EXPRESS (Express con JSON)

Express en un framework que permite construir aplicaciones web con Node.js. Nos va a permitir crear servidores web, manejar rutas, gestionar solicitudes y respuestas.

1. Instalar Express

- 1.1. Instalar Node.js. HECHO
- 1.2. Crear Proyecto de Express. Desde el directorio en el que se va a crear el proyecto. Se va a crear un package.json que almacena información sobre el proyecto.

  npm init -y

- 1.3. Instalar de Express.

  npm install express

2. Primeros pasos.

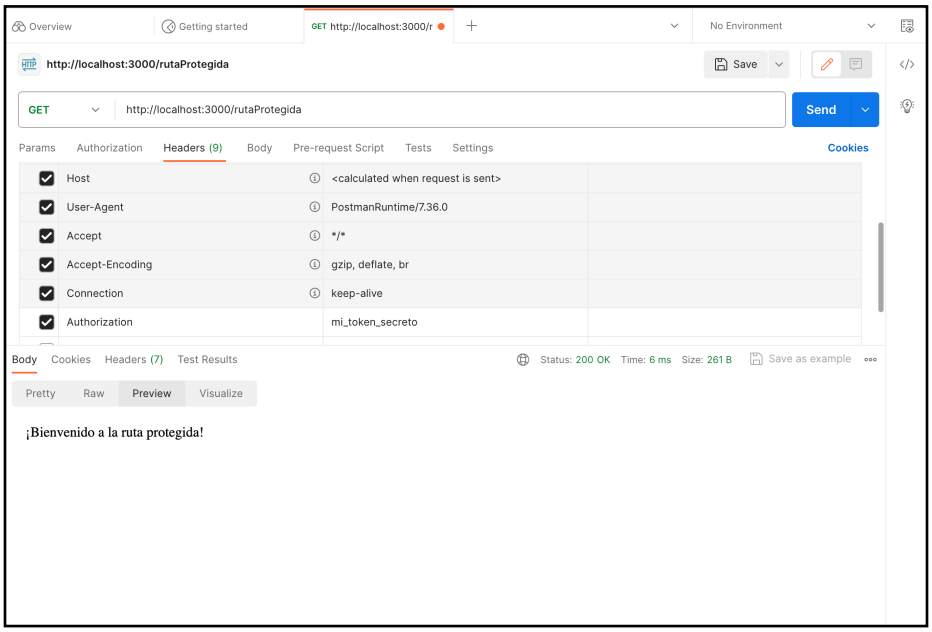
- En primer lugar, será necesario importar el módulo express y crear una instancia del mismo.
- Definir las rutas sobre la instancia creada en el paso anterior, aquí se definen también las acciones. En primer lugar, se define una ruta GET para la URL indicada, después se utilizara el manejador de la ruta que se ejecutará cuando se reciba una solicitud GET en la ruta especificada.

```
app.get('/', (request, response) => {
        response.send(Respuesta de la llamada a la ruta);
};
```

- Iniciar el servidor con app.listen(PORT, () => { … });

Ejercicio 1: Crear un servidor con Express con el mensaje Hola Mundo

3. Pruebas - POSTMAN



Recuerda:

Para acceder a las cabeceras de una solicitud HTTP en Express.

request.headers.authorization;

Donde request es el objeto de solicitud que se pasa como argumento a la función manejadores, contiene información sobre la solicitud HTTP entrante, como parámetros, cuerpo, encabezados, y más.

El headers (cabecera) es un objeto que contiene todos los encabezados de la solicitud. Recuerda, que los encabezados son pares clave-valor que transportan información sobre la solicitud o la respuesta..

Se accede al valor del encabezado que se introduce como clave. .

4. Middleware en Express: se trata de una función que tiene acceso al objeto de solicitud (`req`), al objeto de respuesta (`res`), y a la siguiente función en la pila de middleware (`next`). Se utiliza para realizar tareas intermedias durante el ciclo de vida de una solicitud. Pueden modificar la solicitud, la respuesta o terminar el ciclo llamando a la función `next()`.

4.1. use(). Para utilizar middleware y realizar acciones intermedias se utiliza use. Es posible tener más de un middleware que serán ejecutados en orden, pasando cada next la ejecución a la siguiente función middleware.

```
app.use((req, res, next) => {
  next(); // Llama a la siguiente función en la pila de middleware
});
```

Ejercicio 2: Crear un servidor con Express que implemente un sistema de verificación de autenticación utilizando middleware con tantas funciones como sean necesarias:

5. Parámetros de Ruta y Consulta.

- **PARÁMETROS DE RUTA:** Se puede capturar parámetros directamente desde la URL utilizando `:` + el nombre del parámetro.

```
app.get('/ruta/:parametro', (req, res) => {
  const parametro1 = req.params.parametro;
  res.send(`PARAM: ${parametro1}`);
});
```

- **PARÁMETROS DE CONSULTA:** Los parámetros de consulta se incluyen en las URL tras del signo de interrogación ?. El acceso a estos parámetros es a través **req.query.**

```
app.get('/ruta', (req, res) => {
  const { q, tipo } = req.query;
  res.send(`Búsqueda: ${q}, Tipo: ${tipo}`);
});
```

- Ejemplo: /ruta?paramA=Hola&paramB=Adios, req.query.paramA será Hola y req.query.paramB es Adios.

Ejercicio 3: Crear una aplicación web que reciba por parámetro varios valores (nombre de una tarea, autor). Imprime su resultado en el navegador. Además, si se recibe por parámetro de ruta el nombre del autor se devolverá un saludo personalizado.

6 GET Y POST

6.1. GET: se utiliza para manejar solicitudes HTTP GET. Estas solicitudes son utilizadas para obtener datos del servidor:

```
const express = require('express');
const app = express();
const PORT = 3000;

app.get('/', (req, res) => {
  res.send('¡Hola, mundo! Esta es una solicitud GET.');
});

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

6.2. POST: se utiliza para manejar solicitudes HTTP POST. Estas solicitudes son utilizadas para enviar datos al servidor, por ejemplo, al enviar un formulario.

```
const express = require('express');
const app = express();
const PORT = 3000;

app.use(express.json()); //Ejemplo para el caso en el que se requiera parsear JSON

app.post('/formulario', (req, res) => {
  const { nombre, correo } = req.body;
  res.send(`¡Hola, ${nombre}! Correo: ${correo}. Esta es una solicitud POST.`);
});

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

7. Formularios:

Para manejar datos de formularios en Express, se utiliza el middleware especializado para procesar los datos enviados desde un formulario, permite pasear el cuerpo de la solicitud.

```
app.use(express.urlencoded({ extended: true }));
```

Tras configurar el middleware para analizar datos de formularios, se podrá acceder a los datos:

```
app.post('/ruta', (req, res) => {
  const { param1, param2, param3,... } = req.body;
  res.send(`${param1}, ${param2}`);
});
```

8. Archivos Estáticos. Como en el caso anterior, se querría recoger los datos un formulario, que prevemos estará contenido en un fichero html. ¿Cómo nos permite Express servir este tipo de archivos (html, CSS, imágenes o scripts)?. En el ejemplo anterior, se colocarán en el directorio public.

app.use(express.static(…));

```
proyecto/
├── public/
│   ├── styles.css
│   ├── images/
│   │   └── logo.png
│   └── scripts/
│       └── main.js
├── app.js
└── ...
```

```
app.get('/ruta', (req, res) => {
  res.sendFile(__dirname + '/public/pagina.html');
});
```

Ejercicio 4: Crear una aplicación web con un formulario HTML como el que se muestra en la imagen. Debe mostrar luego un saludo en el que se muestran los valores recogidos en el formulario.

9. CORS es una política de seguridad implementada por los navegadores web que restringe cómo los recursos de una página web pueden ser solicitados desde otro dominio diferente al dominio que sirvió la página original. Esto es una medida de seguridad para evitar solicitudes no autorizadas que podrían comprometer la seguridad de un sitio web. El navegador bloqueará estas solicitudes por defecto para evitar posibles ataques.

npm install cors

```
const express = require('express');
const cors = require('cors'); // Importa el módulo cors
const app = express();
```

```
const PORT = 3000;

// Middleware para habilitar CORS
app.use(cors());

// Ruta de ejemplo
app.get('/', (req, res) => {
  res.send('¡Hola, mundo! Esta es una solicitud GET.');
});

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

10**. Acceder a JSON.**

app.use.express.json()

Este middleware está diseñado para analizar el cuerpo de las solicitudes HTTP que tienen el tipo de contenido `application/json`.

Al usar `app.use(express.json());`, estás aplicando este middleware a todas las rutas de tu aplicación. Para todas las solicitudes entrantes, Express intentará analizar el cuerpo de la solicitud como JSON y lo hará disponible en `req.body`.

La diferencia principal entre las operaciones asíncronas y síncronas radica en cómo manejan el flujo de ejecución del programa:

1. Operaciones Síncronas: las instrucciones se ejecutan una tras otra, de manera secuencial. Es decir, cada instrucción espera a que la anterior se complete antes de ejecutarse. Esto significa que una operación bloqueante, como la lectura o escritura de un archivo, detendrá la ejecución del programa hasta que se complete.

2. Operaciones Asíncronas: las instrucciones no esperan a que la anterior se complete antes de ejecutarse. En lugar de eso, el programa continúa ejecutándose mientras se llevan a cabo las operaciones en segundo plano. Este enfoque es especialmente útil en situaciones en las que no se quiere bloquear la ejecución del programa mientras se realizan operaciones que pueden llevar tiempo, como la lectura de archivos o solicitudes a una API.

Se debe tener en cuenta las siguientes instrucciones clave:

```
const fs = require('fs').promises; /* El módulo fs.promises proporciona funciones del sistema de archivos como promises, lo que facilita escribir código asíncrono de manera más concisa y legible.*/

app.get('/leer-datos', async (req, res) => { /*el uso de promises permite el uso asíncrono, para ello se utiliza la palabra async que indica que la función es asíncrona y puede contener operaciones asíncronas que usan await.*/
```

const data = await fs.readFile(filePath, 'utf-8'); /*await: La palabra clave await se utiliza dentro de funciones marcadas como async para esperar a que una promise se resuelva antes de continuar con la ejecución del código.*/

Ejercicio 5: Crear una aplicación web con Express y JSON, que añada nuevos datos a un fichero de datos.json, NO LOS SOBREESCRIBA.

11. FETCH

La función `fetch` es una interfaz de JavaScript que proporciona una forma de realizar solicitudes de red, como recuperar recursos de una URL o enviar datos a un servidor. Se utiliza comúnmente para interactuar con servicios web y API en el contexto de aplicaciones web.

1. Para realizar una solicitud POST a una URL se tiene que implementar un objeto de configuración que incluye el método (method: 'POST'), las cabeceras (headers) en las que se indican que se envía JSON y el cuerpo (body) de la solicitud, que se convierte a JSON mediante JSON.stringify.

2. Después, la respuesta de la solicitud es procesada con then, primero, se convierte la respuesta a JSON (response.json()), y después con los datos resultados del procesamiento anterior se realiza la transformación que se necesite de los datos, por ejemplo, crear una lista de elementos li para mostrar en la página html.

```
fetch('url', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ descripcion }),
  })
  .then(response => response.json())
  .then(data => {
    …
  })
  .catch(error => console.error('Error al agregar tarea:', error));
});
```