API REST

INTRODUCCION

DEFINICION

Una **API** *(Application Programming Interface)* es un conjunto de reglas y protocolos que permiten la comunicación entre diferentes aplicaciones y sistemas informáticos. De esta forma, se pueden intercambiar datos y funcionalidades de manera estandarizada y automatizada, sin necesidad de conocer los detalles internos de cada sistema.

REST es un estilo de arquitectura de software que busca crear aplicaciones independientes y distribuidas en la red. Fue creado por *Roy Fielding* en el año 2000 y está destinado principalmente a la Web. **REST** establece un conjunto de reglas y sugerencias para diseñar aplicaciones centradas en recursos, como documentos y archivos, sin entrar en detalles técnicos específicos de implementación.

RESTRICCIONES

Las **restricciones** son un conjunto de principios de diseño que deben seguirse para crear una arquitectura de software que cumpla con los principios de **REST**.

- Usa el paradigma cliente-servidor
- Sin estado
- Cacheable (debe poder almacenarse en una memoria caché)
- Interfaz uniforme
- Sistema de capas
- Código bajo demanda (opcional)

CLIENTE-SERVIDOR

Un cliente, normalmente un dispositivo externo, se conectan a un servidor, el cual ofrece un servicio.

El cliente envía peticiones al servidor, que las procesa y envía una respuesta al cliente.

SIN ESTADO

Esto quiere decir que el servidor no guarda datos del estado del cliente.

Todas las peticiones, tanto del cliente como del servidor, deben ser completas, esto quiere decir que deben tener toda la información necesaria que el cliente o el servidor pueda necesitar para poder procesarlas.

CACHEABLE

Esto quiere decir que en medio del cliente y servidor podemos poner otro servicio que tiene una caché. Esta caché almacenará peticiones recientes junto con el tiempo de validez de dicha petición.

Esto evita que al solicitar de nuevo una misma petición saturemos el servidor.

INTERFAZ UNIFORME

- Debemos usar **URI** para identificar los recursos
- Separar un recurso de su representación
- Debemos usar mensajes descriptivos (método + cabecera + contenido)
- Uso de hipermedia (HATEOAS), que son metadatos adicionales que dan contexto a los datos

SISTEMA DE CAPAS

Consiste en interponer capas entre el cliente y el servidor, de manera que el cliente no se conecte directamente con el servidor, dichas capas además pueden añadir funcionalidad.

- Esto permite encapsular servicios legados, por ejemplo, disponemos de un servicio que no es REST y mediante estas capas lo podemos transformar en un servicio REST
- Introduce intermediarios (que añaden funcionalidad)
- Reducen complejidad, distribuye las tareas entre diferentes pequeños servicios
- Incrementa la escalabilidad

CODIGO BAJO DEMANDA (OPCIONAL)

- Permite enviar código al cliente para que lo ejecute, para poder ejecutarlo sobre los datos que se le han enviado: *Javascript, Applets, Flash*
- El cliente debe soportar la ejecución del código
- Es importante tener en cuenta que esto debe hacerse si la ejecución es más eficiente en el front-end, por ejemplo, la comprobación de los datos de un formulario es más eficiente en el front-end antes de mandarlos

OTROS

- REST se basa en el protocolo **HTTP(S)**
- Los recursos se identifican con **URI**
- Se pueden incluir parámetros opcionales en la URL
- Uso de verbos HTTP: `GET POST PUT DELETE OPTIONS HEAD`
- Podemos recibir/enviar metadatos usando cabeceras HTTP
- Podemos incluir un cuerpo en cualquier formato (soporte mínimo de JSON)
- Para conocer el resultado de las peticiones que enviamos, usamos códigos de respuesta HTTP
- Una REST API no debe limitarse a ser un CRUD, es decir no debe limitarse a exponer los datos
- El cliente no debería saber cómo funciona el servicio
- Debe diseñarse pensando en el cliente

- Solo debe exponerse la funcionalidad necesaria que solicita el cliente, sin ninguna información adicional
- REST no es un estándar, hay cosas que se consideran buenas y malas prácticas

RECURSOS

DEFINICION

Es una referencia explícita a una entidad que puede identificarse y asignarse. Ejemplos:

- Página web
- Un artículo
- Publicaciones en redes sociales
- Ficheros
- Entidades

> *Ejemplo: Usuario, producto, factura...*

REST pone a los recursos en el centro. Con una API REST podemos añadir recursos, modificarlos, borrarlos, etc.

Un recurso y su representación/formato NO so lo mismo, es decir que lo que recibe o envía el cliente no es el recurso, solo es una representación de este.

Un recurso puede servirse en diferentes formatos

> *Ejemplo: XML, JSON, HTML, texto plano...*

Los recursos se identifican con URIs (URLs en web), y solo puede haber una relación 1:1.

URI

```
URI = ESQUEMA :// AUTORIDAD / RUTA [? QUERY][# FRAGMENTO]
```

TERMINO	EQUIVALENCIA EN API REST
Esquema	HTTP
Autoridad	Dominio o IP que tiene alojado el servicio que estamos consultando
Ruta	Ruta que nos lleva hasta el recurso que queremos consultar
Query	Ruta que nos lleva hasta el recurso que queremos consultar
Fragmento	No se utiliza en rest

Conteptos importantes sobre las **URIs** en un servicio REST:

- El separador de las URIs es **barra "/"**
- Las URIs no deben terminar con **barra "/"**

- Se pueden usar **guiones "-"**
- Usar siempre minúsculas
- Todos los elementos de una **ruta** identifican recursos
- **NO** se deben usar **guiones bajos "_"**
- **NO** incluir **extensiones de ficheros** (.json, .xml, .html) dentro de la URI

TIPOS DE RECURSOS (ARQUETIPOS)

- **Documentos:** un documento es algo individual, por ejemplo, un registro de una tabla de una BD. Se considera el arquetipo base, todos los demás tipos de recursos son especializaciones de un documento. Se representa con un nombre en singular o con un identificador

  > *Ejemplo: https://localhost/api/users/**mykyta***
  > *Ejemplo: https://localhost/api/users/**Y3573309R***

- **Colecciones:** es un directorio/lista/conjunto de recursos, que pueden ser de cualquier tipo (documento, colección, etc). La colección es propiedad del servicio (normalmente). Debe tener un nombre en plural (users/items/invoices...)

  > *Ejemplo: https://localhost/api/**users**/mykyta*

- **Almacenes:** la diferencia entre un almacén y una colección es que el propietario de un almacén es el usuario. Se utiliza muy poco. Al igual que las colecciones, tienen un nombre en plural. Ejemplo: una playlist de Spotify podría ser un ejemplo de un almacén, ya que es el usuario es el encargado de añadir las canciones a dicha playlist

  > *Ejemplo: https://localhost/api/users/mykyta/**favorites***

- **Controladores:** son ejecutables/funciones/procedimientos que podemos lanzar y que afectan a los recursos. Podemos enviarles parámetros y que nos devuelvan resultados. Los controladores utlizan **verbos**

  > *Ejemplo: https://localhost/api/users/1/**process***

  > *Importante: **NO** debemos utilizar verbos CRUD: INSERT/DELETE/UPDATE...*

QUERY

Se utiliza para pasar parámetros, los posibles parámetros admisibles en un API REST son:

PARAMETRO	DESCRIPCION
Filtros	Se utiliza para realizar búsquedas o para limitar el número de recursos a devolver
Paginación	Lo mismo que en el caso de los filtros, usado para no sobrecargar la red enviando datos
Ordenación	Su nombre es bastante descriptivo: se utliza para ordenar datos, por ejemplo, por orden alfabético

> *Ejemplo: https://localhost/api/invoices**?limit=40&order-by=date***

METODOS HTTP

Para interactuar con un API REST usamos métodos HTTP. Cuando queremos realizar alguna acción sobre los recursos que tenemos disponibles en un servicio, enviamos una petición con uno de estos métodos HTTP.

Los métodos HTTP son:

- **GET**
- **HEAD**
- **PUT**
- **PATCH**
- **DELETE**
- **OPTIONS**
- **POST**

GET

- Se utiliza para **obtener la representación de un recurso**
- **NO debe tener efectos colaterales**, esto quiere decir que cuando utilizamos GET, no debe producirse ningún cambio de estado en el servicio (ningún servicio debe modificarse, es decir es **READ-ONLY**)
- La solicitud **NO debe contener cuerpo**, ya que el cuerpo se utiliza para enviar datos al servidor y lo que queremos es obtener un recurso
- La respuesta puede contener cabeceras HTTP (metadatos)
- La respuesta suele contener un cuerpo

HEAD

- Es similar al método GET
- Se utiliza para **saber si un recurso existe u obtener sus metadatos**, todo esto sin tener que descargar ninguna representación del recurso
- Al igual que GET, este método es **READ-ONLY**
- Tanto la petición como la respuesta **NO tienen cuerpo**
- La respuesta puede contener cabeceras HTTP

> *Ejemplo: Comprobar la existencia de un fichero de gran tamaño sin necesidad de descargarlo*

PUT

- Se utiliza para **actualizar** un recurso **mutable existente**, esto quiere decir que:
  - El recurso **debe existir**
  - El recurso **debe poder modificarse**

- En el cuerpo de la petición debe incluirse una **representación completa** del recurso, en un formato que entienda el servidor y debe incluir **TODOS** los campos
- El servidor lo que hace es **sustituir TODOS los datos** por los que envía el cliente, los datos que no se incluyan serán eliminados
- Este método produce un cambio de estado en el servicio
- La respuesta puede incluir una representación del recurso actualizado, esto no siempre se hace, para ahorrar ancho de banda
- También se utiliza para insertar/actualizar un recurso en un almacén.

PATCH

- Al igual que PUT, se utiliza para actualizar un recurso existente
- A diferencia de PUT, no es necesario enviar la representación entera del recurso, **SOLO** debemos incluir aquellos campos que queramos modificar
- El cuerpo de la respuesta suele incluir una representación actualizada del recurso

DELETE

- Se utiliza para eliminar un recurso de su padre, por ejemplo, para eliminar un recurso de una colección
- Después de la ejecución de un método DELETE, la respuesta del servidor debe ser **404 - Not Found**
- No debe utilizarse para desactivar u ocultar recursos

OPTIONS

- Se utiliza para saber qué otros métodos podemos utilizar sobre un recurso
- **SIEMPRE** devuelve una cabecera **Allow** con una lista de métodos admitidos

  > *Ejemplo: Allow: GET, PUT, DELETE*

- Puede devolver un cuerpo con más detalle

POST

- Se usa para **crear un nuevo recurso** en una colección
- En el POST **debemos enviar la URI de la colección** donde se va a crear el recurso
- En el cuerpo especificamos la representación completa del recurso que vamos a crear
- Esta representación se considera una sugerencia, ya que la colección es siempre propiedad del servicio
- Normalmente, se devuelve una representación del recurso creado
- Con POST podemos ejecutar controladores
  - Se pueden especificar parámetros, tanto en la cabecera, como en el cuerpo de la petición
  - La respuesta puede incluir información sobre el estado de la ejecución

    - Es posible que se reciba una respuesta antes de que termine la ejecución. Esto ocurrirá cuando el método sea asíncrono
- No es idempotente, esto quiere decir que:
    - Si se ejecuta dos veces seguidas puede dar resultados diferentes, a diferencia de los demás métodos
    - Este método puede ser inseguro, hay que tener cuidado con su uso

RESUMEN

Método	Uso	Equivalente
GET	Obtener la representación de un recurso	SELECT
HEAD	Obtener metadatos de un recurso	---
PUT	Modificar completamente un recurso mutable existente	UPDATE
PUT	Crear un recurso en un almacén	---
PATCH	Modificar parcialmente un recurso existente	UPDATE
DELETE	Eliminar un recurso existente	DELETE
OPTIONS	Obtener los métodos HTTP aplicables a un recurso	---
POST	Crear un recurso en una colección	INSERT
POST	Ejecutar un controlador	EXECUTE

CABECERAS HTTP

REST utiliza el protocola HTTP, por lo que podemos indicarle al servidor lo que queremos mediante cabeceras HTTP, y a su vez las respuestas del servidor también pueden incluir metadatos relacionados con el recurso.

Estas cabeceras pueden ser útiles para los gestores de caché.

CABECERAS MAS UTILIZADAS

CABECERA	OBLIGATORIO	DESCRIPCION
Content-Type	SI	Indica el formato del cuerpo usado en peticiones y respuestas
Location	SI	Indica la dirección de un recurso que puede interesar al cliente. Se usa, por ejemplo, tras realizar un POST, indicándonos la dirección del recurso creado. Se utiliza solo en respuestas
Accept	SI	Inidca el formato en el que queremos recibir la representación del recurso. Solo se usa en la peticiones
Content-Length	NO	Especifica la longitud del cuerpo, usado en peticiones y respuestas. Es muy útil ya que podemos saber si hemos recibido la respuesta completa

CABECERA	OBLIGATORIO	DESCRIPCION
ETag	NO	Es una cadena vinculada con el estado del recurso. Se utiliza solo en las respuestas
Cache-Control	NO	Indica cuánto tiempo puede almacenarse un recurso en caché, sin necesidad de volver a solicitar la información al servidor
Expires	NO	Es parecido a Cache-Control, indica hasta cuándo puede alamacenarse un recurso (fecha + hora), sin embargo, tiene el mismo uso

Podemos incluir cabeceras HTTP propias mediante el prefijo `X-`. Hay que tener en cuenta que las cabeceras personalizadas no deben cambiar el comportamiento de los métodos HTTP.

RESPUESTAS HTTP

Empleando códigos de estado HTTP, el API REST nos puede devolver diferentes respuestas a una petición.

CODIGOS DE ESTADO HTTP

CODIGO	DESCRIPCION
1XX INFORMACIONALES	Están relacionados con el propio protocolo HTTP, no son importantes para el API REST en sí
2XX EXITO	Siempre indican condiciones de éxito, es decir, que la petición se ha podido satisfacer sin problemas
3XX REDIRECCION	Indican que el cliente debe redirigir la petición a otra dirección
4XX ERROR CLIENTE	Se ha producido un error debido al cliente
5XX ERROR SERVIDOR	Indica que se ha producido un error causado por el servidor

RESPUESTAS 2XX

CODIGO	DESCRIPCION
200 OK	Indica que todo ha ido bien, nunca debe usarse para representar un error. Esta respuesta siempre debe incluir un cuerpo
201 Created	Indica que se ha creado un recurso, por ejemplo, al haber realizado una método POST o al haber ejecutado un controlador
202 Accepted	Es una respuesta al haber enviado utilizado un controlador asíncrono
204 No content	Indica que todo ha ido bien pero no contiene un cuerpo. Esta respuesta tendría sentido al haber utilizado un DELETE, ya que al haberse borrado el recurso no tendría sentido mandar ningún cuerpo de vuelta

RESPUESTAS 3XX

CODIGO	DESCRIPCION
301 Permanently moved	Indica que el recurso se ha movido de forma permanente a otro sitio. Indica que se está intentando utilizar la URI antigüa. Se debe especificar la URI nueva
302 Found	NO SE DEBE UTILIZAR
303 See other	Se envía cuando un controlador ha terminado un trabajo y como resultado te envía una URI de un recurso con el resultado
304 Not modified	No devuelve un cuerpo. Se envía cuando el Etag de la petición coincide coincide con la respuesta, indicando que no es necesario volver a enviar la petición de nuevo
307 Temporary redirect	Es parecido al 301, se utiliza como una manera de decir que el servicio no está disponible en este momento y redirigir al cliente a otra instancia de la API

RESPUESTAS 4XX

CODIGO	DESCRIPCION
400 Bad request	Indica que se ha producido un error en los parámetros de la petición
401 Unauthorized	Indica que el recurso solicitado requiere unas credenciales que no se han especificado
403 Forbidden	Indica que el usuario que intenta solicitar la petición no tiene permisos para solicitar dicha petición
404 Not found	No hay ningún recurso que coincida con la URI proporcionada
405 Method not allowed	Indica que se está intentando utilizar un método sobre un recurso que no lo permite, por ejemplo, un DELETE sobre un recurso que no puede ser eliminado. Junto con esta respuesta, deben incluirse una lista de métodos que sí que se pueden utilizar
406 Not aceptable	Indica que se ha solicitado un recurso en determinado formato, pero el servidor no es capaz de proporcionar el recurso en dicho formato
409 Conflict	Indica que el cliente está intentando poner al servicio en un estado inconsiste. Por ejemplo, si un cliente intenta elminar una colección que tiene recursos dentro
415 Unsupported media type	Indica que la petición se está realizando en un formato no soportado por el servidor

> *Importante: Todas las respuestas de error 4xx deben incluir un cuerpo con información sobre el error producido*

RESPUESTAS 5XX

CODIGO	DESCRIPCION
500 Internal server error	Es un error genérico que indica que se ha prducido un error en el servidor o servicio

> *Importante: Todas las respuestas de error 5xx deben incluir un cuerpo con información sobre el error producido*

EJEMPLOS

OBTENER UN RECURSO

PETICION

```http
GET http://nunsys.com/api/provincias/46
Accept: application/json
```

Esta petición utiliza el *método GET*, indicando que queremos obtener el recurso y usamos la cabecera *Accept* para indicar el formato en el que queremos recibir el recurso

RESPUESTA

```http
//Cabecera de la respuesta
200 - OK
Content-Type: applicacion/json
Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT
ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4"
Cache-Control: max-age=604800, must-revalidate

//Cuerpo de la respuesta
{
    "id": 46,
    "nombre": "Valencia"
}
```

La respuesta nos indica en la cabecera que todo ha ido bien, que el formato del cuepor es JSON, la última vez que ha sido modificado, el ETag del contenido y el cache-control, que no indica el tiempo máximo que tiene validez la respuesta y que una vez pasado ese tiempo tendrá que volver a solicitar la información al servidor.

POSTMAN (HERRAMIENTA)

Postaman es un software que actua como cliente de servicios REST para comprobar que todo funciona correctamente.

También permite explorar servicios de terceros.

Postman puede utilizarse sin crear una cuenta, sin embargo, la creación de una cuenta ofrece algunas ventajas, como la conservación de datos y configuraciones entre equipos.

MOCKOON (HERRAMIENTA)

Mockoon es un software que permite emular una API funcional, esto permite realizar peticiones HTTP para realizar pruebas y comprobar que el cliente funciona correctamente.

HTTP REQUEST EN ANGULAR

REALIZAR UNA PETICION HTTP EN ANGULAR

1. Nos dirigimos a `src/app/app.module.ts`
2. Importamos la clase **HttpClientModule**:

```typescript
import {HttpClientModule} from '@angular/common/http';
```

3. Añadimos la clase a nuestras importaciones:

```typescript
imports:[
  HttpClientModule
 ]
```

4. En nuestro *servicio* importamos dicha clase:

```typescript
import { HttpClient } from '@angular/common/http';
```

5. Inyectamos la clase en el constructor:

```typescript
export class ExampleService {

  constructor(private http: HttpClient){}
}
```

6. Creamos un método que se encargará de realizar la petición y nos devolverá un resultado *(en este ejemplo se utiliza un método GET)*:

```typescript
export class ExampleService {

  constructor(private http: HttpClient){}

  public getModel() : Obsevable<ModelClass[]> {
    const urlEndPoint: string = "http://localhost:3000/models";
    return this.http.get<ModelClass[]>(urlEndPoint);
  }
}
```

7. Importamos e inyectamos el servicio creado dentro del componente en el que lo vayamos a utilizar:

```typescript
import {ExampleService} from '../services/example.service';
export class ExampleComponent implements OnInit{

  constructor(private exampleService: ExampleService){}
}
```

8. Creamos los campos donde se guardará los datos obtenidos de la petición:

```typescript
modelClass: ModelClass[] = [];
```

9. Creamos un método que usará el servicio importado:

```typescript
getModelClass(): void {
  this.exampleService.getModel().suscribe({
    next: (request) => {this.modelClass = request}, //si la petición ha ido bien, guardamos los resultados en el campo que hemos creado previamente
    error: (err) => {} //en esta parte introduciremos código para gestionar el error
  })
}
```

EXTRA

Podemos introducir un pipe en nuestra petición, dentro de nuestro servicio:

```typescript
export class ExampleService {

        constructor(private http: HttpClient){}

        public getModel() : Obsevable<ModelClass[]> {
       const urlEndPoint: string = "http://localhost:3000/models";
       return this.http.get<ModelClass[]>(urlEndPoint)
       .pipe(retry(3), catchError(this.handleError())); //en caso de fallo, reintenta la petición 3 veces más. En caso de seguir dando error, lo gestionamos
   }
}
```