1 Rutas en Express / Guion de explicaciones y código

```
// Dependencias
const express = require("express");
const app = express();


// Settings
// Seteamos el puerto
app.set("PORT", 5000);


// Middlewares
/*  Este middleware es una funcion que se ejecuta entre la (peticion) request y la (respuestsa) response
    Nos permiten modificar, transformar o procesar los datos de la solicitud y la repuesta

    Analiza el cuerpo (body) de solicitud y convierte el JSON que le mandamos en un objeto JavaScript
    Este objeto JavaScript es accesible a traves del req.body
    Con este middleware, la informacion de la solicitud que mandamos con Postman se convierte siempre en
*/
app.use(express.json());


////////////////////
// RUTAS PUBLICAS //

// RUTA PRINCIPAL
app.get("/", (req, res) => {

    res.status(200).send("Hola esta es la ruta principal");
});

// RUTA ABOUT
app.get("/about", (req, res) => {
    res
    .status(200)
    .send("Sobre nosotros");
});

// RUTA CONTACTO
app.get("/contact", (req, res) => {
    res
    .status(200)
    .send("Contacto");
});



////////////////////
// RUTAS USUARIOS //

// CRUD -> Create, Read, Update, Delete (Crear, Leer, Actualizar y Borrar) -> POST, GET, PUT, DELETE

// Simulamos una BBDD
const users = [];

// Aca definimos la  API
// Rutas disponibles para desarrolladores, devuelve JSON

// GET
// Traemos todos los usuarios de nuestro array
```

```
app.get("/api/users", (req, res) => {
    res
    .status(200)
    .send(users);
});


// Traemos usuarios por id
app.get("/api/users/:id", (req, res) => {

    // Guardamos en la variable id el dato que le pasamos por parametro
    const { id } = req.params;

    // "1" === 1 devuelve false -> Con el operador + convertimos nuestro string id en un number
    const usuarioEncontrado = users.find((usuario) => usuario.id === +id);

    // Si se encuentra el usuario (recordemos los truthy y los falsy en javascript)
    if(usuarioEncontrado) {
        console.log(usuarioEncontrado);
        res.status(200);
        res.send(usuarioEncontrado);

    } else { // Si no se encuentra el usuario

        res.status(404).send("Usuario no encontrado")

    }
})


// POST
// Tambien pueden usar la libreria FakerJS para crear datos y usuarios random
app.post("/api/users", (req, res) => {
    res.status(201); // Nuevo recurso creado
    console.log(req.body);

    // Guardamos en variables usando destructuring, la informacion que viene del body de nuestra peticion
    const { nombre, apellido } = req.body;


    // Hacemos una validacion en caso de no recibir nombre o apellido
    if(!nombre || !apellido) {
        res.status(400).send("Todos los datos son requeridos")

    } else {

        // Creamos el nuevo objeto usuario que recibe los parametros que le mandamos del body
        const newUser = {
            id: users.length + 1,
            nombre: nombre,
            apellido: apellido
        };

        // Enviamos nuestro objeto usuario a nuestro array de objetos
        users.push(newUser)

        res.send(`Usuario ${nombre} ${apellido} creado`);
    }

});


// PUT
// Actualizar usuarios
app.put("/api/users/:id", (req, res) => {
```

```
    // Guardamos el id del usuario en una variable id
    const { id } = req.params;
    const { nombre, apellido } = req.body;

    // Guardamos en una variable el resultado de nuestra busqueda de usuario por su id (que agarramos del
    const usuarioEncontrado = users.find((usuario) => usuario.id === +id);

    if(usuarioEncontrado) {
        console.log(`Usuario encontrado`);

        // Actualizamos los datos de nuestro objeto usuario y rellenamos los datos con lo recibido del re
        // Si no actualizamos una propiedad, queda como estaba
        usuarioEncontrado.nombre = nombre || usuarioEncontrado.nombre;
        usuarioEncontrado.apellido = apellido || usuarioEncontrado.apellido;

        res
        .status(204)
        .send(`Usuario con el id ${id} actualizado`);

    } else {
        res.status(404).send(`Usuario no encontrado`);
    }

});


// DELETE
// Eliminar usuario por id
app.delete("/api/users/:id", (req, res) => {

    const { id } = req.params;

    // Buscamos al usuario por su indice
    const indiceUsuarioEncontrado = users.findIndex((usuario) => usuario.id === +id);

    // Con findIndex o nos devuelve el indice de ese elemento en el array, o nos devuelve -1 (no existe)
    console.log(indiceUsuarioEncontrado);

    // Si nos devolvio su indice, existe
    if(indiceUsuarioEncontrado != -1) {
        users.splice(indiceUsuarioEncontrado, 1);
        res.status(204).send("Usuario eliminado");

    } else {
        // Si el usuario no existe, porque indiceUsuarioEncontrado nos devolvio -1
        res.status(404).send("Usuario no encontrado");
    }
});


// Corremos el servidor a traves del listener
app.listen(app.get("PORT"), () => {
    console.log(`Servidor corriendo en el puerto ${app.get("PORT")}`);
});
```

Codigos de estado HTTP

1xx: Informativos

No son comunes en API REST. Indican que el servidor recibió la solicitud y que el cliente debe esperar más info para completar el proceso

- **100**: `Continue`. El cliente debe continuar con la solicitud
- **101**: `Switching Protocols`. El servidor acepta cambiar a un protocolo diferente

---

2xx: Éxito

Indican que la solicitud fue recibida, entenda y procesada correctamente

- **200**: `OK`. La solicitud se procesó con éxito y el servidor envió una respuesta
- **201**: `Created`. Un nuevo recurso se creó con éxito. Se utiliza en operaciones `POST`.
- **202**: `Accepted`. La solicitud fue aceptada pero aún no se procesó completamente.
- **204**: `No Content`. La solicitud fue exitosa, pero no se envía contenido en la respuesta.

3xx: Redirecciones

No son comunes en API Rest. Indican que el cliente necesita realizar acciones adicionales para completar la solicitud.

- **301**: `Moved Permanently`. El recurso se movió permanentemente a otra URL
- **302**: `Found`. El recurso se encuentra temporalmente en otra URL
- **303**: `Not Modified`. No hay cambios en el recurso desde la ultima solicitud del cliente

---

4xx: Errores del Cliente

Estos codigos incican que la solicitud enviada por el cliente tiene algún problema

- **400**: `Bad Request`. Solicitud invalida por error en los datos enviados
- **401**: `Unauthorized`. El cliente no tiene credenciales validas para acceder al rescurso
- **403**: `Forbidden`. El cliente no tiene permisos para acceder al recurso.
- **404**: `Not Found`. El recurso solicitado no existe.

5xx: Errores del Servidor

Estos codigos incican que hubo un problema interno en el servidor al procesar la solicitud

- **500**: `Internal Server Error`. Ocurrió un error genérico en el servidor
- **502**: `Bad Gateway`. El servidor recibió una respuesta inválida de otro servidor
- **503**: `Service Unavailable`. El servidor no está disponible temporalmente
- **504**: `Gateway Timeout`. El servidor no recibió respuesta a tiempo desde otro servidor

---

Introduccion a protocolo HTTP

El protocolo HTTP (Hypertext Transfer Protocol) es un **conjunto de normas y reglas** que se aplican para guiar una acción o un **proceso de comunicación entre diversos equipos o sistemas en la World Wide Web**. **Es el protocolo de**

transmisión de información de la web, es decir, el código que se establece para que el computador solicitante y el que contiene la información solicitada puedan "hablar" un mismo idioma a la hora de transmitir información por la red.

Funcionamiento del protocolo HTTP

El funcionamiento del protocolo HTTP se basa en un esquema de petición-respuesta entre el servidor web y el "agente usuario" (del inglés user agent) o cliente que realiza la solicitud de transmisión de datos. El protocolo HTTP es de estructura cliente-servidor, esto quiere decir que una petición de datos es iniciada por el elemento que recibirá los datos (el cliente), normalmente un navegador web.

Encabezados y comandos HTTP

Los encabezados HTTP se dividen en dos grupos, los headers de petición y los headers de respuesta. Los headers de petición suelen estar definidos por el método de petición, hacia donde se hará la petición y que información enviara la petición. Los headers de respuesta suelen estar definidos por la respuesta principalmente del servidor, también por las características de la respuesta, como su tipo de respuesta.

Los comandos HTTP más comunes son:

- GET: solicita el recurso ubicado en la URL especificada
- HEAD: solicita el encabezado del recurso ubicado en la URL especificada
- POST: envía datos al programa ubicado en la URL especificada
- PUT: envía datos a la URL especificada
- DELETE: borra el recurso ubicado en la URL especificada

Códigos de respuesta HTTP

Los códigos de respuesta HTTP se utilizan para indicar el resultado de una solicitud. Los códigos de respuesta más comunes son:

- 200: OK, la solicitud se llevó a cabo de manera correcta
- 404: NOT FOUND, el servidor no halló nada en la dirección especificada
- 500: INTERNAL ERROR, el servidor encontró una condición inesperada que le impide seguir con la solicitud

En resumen, el protocolo HTTP es un conjunto de normas y reglas que se aplican para guiar una acción o un proceso de comunicación entre diversos equipos o sistemas en la World Wide Web. Su funcionamiento se basa en un esquema de petición-respuesta entre el servidor web y el cliente que realiza la solicitud de transmisión de datos.

Estructura de una solicitud HTTP

Estructura de una Petición HTTP

Según los resultados de búsqueda proporcionados, la estructura de una petición HTTP se compone de tres partes:

1. **Línea de petición**:
    - Método HTTP (GET, POST, PUT, DELETE, etc.)
    - URI (Uniform Resource Identifier) de la petición
    - Versión del protocolo HTTP (por ejemplo, HTTP/1.1) Ejemplo: `GET /index.html HTTP/1.1`
2. **Cabecera**:
    - Una o varias cabeceras, separadas por un salto de línea (`\r\n`)
    - Cada cabecera tiene un nombre (por ejemplo, `Host` , `Content-Type` , `Cache-Control`) seguido de dos puntos

(`:`) y el valor de la cabecera Ejemplo:

```
Host: en.wikipedia.org
Content-Type: application/json
Cache-Control: no-cache
```

3. **Cuerpo** (opcional):
    - Contenido adicional que se envía con la petición, como datos en formato JSON o HTML Ejemplo (JSON):

```
{
    "Nombre": "Felipe Gavilán",
    "Edad": 999
}
```

En resumen, la estructura de una petición HTTP está compuesta por:

- Línea de petición (método, URI y versión del protocolo)
- Cabecera (una o varias cabeceras con nombre y valor)
- Cuerpo (opcional, contiene datos adicionales)

Es importante destacar que la estructura de la petición puede variar dependiendo del método HTTP utilizado y del tipo de datos que se están enviando.

API / Application Programming Interface

¿Qué es una API?

Una **API (Application Programming Interface)** es un conjunto de definiciones, protocolos y herramientas que permite la comunicación entre diferentes sistemas de software. Es una interfaz que especifica cómo interactuar con un sistema o una aplicación, proporcionando métodos y reglas que facilitan el intercambio de datos y funcionalidades sin necesidad de conocer su implementación interna.

---

1. Características Clave de una API

- **Interoperabilidad:** Facilita la comunicación entre diferentes aplicaciones o sistemas, independientemente del lenguaje de programación o la plataforma.
- **Encapsulación:** Oculta la complejidad del sistema interno, exponiendo solo lo necesario para la interacción.
- **Reusabilidad:** Las APIs permiten reutilizar funcionalidades existentes en nuevos desarrollos.
- **Estándares:** Las APIs suelen seguir estándares como REST, SOAP o GraphQL, lo que las hace predecibles y consistentes.

---

2. Tipos de APIs

Las APIs pueden clasificarse según su uso, accesibilidad y el estilo arquitectónico utilizado.

2.1. Según Accesibilidad

1. **APIs Públicas (Open APIs):**

- Están disponibles públicamente para cualquier desarrollador.
- Generalmente requieren autenticación mediante claves API.
- Se utilizan para ampliar el alcance de un servicio.
- **Ejemplo:** API de Google Maps, API de Twitter.

2. **APIs Privadas:**
   - Solo son accesibles dentro de una organización.
   - Diseñadas para integrar sistemas internos.
   - Mejoran la eficiencia operativa y la seguridad al restringir el acceso.
   - **Ejemplo:** API para gestionar los datos internos de empleados en una empresa.
3. **APIs de Socios (Partner APIs):**
   - Limitan el acceso a socios comerciales específicos.
   - Están diseñadas para fomentar colaboraciones controladas entre organizaciones.
   - **Ejemplo:** API de un proveedor de pagos para integrarla con socios comerciales.
4. **APIs Compuestas (Composite APIs):**
   - Combinan múltiples llamadas de API en una sola solicitud.
   - Útiles para operaciones que involucran múltiples recursos o datos.
   - **Ejemplo:** Una API que devuelva el historial de pedidos de un usuario junto con sus detalles de perfil.

2.2. Según el Estilo Arquitectónico

1. **APIs REST (Representational State Transfer):**
   - Arquitectura basada en recursos accesibles a través de URLs.
   - Usa métodos HTTP (GET, POST, PUT, DELETE) para interactuar con recursos.
   - Es ligera, rápida y ampliamente utilizada.
   - **Ejemplo:** API de GitHub.

   Ejemplo de interacción REST:

   ```
   GET https://api.example.com/users/123
   ```

   Respuesta:

   ```
   {
       "id": 123,
       "name": "John Doe",
       "email": "john.doe@example.com"
   }
   ```

2. **APIs SOAP (Simple Object Access Protocol):**
   - Protocolo basado en XML para intercambio de información.
   - Ofrece seguridad avanzada y transacciones robustas.
   - Más pesado en comparación con REST.

- **Ejemplo:** APIs en sistemas bancarios o gubernamentales.

Ejemplo de solicitud SOAP:

```
<soap:Envelope>
    <soap:Body>
        <GetUserDetails>
            <UserId>123</UserId>
        </GetUserDetails>
    </soap:Body>
</soap:Envelope>
```

3. **APIs GraphQL:**
   - Permite que los clientes soliciten exactamente los datos que necesitan.
   - Usa un único endpoint para todas las consultas.
   - Ideal para aplicaciones con datos complejos.
   - **Ejemplo:** API de Facebook.

   Ejemplo de consulta GraphQL:

```
query {
    user(id: "123") {
        name
        email
    }
}
```

4. **APIs RPC (Remote Procedure Call):**
   - Invocan funciones en un servidor remoto como si fueran locales.
   - Hay dos variantes: JSON-RPC y XML-RPC.
   - Más simples, pero menos flexibles.
   - **Ejemplo:** Sistemas de control remoto.

2.3. Según Funcionalidad

1. **APIs de Datos:**
   - Proporcionan acceso a datos almacenados en una base de datos o sistema.
   - **Ejemplo:** APIs meteorológicas que devuelven pronósticos del tiempo.
2. **APIs de Servicios:**
   - Permiten interactuar con servicios externos como pagos, mensajería, etc.
   - **Ejemplo:** API de PayPal para procesar pagos.
3. **APIs de Hardware:**
   - Interactúan con dispositivos físicos como cámaras, sensores o impresoras.
   - **Ejemplo:** API de la cámara en dispositivos móviles.
4. **APIs de Sistemas Operativos:**

- Proveen acceso a funciones del sistema operativo.
- **Ejemplo:** API de Windows para manipular el sistema de archivos.

5. **APIs de Bibliotecas/Frameworks:**
   - Permiten utilizar funcionalidades predefinidas en bibliotecas o frameworks.
   - **Ejemplo:** API de jQuery o API de React.

---

3. Cómo Funciona una API

1. **Solicitud:** Un cliente realiza una solicitud (por ejemplo, a través de HTTP) especificando el recurso o servicio requerido.
2. **Procesamiento:** El servidor que implementa la API procesa la solicitud, ejecutando las operaciones necesarias.
3. **Respuesta:** El servidor envía una respuesta, generalmente en formato JSON o XML, con los datos solicitados o un estado del resultado.

---

4. Componentes Principales de una API

1. **Endpoint:** Es la URL que expone la funcionalidad o recurso.
2. **Métodos:** Son las operaciones disponibles para interactuar con la API (GET, POST, PUT, DELETE).
3. **Formato de Datos:** JSON, XML o cualquier formato estándar para el intercambio de datos.
4. **Autenticación:** Métodos para garantizar que solo los usuarios autorizados accedan a la API (OAuth, tokens API).
5. **Documentación:** Proporciona detalles sobre cómo usar la API.

---

5. Ejemplo de Uso Práctico de una API

API RESTful: Clima

1. **URL del Endpoint:**

```
GET https://api.openweathermap.org/data/2.5/weather?q=London&appid=tu_api_key
```

2. **Respuesta (JSON):**

```
{
    "weather": [
        {
            "description": "clear sky"
        }
    ],
    "main": {
        "temp": 285.32
    },
    "name": "London"
}
```

---

6. Ventajas de las APIs

- **Facilitan la integración:** Permiten que diferentes sistemas trabajen juntos sin esfuerzo.
- **Aceleran el desarrollo:** Ofrecen funcionalidades preexistentes para evitar desarrollarlas desde cero.

- **Escalabilidad:** Permiten dividir sistemas grandes en componentes independientes.
- **Innovación:** Abren nuevas posibilidades para aplicaciones de terceros.

---

7. Conclusión

Las **APIs** son fundamentales en el desarrollo de software moderno, permitiendo la integración, reusabilidad y escalabilidad de sistemas. Existen múltiples tipos de APIs, cada uno diseñado para cumplir propósitos específicos, desde compartir datos hasta interactuar con hardware. Su correcta implementación y uso puede ser clave para el éxito de una aplicación o sistema.

Relación entre un servidor y una API

La diferencia principal entre un **servidor** y una **API** radica en sus roles y funcionalidades dentro de una arquitectura de software. Mientras que un servidor es la infraestructura física o virtual que alberga y ejecuta servicios, una API es una interfaz que define cómo interactuar con esos servicios. A continuación, se detalla la relación y diferencias entre ambos conceptos:

---

1. Qué es un Servidor

Un servidor es una máquina (física o virtual) o programa que ofrece servicios a otras máquinas o programas (clientes) a través de una red. Estos servicios pueden incluir almacenamiento de datos, procesamiento de solicitudes o entrega de contenido.

Características principales de un servidor:

- **Físico o virtual:** Puede ser un hardware dedicado o un sistema en la nube.
- **Función general:** Maneja solicitudes de los clientes y las procesa según su propósito.
- **Versatilidad:** Puede albergar varios servicios, como servidores web, de correo, de bases de datos, etc.
- **Ejemplo común:** Un servidor web como **Apache** o **NGINX**, que entrega contenido HTML al navegador.

Ejemplo de un servidor en acción:

1. Un usuario accede a `www.example.com`.
2. El servidor recibe la solicitud, busca el archivo correspondiente y lo envía al navegador del usuario.

---

2. Qué es una API

Una **API** es una interfaz que define cómo los clientes (aplicaciones, sistemas o dispositivos) pueden interactuar con un servicio que normalmente se ejecuta en un servidor. Es el "puente" que permite acceder a los servicios de un servidor sin conocer sus detalles internos.

Características principales de una API:

- **Interfaz de comunicación:** Define un conjunto de reglas para enviar y recibir datos.
- **No es hardware:** Es puramente un conjunto de métodos, rutas y reglas.
- **Estándares específicos:** Como REST, SOAP o GraphQL.
- **Usada para integrar:** Facilita la interacción entre diferentes aplicaciones o sistemas.

Ejemplo de una API en acción:

1. Una aplicación móvil quiere mostrar el clima.
2. Envía una solicitud a una API de clima (como la de OpenWeatherMap).
3. La API responde con datos estructurados (generalmente en JSON) sobre la temperatura y condiciones actuales.

Diferencias entre un Servidor y una API

Aspecto	Servidor	API
Definición	Máquina física o virtual que ofrece servicios a través de una red.	Interfaz que define cómo interactuar con un servicio o aplicación alojado en un servidor.
Función principal	Ejecutar y albergar aplicaciones, bases de datos, archivos o servicios.	Proveer acceso a funcionalidades o datos de una aplicación o servicio de forma estructurada.
Tipo de entidad	Puede ser hardware (servidor físico) o software (servidor virtual o servicio).	Es un conjunto de rutas, métodos y reglas para acceder a funcionalidades o datos de un sistema.
Interacción	Responde solicitudes de clientes (navegadores, dispositivos, etc.) y entrega recursos (HTML, datos, etc.).	Responde solicitudes específicas sobre datos o funcionalidades, generalmente en un formato como JSON o XML.
Ejemplo	Un servidor web que entrega un archivo HTML a un navegador.	Una API que entrega información del clima en formato JSON cuando se llama desde una aplicación móvil.
Relación	Alberga y ejecuta la lógica que permite que la API funcione.	Define cómo interactuar con los servicios ofrecidos por el servidor.

---

3. Relación entre un Servidor y una API

1. **Complementarios:** Un servidor a menudo es el "hogar" donde reside la lógica que define las funcionalidades expuestas por una API.
2. **Ejemplo:**
   - **Servidor:** Un servidor ejecuta un backend hecho en **Node.js** con Express.
   - **API:** Ese backend expone rutas RESTful como `/users` o `/products` que los clientes pueden consumir.

---

4. Caso Práctico

Imagina que tienes una aplicación móvil para pedir comida en línea.

Servidor:

- Es el sistema central que:
  - Procesa pedidos.
  - Interactúa con bases de datos para obtener información del menú.
  - Administra usuarios y pagos.

API:

- Es la interfaz que:

- Define rutas como `/menu` (para obtener el menú) y `/order` (para hacer pedidos).
- Se asegura de que la aplicación móvil pueda interactuar con el servidor sin conocer detalles internos.

Conclusión

En resumen, un **servidor** es el lugar donde se ejecutan las aplicaciones o servicios, mientras que una **API** es el conjunto de reglas que permite que otros sistemas accedan a esos servicios de forma controlada y estructurada. Ambos son fundamentales en la arquitectura moderna de aplicaciones, pero cumplen roles distintos.

Explicacion 2 / APIs y APIs REST

¿Qué es una API en programación?

Una **API** (Application Programming Interface, o Interfaz de Programación de Aplicaciones) es un conjunto de reglas y herramientas que permite a diferentes sistemas de software comunicarse entre sí. En términos más simples, es un intermediario que permite que dos aplicaciones hablen entre sí.

Por ejemplo, cuando usas una aplicación en tu teléfono móvil para consultar el clima, la aplicación realiza una solicitud a un servidor y obtiene los datos meteorológicos en tiempo real a través de una API.

Componentes clave de una API

1. **Interfaz**: Define cómo las aplicaciones interactúan con el sistema. Esto incluye las funciones o endpoints disponibles, los parámetros requeridos y los formatos de las respuestas.
2. **Protocolo**: Define las reglas para la comunicación. Por ejemplo, las API web suelen usar el protocolo HTTP.
3. **Entrada y salida**:
   - **Entrada**: Una solicitud que el cliente envía al servidor con parámetros específicos.
   - **Salida**: Una respuesta del servidor al cliente con los datos solicitados.

Ejemplo de una API simple

Supongamos que una tienda tiene una API para consultar productos. Podríamos usar un endpoint como:

- **URL de la API**: `https://api.tienda.com/productos`
- **Solicitud**: `GET /productos`
- **Respuesta**:

```
[
    { "id": 1, "nombre": "Laptop", "precio": 800 },
    { "id": 2, "nombre": "Mouse", "precio": 20 }
]
```

En este ejemplo:

- El cliente solicita todos los productos disponibles en la tienda.
- El servidor devuelve una lista de productos en formato JSON.

¿Qué es una API REST en programación web?

Una **API REST (Representational State Transfer)** es un estilo arquitectónico para diseñar APIs que permiten la comunicación entre un cliente (como un navegador o aplicación móvil) y un servidor. REST se basa en principios sencillos que aprovechan el protocolo HTTP, lo que lo convierte en una de las formas más populares de diseñar APIs en la programación web.

Principios de una API REST

1. **Cliente-servidor**: El cliente (por ejemplo, el navegador) y el servidor (donde se alojan los datos) están separados. Esto mejora la escalabilidad y permite que ambos evolucionen de manera independiente.
2. **Sin estado**: Cada solicitud del cliente al servidor debe contener toda la información necesaria para procesarla. El servidor no guarda información sobre el estado de la sesión del cliente entre solicitudes.
3. **Caché**: Las respuestas del servidor deben ser cacheables para mejorar el rendimiento y reducir la carga en el servidor.
4. **Uniformidad**:
    - Usar rutas coherentes para acceder a recursos.
    - Por ejemplo:
        - `GET /usuarios` para obtener todos los usuarios.
        - `GET /usuarios/1` para obtener un usuario con ID 1.
5. **Representación de recursos**: Los datos se representan en formatos legibles como JSON o XML. REST no se limita a un formato específico, pero JSON es el más común.
6. **Uso de métodos HTTP**: REST utiliza los métodos HTTP estándar para realizar diferentes acciones:
    - **GET**: Obtener datos (sin modificar el recurso).
    - **POST**: Crear nuevos recursos.
    - **PUT**: Actualizar un recurso existente.
    - **DELETE**: Eliminar un recurso.
    - **PATCH**: Actualizar parcialmente un recurso.

Ejemplo de API REST

Supongamos que queremos crear, leer, actualizar y eliminar usuarios de una aplicación. Nuestra API REST podría tener los siguientes endpoints:

Método HTTP	Endpoint	Acción
`GET`	`/usuarios`	Obtener una lista de usuarios
`GET`	`/usuarios/:id`	Obtener un usuario específico por ID
`POST`	`/usuarios`	Crear un nuevo usuario
`PUT`	`/usuarios/:id`	Actualizar un usuario existente
`DELETE`	`/usuarios/:id`	Eliminar un usuario por ID

Ejemplo práctico (en JSON):

1. **Crear un usuario**:

- **Solicitud**:

```
POST /usuarios
Content-Type: application/json
Body: { "nombre": "Juan", "edad": 25 }
```

- **Respuesta**:

```
{ "mensaje": "Usuario creado", "id": 1 }
```

2. **Obtener todos los usuarios**:

- **Solicitud**:

```
GET /usuarios
```

- **Respuesta**:

```
[
    { "id": 1, "nombre": "Juan", "edad": 25 },
    { "id": 2, "nombre": "Ana", "edad": 30 }
]
```

3. **Actualizar un usuario**:

- **Solicitud**:

```
PUT /usuarios/1
Content-Type: application/json
Body: { "nombre": "Juan Pérez", "edad": 26 }
```

- **Respuesta**:

```
{ "mensaje": "Usuario actualizado" }
```

4. **Eliminar un usuario**:

- **Solicitud**:

```
DELETE /usuarios/1
```

- **Respuesta**:

```
{ "mensaje": "Usuario eliminado" }
```

Beneficios de usar una API REST

1. **Simplicidad**: REST utiliza los principios y verbos HTTP estándar, lo que lo hace fácil de entender.
2. **Escalabilidad**: La separación cliente-servidor permite una escalabilidad eficiente.
3. **Portabilidad**: REST usa formatos estándar como JSON, lo que permite que sea utilizado por diferentes lenguajes de programación.
4. **Independencia de plataforma**: El cliente y el servidor pueden estar en plataformas diferentes y aún comunicarse sin problemas.

Resumen

- **API**: Es un puente entre sistemas, permitiendo la comunicación entre aplicaciones.
- **API REST**: Es un tipo de API diseñada siguiendo los principios de REST. Es popular en aplicaciones web por su simplicidad, flexibilidad y uso de estándares HTTP.

REST es ampliamente utilizado para construir aplicaciones modernas, especialmente en el desarrollo de frontends que necesitan comunicarse con backends. Su diseño basado en recursos y el uso de JSON como formato estándar lo hacen una opción versátil y poderosa.

Definir una API

2. Qué es una API

Una **API** es una interfaz que define cómo los clientes (aplicaciones, sistemas o dispositivos) pueden interactuar con un servicio que normalmente se ejecuta en un servidor. Es el "puente" que permite acceder a los servicios de un servidor sin conocer sus detalles internos.

3. ¿Qué significa "Definir una API"?

Definir una API significa establecer las reglas, rutas y funcionalidades que la API expondrá a los clientes. En otras palabras, es la especificación del conjunto de recursos, métodos y datos que la API pondrá a disposición de las aplicaciones cliente para interactuar con un servidor.

Aspectos Clave de la Definición de una API:

1. **Rutas o Endpoints:**
   - Son las URLs que el cliente utiliza para interactuar con la API.
   - Ejemplo: `/usuarios`, `/productos/:id`, `/auth/login`.
2. **Métodos HTTP:**
   - Especifican la operación que se realizará en un recurso:
     - `GET`: Obtener datos.
     - `POST`: Crear un recurso.
     - `PUT`: Actualizar un recurso existente.
     - `DELETE`: Eliminar un recurso.
3. **Formato de Datos:**
   - Define cómo los clientes y el servidor intercambian datos (usualmente JSON en APIs REST).
4. **Códigos de Estado HTTP:**
   - Informan sobre el resultado de la solicitud:
     - `200 OK`: Éxito.
     - `201 Created`: Recurso creado.
     - `400 Bad Request`: Error en los datos enviados.
     - `404 Not Found`: Recurso no encontrado.
5. **Autenticación y Autorización:**

- Define cómo se asegura que solo usuarios autorizados accedan a ciertos recursos:
  - Tokens de autenticación.
  - Claves API.

6. **Validaciones:**
   - Especificar qué datos son requeridos, qué formatos son aceptables, etc.
   - Ejemplo:
     - El campo `email` debe ser obligatorio y tener un formato válido.

Ejemplo de Definición de una API:

Recurso: Usuarios

Método	Endpoint	Descripción	Cuerpo (JSON)
`GET`	`/usuarios`	Obtener todos los usuarios.	N/A
`POST`	`/usuarios`	Crear un nuevo usuario.	`{ "nombre": "Juan", "email": "juan@example.com" }`
`PUT`	`/usuarios/:id`	Actualizar un usuario existente.	`{ "nombre": "Juan Carlos" }`
`DELETE`	`/usuarios/:id`	Eliminar un usuario.	N/A

Postman y su Uso con APIs REST en Express.js

Postman es una herramienta muy utilizada para probar, documentar y depurar APIs REST. Proporciona una interfaz gráfica que permite a los desarrolladores realizar solicitudes HTTP fácilmente y analizar las respuestas enviadas por el servidor.

En el contexto de una API REST construida con **Express.js**, Postman es útil para:

---

1. ¿Para qué se usa Postman con una API REST en Express.js?

1. **Probar Endpoints:**
   - Puedes enviar solicitudes HTTP (`GET`, `POST`, `PUT`, `DELETE`, etc.) a los endpoints de la API para verificar su funcionalidad.
   - Ejemplo:
     - Probar un endpoint `POST /usuarios` enviando datos JSON como:

```
{
    "nombre": "Juan",
    "email": "juan@example.com"
}
```

2. **Depuración:**
   - Identificar errores o comportamientos inesperados en la API.
   - Examinar las respuestas del servidor, como códigos de estado HTTP, encabezados y datos.

3. **Automatización:**
    - Crear colecciones de solicitudes organizadas por funcionalidad de la API.
    - Realizar pruebas automatizadas utilizando herramientas avanzadas integradas en Postman.
4. **Validar Respuestas:**
    - Verificar que las respuestas de la API son correctas y están en el formato esperado (por ejemplo, JSON).
5. **Simular Clientes:**
    - Emular cómo interactuarían las aplicaciones cliente (como aplicaciones web o móviles) con la API.
6. **Documentar la API:**
    - Crear documentación interactiva de los endpoints directamente en Postman para compartir con otros desarrolladores.

---

2. ¿Qué permite hacer Postman?

- **Enviar Solicitudes HTTP:** Puedes configurar y enviar solicitudes de cualquier tipo (`GET` , `POST` , `PUT` , etc.) con datos personalizados en:
    - Encabezados (Headers).
    - Parámetros de consulta (Query Params).
    - Cuerpo de la solicitud (Body).
- **Configurar Autenticación:** Permite probar APIs que requieren autenticación mediante:
    - Tokens JWT.
    - Claves API.
    - OAuth 2.0.
- **Probar Casos de Uso Complejos:** Simula escenarios como:
    - Errores de validación.
    - Datos incorrectos.
    - Operaciones CRUD completas.
- **Visualizar Respuestas:**
    - Muestra las respuestas del servidor de forma estructurada (JSON, XML, etc.).
    - Permite analizar los códigos de estado HTTP (`200` , `404` , `500` , etc.).
- **Crear Pruebas Automatizadas:**
    - Escribe scripts para validar automáticamente las respuestas.
    - Por ejemplo, puedes verificar que una respuesta tenga un código `200` y contenga un campo específico:

```
pm.test("Código de estado es 200", function () {
    pm.response.to.have.status(200);
});
pm.test("Respuesta contiene 'nombre'", function () {
    pm.response.to.have.jsonBody("nombre");
});
```