FullStack Developer

Servidor con Express de Node.js

¿Qué es un servidor?

Un **servidor** es una computadora o un sistema de computadoras que proporciona recursos, datos, servicios o programas a otros dispositivos, conocidos como clientes, a través de una red. Los servidores pueden tener diferentes roles y ofrecer diversos servicios, como:

1.Servidor web: aloja sitios web y responde a las solicitudes HTTP de los navegadores web de los clientes.

2.Servidor de correo electrónico: gestiona y almacena correos electrónicos para los usuarios y facilita el envío y recepción de correos.

3.Servidor de archivos: almacena y gestiona archivos que pueden ser accedidos y compartidos por los clientes.

4.Servidor de bases de datos: gestiona bases de datos y responde a las consultas de los clientes.

¿Qué es Express?

Express es un framework web minimalista y flexible para Node.js, diseñado para crear aplicaciones y servicios web. Es uno de los frameworks más populares y es ampliamente utilizado para desarrollar APIs y aplicaciones web de manera rápida y eficiente. Algunas de las características principales de Express incluyen:

1.Middleware: Express utiliza middleware para gestionar las solicitudes y respuestas HTTP. El middleware es una serie de funciones que se ejecutan en secuencia para procesar las solicitudes antes de enviarlas a su destino final.

2.Enrutamiento: Proporciona un sistema de enrutamiento robusto que permite definir rutas para manejar diferentes solicitudes HTTP (GET, POST, PUT, DELETE, etc.).

3.Plantillas: Es compatible con motores de plantillas como Pug, EJS y Handlebars para generar HTML dinámico.

4.Compatibilidad con APIs: Facilita la creación de APIs RESTful, lo que permite construir servicios web que pueden ser consumidos por otros clientes o aplicaciones.

5.Modularidad: Permite extender sus funcionalidades mediante la instalación y uso de módulos adicionales de Node.js.

Métodos HTTP

HTTP (Hypertext Transfer Protocol) es un protocolo de comunicación utilizado en la web para la transferencia de datos entre un cliente y un servidor. Los métodos HTTP, también conocidos como verbos HTTP, especifican la acción que el cliente desea realizar en el servidor. Aquí están los métodos HTTP más comunes:

1.GET:

1. **Descripción**: Solicita la representación de un recurso específico. Las solicitudes GET solo deben recuperar datos.
2. **Uso**: Obtener datos (páginas web, imágenes, etc.).
3. **Ejemplo**: Un navegador web hace una solicitud GET para obtener el contenido de una página web.

2.POST:

1. **Descripción**: Envia datos al servidor para crear o actualizar un recurso.
2. **Uso**: Enviar datos de formularios, subir archivos.
3. **Ejemplo**: Un formulario de registro de usuario que envía datos al servidor para crear un nuevo usuario en los detalles del perfil.

3. **PUT**:
    1. **Descripción**: Reemplaza todas las representaciones actuales del recurso de destino con los datos proporcionados.
    2. **Uso**: Actualizar un recurso existente o crear uno nuevo si no existe.
    3. **Ejemplo**: Actualizar los detalles de un producto en una base de datos.

4. **DELETE**:
    1. **Descripción**: Elimina el recurso especificado.
    2. **Uso**: Eliminar recursos.
    3. **Ejemplo**: Eliminar una cuenta de usuario.

5. **PATCH**:
    1. **Descripción**: Aplica modificaciones parciales a un recurso.
    2. **Uso**: Actualizar parcialmente un recurso.
    3. **Ejemplo**: Actualizar solo el nombre de usuario

¿Qué es HTTPS?

HTTPS (Hypertext Transfer Protocol Secure) es una versión segura de HTTP. HTTPS utiliza protocolos de seguridad como SSL (Secure Sockets Layer) o TLS (Transport Layer Security) para cifrar la comunicación entre el cliente y el servidor. Aquí están los aspectos clave de HTTPS:

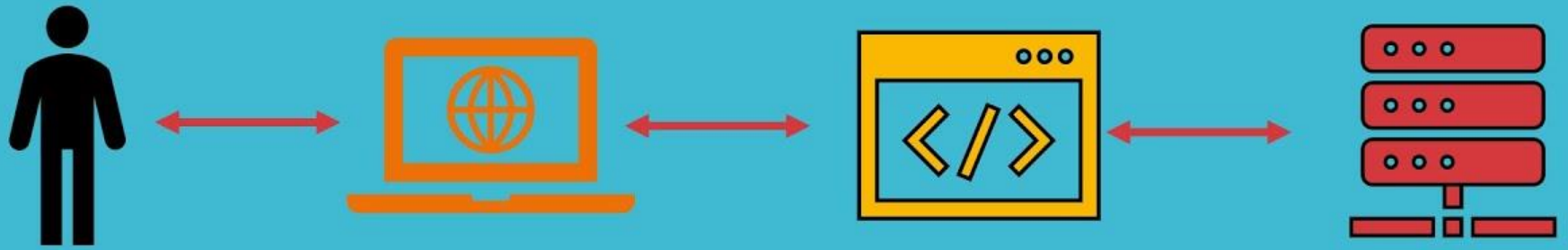
1.Cifrado: HTTPS cifra los datos transmitidos, lo que significa que los datos no pueden ser leídos por terceros mientras están en tránsito. Esto protege información sensible como contraseñas, números de tarjetas de crédito y otros datos personales.

2.Integridad de los datos: HTTPS asegura que los datos no sean modificados o corrompidos durante la transferencia, ya que cualquier alteración sería detectada.

3.Autenticación: HTTPS garantiza que el cliente está comunicándose con el servidor correcto, evitando ataques de suplantación de identidad mediante el uso de certificados digitales emitidos por autoridades certificadoras (CAs).

4.Confidencialidad: Solo el cliente y el servidor pueden leer los datos transmitidos, garantizando la privacidad de la comunicación.
La adopción de HTTPS es fundamental para la seguridad en la web y es especialmente importante para sitios que manejan información sensible. Los navegadores modernos muestran advertencias cuando un sitio no utiliza HTTPS, incentivando a los desarrolladores y administradores de sitios web a implementar HTTPS.

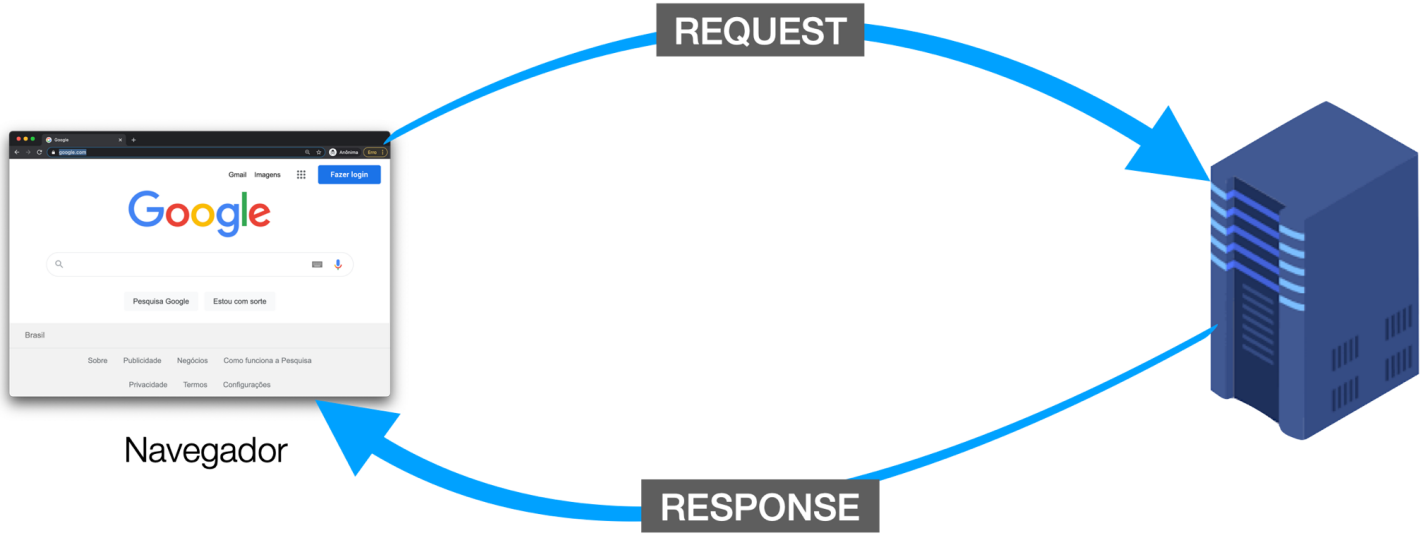
Cliente / Servidor



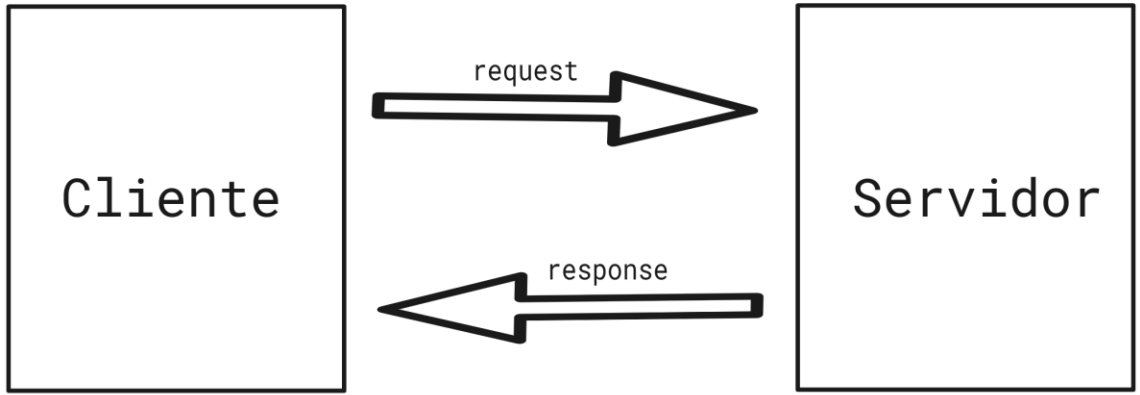

Cliente
1
Servidor
Petición
Respuesta
2

CLIENTE

SERVIDOR



Protocolo HTTP



Una API (Interfaz de Programación de Aplicaciones) es un conjunto de reglas y definiciones que permite a diferentes software comunicarse entre sí. En esencia, es una interfaz que permite que diferentes sistemas interactúen de manera estandarizada y eficiente. Aquí te explico más a fondo:

1. Definición y Función

•**Interfaz**: Una API define cómo se deben realizar las solicitudes y cómo se deben manejar las respuestas. Proporciona un conjunto de métodos y protocolos que las aplicaciones pueden utilizar para interactuar con otros servicios o sistemas.

•**Comunicación**: Permite que diferentes aplicaciones o componentes de software se comuniquen entre sí, incluso si están escritos en diferentes lenguajes de programación o ejecutándose en diferentes plataformas.

2. Tipos de APIs

•**APIs Web**: Son APIs que se comunican a través de la web usando protocolos HTTP/HTTPS. Las APIs REST y GraphQL son ejemplos comunes de APIs web. Permiten a los clientes (como aplicaciones web o móviles) enviar solicitudes y recibir respuestas a través de la web.

•**APIs de Biblioteca o SDK**: Proporcionan interfaces para interactuar con bibliotecas de software o kits de desarrollo. Por ejemplo, una API de una biblioteca gráfica permite a los desarrolladores dibujar gráficos en sus aplicaciones sin tener que escribir todo el código desde cero.

•**APIs de Sistema**: Permiten la comunicación entre aplicaciones y el sistema operativo, por ejemplo, para acceder a funciones del hardware o gestionar recursos del sistema.

3. Componentes de una API

•**EndPoints**: Son las URLs a las que se puede enviar una solicitud. Cada endpoint suele estar asociado a una función específica en el servicio que proporciona la API.

•**Métodos HTTP**: Definen la acción que se realizará en el recurso. Los métodos comunes son:

- •GET: Para recuperar datos.
- •POST: Para enviar datos o crear nuevos recursos.
- •PUT: Para actualizar recursos existentes.
- •DELETE: Para eliminar recursos.

•**Parámetros**: Son datos adicionales enviados en la solicitud para especificar qué recursos o acciones se desean. Pueden ser parte de la URL, el cuerpo de la solicitud o los encabezados HTTP.

•**Códigos de Estado HTTP**: Indican el resultado de la solicitud. Ejemplos incluyen:

- •200 OK: La solicitud fue exitosa.
- •404 Not Found: El recurso solicitado no fue encontrado.
- •500 Internal Server Error: Hubo un error en el servidor.

Códigos de estado HTTP

Los códigos de estado HTTP son códigos numéricos que indican el resultado de una solicitud HTTP. Estos códigos están organizados en cinco categorías principales, cada una representando un tipo de respuesta. Aquí tienes una lista de los códigos de estado más comunes y sus significados:

1. Códigos de Respuesta Informativa (100-199)
Estos códigos indican que la solicitud se ha recibido y que el proceso está en curso.
•**100 Continue**: El servidor ha recibido parte de la solicitud y el cliente debe continuar enviando el resto de la solicitud.
•**101 Switching Protocols**: El servidor está cambiando los protocolos según lo solicitado por el cliente.

2. Códigos de Respuesta Exitosos (200-299)
Estos códigos indican que la solicitud fue exitosa y el servidor ha devuelto la respuesta solicitada.
•**200 OK**: La solicitud ha sido exitosa. La respuesta depende del método HTTP utilizado.
•**201 Created**: La solicitud ha sido exitosa y ha resultado en la creación de un nuevo recurso.

•**202 Accepted**: La solicitud ha sido aceptada para procesamiento, pero el procesamiento no se ha completado.
•**204 No Content**: La solicitud ha sido exitosa, pero no hay contenido que devolver.

3. Códigos de Respuesta Redirección (300-399)
Estos códigos indican que el cliente debe realizar acciones adicionales para completar la solicitud.
•**301 Moved Permanently**: El recurso solicitado ha sido movido de manera permanente a una nueva URL.
•**302 Found**: El recurso solicitado ha sido encontrado en una URL diferente, pero la solicitud debe seguir usando la URL original.
•**304 Not Modified**: El recurso no ha sido modificado desde la última solicitud. El cliente puede usar la versión en caché.

4. Códigos de Respuesta de Error del Cliente (400-499)
Estos códigos indican que hubo un error en la solicitud realizada por el cliente.
•**400 Bad Request**: La solicitud no se puede procesar debido a un error del cliente (por ejemplo, sintaxis incorrecta).
•**401 Unauthorized**: La solicitud requiere autenticación. El cliente debe proporcionar credenciales válidas.
•**403 Forbidden**: El cliente está autenticado, pero no tiene permiso para acceder al recurso.

•**404 Not Found**: El recurso solicitado no se encuentra en el servidor.
•**405 Method Not Allowed**: El método HTTP utilizado no está permitido para el recurso solicitado.

5. Códigos de Respuesta de Error del Servidor (500-599)
Estos códigos indican que hubo un error en el servidor al procesar la solicitud.
•**500 Internal Server Error**: El servidor encontró un error inesperado y no pudo completar la solicitud.
•**502 Bad Gateway**: El servidor actuó como una puerta de enlace o proxy y recibió una respuesta inválida del servidor upstream.
•**503 Service Unavailable**: El servidor no está disponible temporalmente (por ejemplo, debido a mantenimiento).
•**504 Gateway Timeout**: El servidor actuó como una puerta de enlace o proxy y no recibió una respuesta a tiempo del servidor upstream.
Estos códigos son parte del protocolo HTTP y ayudan a que el cliente y el servidor se comuniquen de manera efectiva, proporcionando información sobre el resultado de las solicitudes y facilitando la gestión de errores y redirecciones.

JSON (JavaScript Object Notation)

•**Qué es**: JSON es un formato ligero de intercambio de datos que es fácil de leer y escribir para los humanos y fácil de analizar y generar para las máquinas.
•**Uso**: Se utiliza comúnmente para enviar datos entre un cliente y un servidor en aplicaciones web. Por ejemplo, una API puede devolver datos en formato JSON que el cliente puede procesar.
•**Formato**: Los datos en JSON están representados como objetos y arreglos. Un objeto JSON se escribe entre llaves {}, y un arreglo JSON se escribe entre corchetes []. Los pares clave-valor están separados por comas.

CORS (Cross-Origin Resource Sharing)

•**Qué es**: CORS es un mecanismo de seguridad que permite a los recursos de un servidor web ser solicitados desde un dominio diferente al dominio del servidor. Esto es relevante cuando una aplicación web en un dominio quiere hacer peticiones a un servidor en otro dominio.
•**Problema que resuelve**: Los navegadores implementan políticas de seguridad llamadas Same-Origin Policy que restringen cómo los documentos o scripts cargados desde un origen pueden interactuar con recursos de otro origen. CORS es una forma de permitir estas interacciones controladas.
•**Configuración**: En el servidor, puedes configurar los encabezados CORS para permitir solicitudes de ciertos orígenes, métodos HTTP, o encabezados específicos.

Crear un Nuevo Proyecto

Vamos a crear un nuevo proyecto, nuestro primer servidor, vamos a crear una API con Express.js.

Para eso vamos a seguir una serie de pasos que cada vez que vamos a iniciar un proyecto nuevo, seguiremos estos comandos

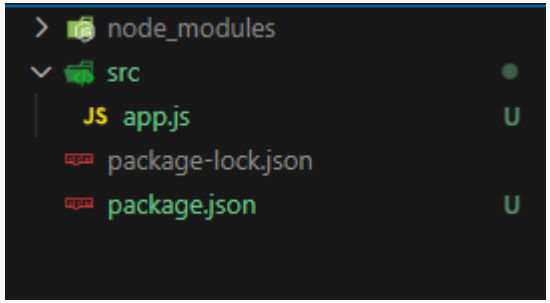
Crearemos nuestro package.json

```
npm init -y
```

Instalamos dependencias

```
npm install express cors
```

Configuramos nuestro package.json y creamos nuestra estructura de carpetas



Dentro de la carpeta “src” creamos el archivo app.js

Y el package.json nos debe quedar como mostramos a la derecha.

Necesitaremos agregar o modificar estar 3 propiedades

```
"type": "module",
  "main": "app.js",
  "scripts": {
    "dev": "nodemon ./src/app.js"
  },
```

```
{
  "name": "code",
  "version": "1.0.0",
  "description": "",
  "type": "module",
  "main": "app.js",
  "scripts": {
    "dev": "nodemon ./src/app.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.19.2"
  }
}
```

Ahora definamos nuestro script para crear nuestro primer servidor

Vamos a importar express y cors de la siguiente manera

```
import express from 'express';
import cors from 'cors';
```

Creamos una constante “app” que almacena el retorno de la función express()

```
const app = express();
```

Utilizaremos estos middlewares que nos permitirá parsear a JSON lo que venga desde un cliente, y permitir a cualquier origen hacer peticiones a nuestro servidor, ya lo vamos a configurar más adelante.

```
// Middlewares
app.use(cors())
app.use(express.json())
```

El primer endpoint, este será el primero de muchos, ya que cada petición que nos haga algún cliente deberá tener un endpoint

```
// Endpoints (rutas)
app.get('/', (request, response) => {

    const miMensaje = 'Mi primera api'

    response.status(200).json({ status: 200, message: miMensaje })
})
```

Y finalmente, levantamos nuestro servidor

```
const PORT = 3000
// Iniciar el servidor
app.listen(PORT, () => {
    console.log(`Servidor escuchando en http://localhost:${PORT}`);
});
```

Definimos un puerto, luego llamamos a la función “listen” que recibe como primer parámetro el puerto y luego una callback que en este caso, solo mostrará un mensaje por consola para saber que nuestro servidor está funcionando

Ejercicio

Crear una api e intentar consumirla desde un cliente (front => una app de js o de React)