RESTful API

Gabriel Rodríguez Flores

November 20, 2025

ADA ITS
INSTITUTO TECNOLÓGICO SUPERIOR

- Teoría y puesta en práctica de una RESTful API
- Trabajo
- RESTful API vs OData vs GraphQL

Contents

1 Teoría

- RESTful API
- Buenas Prácticas RESTful API
- REST en Node.js

1.1 Métodos

- **GET**: Es lo que hace el navegador al entrar en una página
  - Envío
  - Backend
- **POST**: Para envío de datos extensos o ficheros. Encripta los datos.
- **PUT**: Reemplazo total. Si envías solo el nombre, el resto de campos (email, edad) se borran (o se ponen a null).
- **PATCH**: Modificación parcial. Solo actualiza los campos que envías.
- **DELETE**: Eliminar el recurso objetivo.

1.2 Buenas prácticas

- Descripción de los tipos de métodos
- API RESTful buenas prácticas
- Códigos HTTP: Cuáles y cómo usarlos
- Best practices

1.2.1 Tabla de Códigos HTTP Comunes

Código	Significado	Cuándo usarlo
200	OK	Petición exitosa (GET, PUT, PATCH).
201	Created	Recurso creado exitosamente (POST).
204	No Content	Acción exitosa pero sin respuesta (DELETE).
400	Bad Request	El cliente envió datos mal formados o inválidos.
401	Unauthorized	Falta autenticación (no hay token o es inválido).
403	Forbidden	Autenticado, pero sin permisos para este recurso.

Código	Significado	Cuándo usarlo
404	Not Found	El recurso (ID) no existe.
500	Internal Server Error	Error inesperado en el servidor (bug).

1.2.2 Uso de sustantivos

Las rutas han de ser sustantivos, ya que son los métodos (verbos) los que van a decidir la acción a realizar.

- Ej. `/users`

También es interesante el uso de encapsulación, para acceder a un parámetro de un recurso concreto.

- Ej. `/users/:userId/comments`

Donde estaríamos accediendo a los comentarios del usuario con el ID que se inserte.

1.2.3 Manejo y formato de los errores

- Control de errores
- Facebook: Ejemplo de buenas prácticas
  - Objeto y control de errores

El control de los errores ha de ser centralizado, eso significa que toda respuesta ha de ser manejada en un mismo lugar (middleware).

También es conveniente definir un estándar de estructura de error incluyendo los datos que se consideren necesarios. Ha de tener al menos:

```
{
  "code": 404,
  "error": "Not Found",
  "message": "Error: Path not found"
}
```

1.2.4 Filtrado, ordenación y paginado

Filtrado: Se trata de definir uno o varios parámetros en la búsqueda de recursos (BBDD o similar) y limitar la respuesta a los que coincidan con estos criterios.

Ordenación: Se trata de seleccionar un parámetro para que marque el orden de los recursos a devolver.

Se suele indicar con un número positivo (1) si es orden ascendente o negativo (-1) i es orden descendente.

Paginado: Se indica un número de elementos a mostrar, y devolverá esta cantidad junto con otros parámetros, en lugar de devolver la totalidad de los elementos (que pueden ser muchos).

Se suelen ver 2 formas de paginado:

- Offset y limite: Número del elemento por el que se empieza y cuántos se muestran.
- Paginas y cantidad: Número de página actual y elementnos por página.

Otras técnicas y estándares

- HATEOAS proporciona directamente los enlaces para navegación entre páginas
  - Paypal: caso de uso

1.2.5 Seguridad

Es fundamental asegurar que solo los usuarios autenticados y autorizados puedan acceder a ciertos recursos de la API.

- **Autenticación**: Verificar la identidad del usuario. Se puede implementar mediante:
  - **Tokens JWT**: JSON Web Tokens son una forma común de manejar la autenticación en APIs RESTful.
  - **OAuth**: Un protocolo de autorización que permite a las aplicaciones obtener acceso limitado a los recursos de usuario.
- **Autorización**: Determinar si un usuario autenticado tiene permiso para realizar una acción específica.
  - **Roles y permisos**: Definir roles de usuario y permisos asociados para controlar el acceso a los recursos.

1.2.6 Versionado de API

Se incluye en la raiz de la ruta el path `vX`, donde `X` indica el número de la versión de la api que se está utilizando.

- Ejemplo de url con ruta *ping* en la versión 1 de la api: `localhost:3000/api/v1/ping`

1.2.7 Documentación de la API

La documentación es crucial para que los desarrolladores puedan entender y utilizar la API correctamente.

- **Swagger/OpenAPI**: Una herramienta para diseñar, construir, documentar y consumir APIs RESTful.
  - Swagger Editor: Permite crear y visualizar la documentación de la API.
  - Swagger UI: Proporciona una interfaz visual para interactuar con la API.

1.2.8 Pruebas de Integración

Las pruebas de integración aseguran que los diferentes componentes de la API funcionen juntos correctamente.

- **Herramientas de prueba**:
  - **Postman**: Una herramienta para probar APIs que permite crear y ejecutar solicitudes HTTP.
  - **Newman**: Un CLI para ejecutar colecciones de Postman.
  - **Jest**: Un framework de pruebas en JavaScript que puede ser utilizado para pruebas de integración.
- **Prácticas recomendadas**:
  - Crear casos de prueba para todos los endpoints de la API.
  - Incluir pruebas para escenarios de éxito y error.
  - Automatizar las pruebas para ejecutarlas en cada despliegue.

1.3 Alternativas

1.3.1 SOAP

Es anterior a REST y ya no se usa porque es más pesado. Basado en XML

Ventajas: - **Estado de la sesión**: SOAP puede mantener el estado de la sesión entre peticiones, lo que puede ser útil en algunas situaciones. - **Seguridad**: SOAP tiene mecanismos de seguridad built-in como autenticación y cifrado.

Desventajas: - **Complejidad**: SOAP es más complejo que REST, lo que puede hacerlo más difícil de implementar y mantener. - **Overhead**: SOAP introduce un overhead adicional en la comunicación, lo que puede afectar el rendimiento.

1.3.2 GraphQL

Más nuevo que REST, permite consumir los datos a demanda y técnicas de subscripcion

Ventajas: - **Flexible queries**: Los clientes pueden especificar qué datos necesitan en cada petición, lo que reduce el tráfico y la complejidad. - **Single endpoint**: GraphQL solo necesita un endpoint para todas las peticiones, lo que simplifica la implementación y el mantenimiento. - **Strong typing**: GraphQL tiene un sistema de tipos fuerte que permite detectar errores en tiempo de compilación.

Desventajas: - **Complejidad**: GraphQL es más complejo que REST, lo que puede hacerlo más difícil de implementar y mantener. - **Overhead**: GraphQL introduce un overhead adicional en la comunicación, lo que puede afectar el rendimiento. - **Caching**: GraphQL no tiene un sistema de caché built-in, lo que puede requerir implementar un sistema de caché adicional.

1.3.3 OData

OData es un estándar abierto que define una forma común de interactuar con APIs.

Ventajas: - **Tipado**: OData tiene un sistema de tipos fuerte que permite detectar errores en tiempo de compilación. - **Querying**: OData permite a los clientes especificar qué datos necesitan en cada petición, lo que reduce el tráfico y la complejidad. - **Filtros y ordenamiento**: OData tiene un lenguaje de query integrado que permite a los clientes filtrar y ordenar los datos de manera flexible.

Desventajas: - **Complejidad**: OData es más complejo que REST, lo que puede hacerlo más difícil de implementar y mantener. - **Overhead**: OData introduce un overhead adicional en la comunicación, lo que puede afectar el rendimiento. - **Adopción**: Aunque OData es un estándar abierto, su adopción no es tan amplia como la de REST.

1.3.4 gRPC

Basado en HTTP/2 y Protocol Buffers (no JSON). Es muy importante en la industria actual, especialmente en microservicios.

Ventajas: - **Rendimiento extremo**: Mensajes binarios muy pequeños y rápidos. - **Streaming**: Soporte nativo para streaming bidireccional. - **Contratos estrictos**: Se definen con archivos `.proto`.

Desventajas: - **No es legible por humanos**: Formato binario. - **Depuración**: Más difícil de depurar en el navegador que REST.

2 Avanzado

- **Idempotencia**: La propiedad de que una operación produzca el mismo resultado sin importar cuántas veces se ejecute. Más info
- **CORS (Cross-Origin Resource Sharing)**: Los navegadores bloquean por defecto las peticiones de un dominio a otro (ej: de `localhost:3000` a `api.miweb.com`). El servidor debe enviar cabeceras (`Access-Control-Allow-Origin`) para permitirlo.
- **Rate Limiting**: Proteger la API contra ataques de fuerza bruta o saturación limitando el número de peticiones por IP (ej: 100 peticiones cada 15 minutos). Código HTTP asociado: `429 Too Many Requests`.
- **Content Negotiation**: El cliente puede pedir el formato que prefiera usando la cabecera `Accept` (ej: `Accept: application/json` o `Accept: application/xml`).

3 Ejemplos

3.0.1 Ejemplo de Petición y Respuesta (JSON)

Escenario: Crear un nuevo usuario.

Petición (POST /users)

```
POST /api/v1/users HTTP/1.1
Host: api.ejemplo.com
Content-Type: application/json

{
  "name": "Ana García",
  "email": "ana.garcia@email.com",
  "role": "admin"
}
```

Respuesta Exitosa (201 Created)

```
{
  "data": {
    "id": "507f1f77bcf86cd799439011",
    "name": "Ana García",
    "email": "ana.garcia@email.com",
    "role": "admin",
    "createdAt": "2023-10-27T10:00:00Z"
  },
  "links": {
    "self": "http://api.ejemplo.com/api/v1/users/507f1f77bcf86cd799439011"
  }
```

```
}
```

4 Ejercicios

- Realizar un servidor, aplicando el uso de buenas prácticas, que permita:
  - Realizar el CRUD completo de una colección de usuarios bajo la ruta `/users`
    * Listar todos los usuarios que existan
    * Recoger un usuario concreto dado su ID
    * Crear / Actualizar / Sobreescribir / Borrar usuarios
  - Debe permitir el almacenamiento en memoria para mantener los ususarios creados y poder recogerlos
    * Para un uso práctico y no muy pesado, se puede precargar el contenido y tener usuarios iniciados al cargar (integrar el lanzamiento en los loaders).
  - Prestar atención a:
    * Endpoints REST y métodos HTTP
    * Control de errores
    * Uso correcto de códigos HTTP
  - Realización de los test unitarios

4.0.1 Tecnologías sugeridas (Node.js)

- **Framework**: Express.js (el estándar) o Fastify (moderno y rápido).
- **Validación de datos**: Zod o Joi (para validar que el body del POST sea correcto antes de procesarlo).
- **Logging**: Morgan (para ver las peticiones en consola).
- **Seguridad básica**: Helmet (middleware de seguridad para Express).

5 Entregables

5.1 En clase

Dejar tiempo de estudio y hacer un *Kahoot*

- Enlace a Kahoot

5.2 Tarea

Realizar el ejercicio detallado.

5.2.1 Trabajo

- Investigación y comparativa de REST vs OData vs GraphQL
  - Escribir documentación en Markdown (Readme.md)
  - Seleccionar uno entre OData y GraphQL y realizar una puesta en práctica con Nodejs