran tamaño y saber la respuesta que obtendríamos de la API REST antes de proceder a la descarga del recurso).
- **OPTIONS** permite al cliente conocer las opciones o requerimientos asociados a un recurso antes de iniciar cualquier petición sobre el mismo.

Dos términos a tener en cuenta son los de métodos seguros y métodos idempotentes. Se dice que los métodos seguros son aquellos que no modifican recursos (serían GET, HEAD y OPTIONS), mientras que los métodos idempotentes serían aquellos que se pueden llamar varias veces obteniendo el mismo resultado (GET, PUT, DELETE, HEAD y OPTIONS).

La idempotencia es de mucha importancia ya que permite que la API sea tolerante a fallos. Por ejemplo, en una API REST bien diseñada podremos realizar una operación PUT para editar un recurso. En el caso de que obtuviésemos un fallo de Timeout (se ha superado el tiempo de espera de respuesta a la petición), no sabríamos si el recurso fue creado o actualizado. Como PUT es idempotente, no tenemos que preocuparnos de comprobar su estado ya que, en el caso de que se haya creado el recurso, una nueva petición PUT no crearía otro más, algo que sí habría podido pasar con una operación como POST.

Principios de diseño de REST

En el nivel más básico, una API es un mecanismo que permite que una aplicación o servicio acceda a un recurso dentro de otra aplicación o servicio. La aplicación o servicio que realiza el acceso se llama cliente y la aplicación o servicio que contiene el recurso se llama servidor.

Algunas API, como SOAP o XML-RPC, imponen un marco estricto a los desarrolladores. Pero las API REST se pueden desarrollar utilizando prácticamente cualquier lenguaje de programación y admiten una variedad de formatos de datos. El único requisito es que se alineen con los siguientes seis principios de diseño de REST, también conocidos como restricciones de arquitectura:

1. **Interfaz uniforme**. Todas las solicitudes de API para el mismo recurso deben tener el mismo aspecto, sin importar de dónde provenga la solicitud. La API REST debe garantizar que el mismo dato, como el nombre o la dirección de correo electrónico de un usuario, pertenezca a un solo identificador uniforme de recursos (URI). Los recursos no deben ser demasiado grandes, pero deben contener toda la información que el cliente necesite.
2. **Desacoplamiento cliente-servidor.** En el diseño de API REST , las aplicaciones de cliente y servidor deben ser completamente independientes entre sí. La única información que debe conocer la aplicación del cliente es el URI del recurso solicitado; no puede interactuar con la aplicación del servidor de ninguna otra manera. De manera similar, una aplicación de servidor no debería modificar la aplicación del cliente además de pasarla a los datos solicitados a través de HTTP.
3. **Sin memoria de estados previos.** Las API REST no tienen estado, lo que significa que cada solicitud debe incluir toda la información necesaria para tratarla. En otras palabras, las API REST no requieren ninguna sesión del lado del servidor. Las aplicaciones de servidor no pueden almacenar ningún dato relacionado con la solicitud de un cliente.
4. **Capacidad de caché**. Cuando sea posible, los recursos deben almacenarse en la memoria caché en el lado del cliente o del servidor. Las respuestas del servidor también deben contener información sobre si se permite el almacenamiento en caché para el recurso entregado. El objetivo es mejorar el rendimiento en el lado del cliente, mientras aumenta la escalabilidad en el lado del servidor.
5. **Arquitectura del sistema en capas**. En las API REST, las llamadas y respuestas pasan por diferentes capas. Como regla general, no asuma que las aplicaciones cliente y servidor se conectan directamente entre sí. Puede haber varios intermediarios diferentes en el bucle de comunicación. Las API REST deben diseñarse de modo que ni el cliente ni el servidor puedan saber si se comunican con la aplicación final o con un intermediario.
6. **Código a pedido (opcional)**. Las API REST generalmente envían recursos estáticos, pero en ciertos casos, las respuestas también pueden contener código ejecutable (como los subprogramas de Java). En estos casos, el código solo debe ejecutarse a pedido.

Algunas características de una API REST

- **El uso de hipermedios** (procedimientos para crear contenidos que contengan texto, imagen, vídeo, audio y otros métodos de información) para permitir al usuario navegar por los distintos recursos de una API REST a través de enlaces HTML (principio HATEOAS, Hypermedia as the engine of application state o hipermedia como el motor del estado de la aplicación).
- **Independencia de lenguajes**. La separación en capas de la API permite que el cliente se despreocupe del lenguaje en que esté implementado el servidor. Basta a ambos con saber que las respuestas se recibirán en el lenguaje de intercambio usado (que será XML o JSON).
- **Los recursos en una API REST se identifican por medio de URI**. Será esa misma URI la que permitirá acceder al recurso o realizar cualquier operación de modificación sobre el mismo.
- **Las APIs deben manejar cualquier error que se produzca,** devolviendo la información de error adecuada al cliente. Por ejemplo, en el caso de que se haga una petición GET sobre un recurso inexistente, la API devolvería un código de error HTTP 404.

Algunos frameworks con los que podremos implementar nuestras APIs son: JAX-RS y Spring Boot para Java, Django REST framework para Python, Laravel para PHP, Rails para Ruby o Restify para Node.js.