

fundamentos del Backend con Express.js

1. APIs: El Lenguaje entre Aplicaciones

Una **API (Application Programming Interface)** es un conjunto de reglas y protocolos que permiten que diferentes aplicaciones de software se comuniquen entre sí. La mejor forma de entenderlo es con una analogía simple:

Imagina que estás en un restaurante. Tú (el **cliente**) quieres comida (los **datos**), pero no vas directamente a la cocina (la **base de datos** o el **servidor**). En su lugar, le haces un pedido a un camarero (la **API**). La API se encarga de ir a la cocina, asegurarse de que el plato se prepare y luego te lo trae. Este proceso te permite obtener lo que necesitas sin conocer los complejos procesos internos.

En el desarrollo web, las APIs permiten que tu aplicación de frontend (la interfaz de usuario en React) se comunique con el servidor de backend (tu aplicación en Express.js). Este modelo **cliente-servidor** es fundamental para las aplicaciones modernas.

2. APIs RESTful: El Estándar de la Web

REST (Representational State Transfer) es un estilo arquitectónico para diseñar APIs. Las APIs que siguen estos principios se llaman **RESTful**.

- **Recursos:** Todo en una API REST es un **recurso**, una entidad identificable (ej. un usuario, un producto).
 - **Endpoints:** Los recursos se acceden a través de URLs únicas, llamadas **endpoints** (ej. /api/usuarios).
 - **Stateless (Sin Estado):** Una API REST es **sin estado**. Cada solicitud del cliente debe contener toda la información necesaria para que el servidor la procese. El servidor no "recuerda" nada de las solicitudes anteriores del mismo cliente, lo que la hace robusta y escalable.
 - **Formato de Datos:** El formato de datos más utilizado para el intercambio de información es **JSON (JavaScript Object Notation)**, un formato ligero y legible por humanos.
-

3. Métodos HTTP y Operaciones CRUD

Los métodos HTTP son las "acciones" que el cliente pide al servidor que realice sobre un recurso. Estos métodos se correlacionan directamente con las operaciones **CRUD** (**C**reate, **R**ead, **U**pdate, **D**elete).

Método HTTP	Propósito	Operación CRUD	Uso Común
GET	Obtener datos	Read (Leer)	<code>router.get('/usuarios')</code>
POST	Enviar datos para crear un nuevo recurso	Create (Crear)	<code>router.post('/usuarios')</code>
PUT	Reemplazar completamente un recurso	Update (Actualizar)	<code>router.put('/usuarios/:id')</code>
PATCH	Modificar parcialmente un recurso	Update (Actualizar)	<code>router.patch('/usuarios/:id')</code>
DELETE	Eliminar un recurso	Delete (Eliminar)	<code>router.delete('/usuarios/:id')</code>

La diferencia entre **PUT** y **PATCH** es sutil pero importante: **PUT** sobrescribe todo el recurso, mientras que **PATCH** solo modifica los campos específicos que envías.

4. Códigos de Estado HTTP: El Feedback del Servidor

Los códigos de estado HTTP son la forma en que el servidor se comunica con el cliente para informarle sobre el resultado de una solicitud.

2xx - Respuestas satisfactorias

- **200 OK:** La solicitud ha tenido éxito. Es el código más común para una respuesta exitosa.

- **201 Created:** La solicitud ha sido completada y ha resultado en la creación de un nuevo recurso. Se usa comúnmente en `POST`.
- **204 No Content:** La solicitud ha sido completada con éxito, pero **la respuesta no incluye un cuerpo de mensaje**. Este código es útil para solicitudes `PUT` o `DELETE` donde solo se necesita confirmar la acción.
- **205 Reset Content:** La solicitud fue completada y pide al cliente restablecer la "vista" del documento. No debe tener cuerpo.

3xx - Redirecciones

- **301 Moved Permanently:** El recurso se ha movido permanentemente a una nueva URL.
- **304 Not Modified:** El recurso no ha sido modificado desde la última vez que el cliente lo solicitó. Es una respuesta de caché que **no tiene un cuerpo**.

4xx - Errores del cliente

- **400 Bad Request:** El servidor no pudo entender la solicitud debido a una sintaxis incorrecta.
- **401 Unauthorized:** La solicitud requiere autenticación. El cliente debe proporcionar credenciales.
- **403 Forbidden:** El servidor ha entendido la solicitud, pero se niega a autorizarla.
- **404 Not Found:** El servidor no pudo encontrar el recurso solicitado.
- **409 Conflict:** La solicitud no pudo ser completada debido a un conflicto (ej. intentar crear un recurso que ya existe).
- **415 Unsupported Media Type:** El servidor no acepta el formato de datos enviado.

5xx - Errores del servidor

- **500 Internal Server Error:** El servidor ha encontrado una condición inesperada.
- **503 Service Unavailable:** El servidor no está listo para manejar la solicitud, a menudo por sobrecarga o mantenimiento.

5. Middleware en Express.js: El Intermediario

Un **middleware** es una función que tiene acceso a los objetos de petición (`req`), respuesta (`res`) y a la siguiente función middleware en el ciclo de solicitud-respuesta de una aplicación Express.js.

Piensa en el middleware como una serie de estaciones en una cadena de producción. Cada "estación" recibe un paquete (la solicitud), hace su trabajo (ej. verifica un token, registra la hora de llegada, adjunta datos) y luego pasa el paquete a la siguiente estación.

Funciones de un middleware:

- Ejecutar cualquier código.
- Modificar los objetos `req` y `res`.
- Finalizar el ciclo de solicitud-respuesta.
- Llamar a la siguiente función en la pila de middleware, utilizando la función `next()`.

Ejemplo de uso (Autenticación):

Puedes usar un middleware para verificar si un usuario tiene un token de autenticación válido antes de permitirle acceder a una ruta protegida. Si el token es válido, el middleware llama a `next()`. Si no lo es, envía una respuesta de error.

6. Arquitectura Backend: Distinguendo las piezas clave ✨

Para tus alumnos, es vital diferenciar los roles de cada componente en una aplicación de backend.

- **API vs. Base de Datos (DB):** La **Base de Datos** es el sistema de almacenamiento persistente. La **API** es la interfaz de comunicación que permite a la aplicación interactuar con esa base de datos.
- **API vs. Servidor:** El **Servidor** es la máquina o el software (Express.js) que se está ejecutando. La **API** es el conjunto de reglas y endpoints que define cómo interactuar con ese servidor.
- **Ciclo de Petición-Respuesta:** Un **cliente** envía una petición a un **servidor** Express.js a través de una **ruta**. El servidor usa un **controlador** para procesar la petición. El controlador puede invocar un **servicio** para ejecutar la lógica de negocio, que a su vez interactúa con la **base de datos**. Finalmente, el servidor envía una **respuesta** al cliente.

7. Herramientas Clave: MockAPI, .env y Routing

- **MockAPI:** Una herramienta en línea para crear APIs simuladas de forma rápida. Ideal para probar el frontend sin tener el backend listo.
- **API Key y Variables de Entorno (.env):** Las **API Keys** son credenciales para autenticar peticiones. Es una **mejor práctica** guardar estas claves sensibles en **variables de entorno** dentro de un archivo `.env`, que debe ser excluido del control de versiones (`.gitignore`) para evitar que se suban a repositorios públicos como GitHub.
- **Routing con react-router-dom:** En el frontend (ej. React), librerías como `react-router-dom` manejan la navegación entre URLs sin recargar la página completa. Conceptos como `createBrowserRouter`, `RouterProvider` y `<Outlet />` son fundamentales para crear una navegación fluida en una Single Page Application (SPA).