

FullStack Developer

Express JS – Controllers, routes y
modularización

Desarrollo

Cuando desarrollamos aplicaciones web con Express.js, una estructura clara y organizada es esencial para mantener el código mantenible y escalable.

1. Controladores

Los controladores son componentes que manejan la lógica de negocio de la aplicación. En Express.js, típicamente un controlador se encarga de recibir la solicitud HTTP, procesar la información, interactuar con modelos o bases de datos si es necesario, y finalmente enviar una respuesta al cliente.

2. Rutas

Las rutas son las definiciones de los endpoints (URLs) que la aplicación expone y que están vinculadas a los controladores para procesar las solicitudes. Express.js utiliza el método Router() para separar las rutas y hacerlas más manejables.

3. Modularización

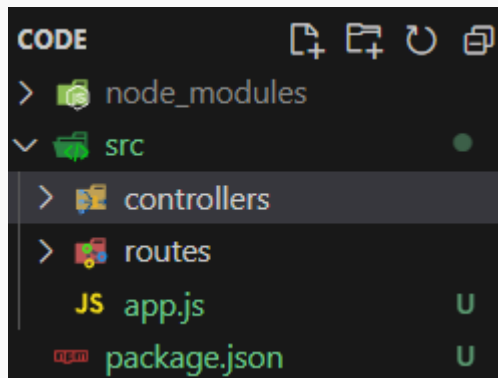
La modularización implica estructurar la aplicación de manera que cada parte (como rutas, controladores, modelos, etc.) esté separada en diferentes archivos o módulos. Esto mejora la legibilidad y el mantenimiento del código.

Proyecto - Portfolio

Empecemos a realizar un proyecto con estos conceptos, vamos a empezar por la configuración principal y luego crear la carpeta src con el entrypoint app.js

Esto ya lo empezamos a hacer en la clase pasada, así que es cuestión de repetir el proceso.

Entonces, sigamos con los temas, nuevos, como separar la lógica de programación, empecemos creando nuevas carpetas.



```
//app.js
import express from 'express';
import cors from 'cors';

const app = express();

app.use(express.json());
app.use(cors());

const PORT = 3000;

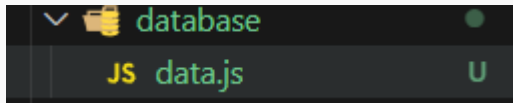
app.listen(PORT, () => console.log(`Listening on ${PORT}`));
```

Vamos a crear un controlador llamado “proyectos” el módulo se llamará **proyectos.controller.js**. Además su respectivo módulo de rutas llamado **proyectos.route.js**. Cada uno en sus respectivas carpetas.

Vamos a crear un array de proyectos que simulará la existencia y manipulación de datos

```
export default [  
  {  
    id: 1,  
    url: "https://github.com/carlos8788/proyecto1",  
    privado: false,  
    fecha: "2022-03-15",  
    tecnologias: ["React", "Node.js", "Express", "MongoDB", "Docker"],  
    nombre: "Dashboard de Analítica"  
  },  
  {  
    id: 2,  
    url: "https://github.com/carlos8788/proyecto2",  
    privado: true,  
    fecha: "2023-01-20",  
    tecnologias: ["Angular", "Firebase", "Tailwind CSS", "TypeScript", "Jest"],  
    nombre: "Aplicación de Gestión de Tareas"  
  }  
]
```

Esto lo vamos a colocar en un archivo llamado data.js en una carpeta aparte en src/database



Luego vamos a crear los controladores

En proyectos.controller.js

```
import proyectos from '../database/data.js'
```

Traer todos los proyectos

```
export const getProyectos = (req, res) => {  
  res.status(200).json({status: 200, message: 'Proyectos', payload: proyectos})  
}
```

Con este controlador vamos a poder traer todos los proyectos que se encuentren en nuestro array

Traer un proyecto por ID

```
export const getProyectoById = (req, res) => {  
  const { id } = req.params;  
  const proyecto = proyectos.find(p => p.id === parseInt(id));  
  if (proyecto) {  
    res.status(200).json({status: 200, message: 'Proyecto encontrado', payload: proyecto})  
  } else {  
    res.status(404).json({status: 404, message: 'Proyecto no encontrado'})  
  }  
}
```

En este caso vamos a ver que estamos devolviendo un producto pero por su ID, usando parámetros

Parámetros en Express

En Express, los parámetros de ruta son una parte esencial para manejar las solicitudes dinámicas. Los parámetros son segmentos de la URL que se esperan que varíen entre diferentes solicitudes. Por ejemplo, en una ruta como `/proyectos/:id`, el `:id` es un parámetro que Express captura y hace accesible a través del objeto `req.params`.

```
const { id } = req.params;
```

Aquí, `req.params` contiene todos los parámetros de ruta incluidos en la URL de la solicitud. En este caso, `id` se extrae de `req.params`, lo cual implica que la URL esperada sería algo como `/proyectos/123`, donde 123 es un ejemplo de un ID de proyecto.

El uso de parámetros permite que las aplicaciones web sean dinámicas y adaptables. Pueden responder de manera específica a diferentes entradas sin necesidad de tener rutas estáticas para cada posible solicitud. Por ejemplo, en lugar de tener rutas separadas para cada proyecto, una sola ruta parametrizada maneja todas las posibles solicitudes de proyectos basadas en su ID. Esto hace que el código sea más limpio, más fácil de mantener y escalar.

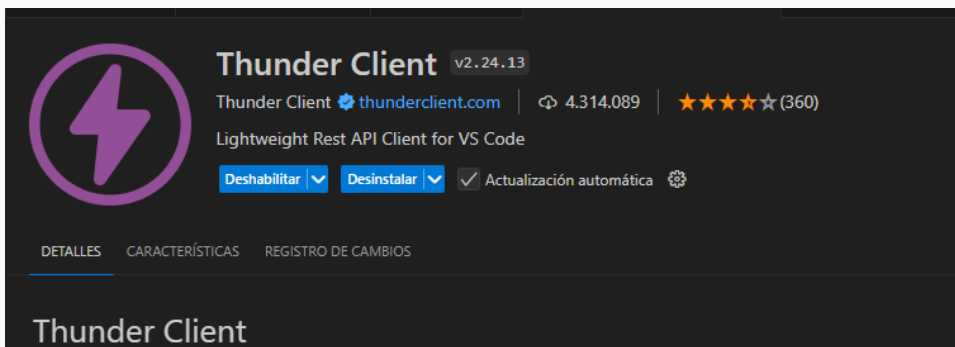
Eliminar un proyecto por ID

```
export const deleteProyecto = (req, res) => {
  const { id } = req.params;
  const index = proyectos.findIndex(p => p.id === parseInt(id));
  if (index !== -1) {
    proyectos.splice(index, 1);
    res.status(200).send({staus: 200, message: 'Proyecto eliminado'});
  } else {
    res.status(404).json({staus: 404, message: 'Proyecto no encontrado'})
  }
}
```

Acá podremos eliminar un proyecto utilizando también el ID, dado que esto nos va a permitir eliminar un solo proyecto si es que encuentra el id

Thunder Client

Thunder Client es una extensión de Visual Studio Code diseñada para simplificar las pruebas de API. Funciona como un cliente HTTP ligero directamente dentro del entorno de desarrollo integrado (IDE), lo que permite a los desarrolladores probar, validar y documentar APIs sin necesidad de salir del editor o utilizar herramientas externas más complejas

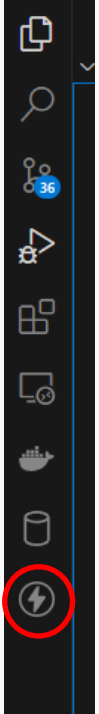


Instalar Thunder Client

En **Marketplace de Extensiones** vamos a buscar Thunder Client e instalamos, es un proceso sencillo.

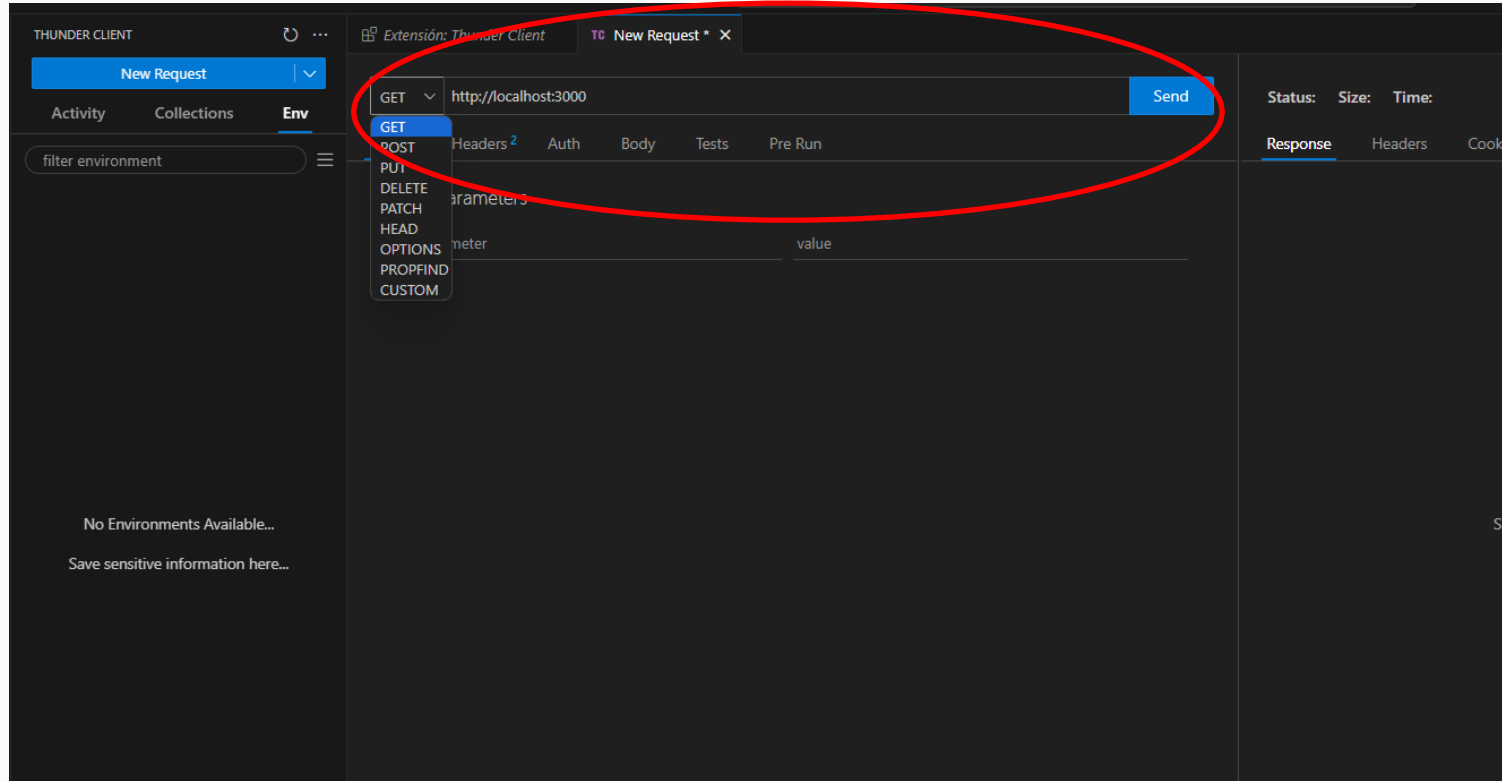
Como se usa

Una vez instalada, podemos acceder a Thunder Client desde la barra lateral izquierda de Visual Studio Code.



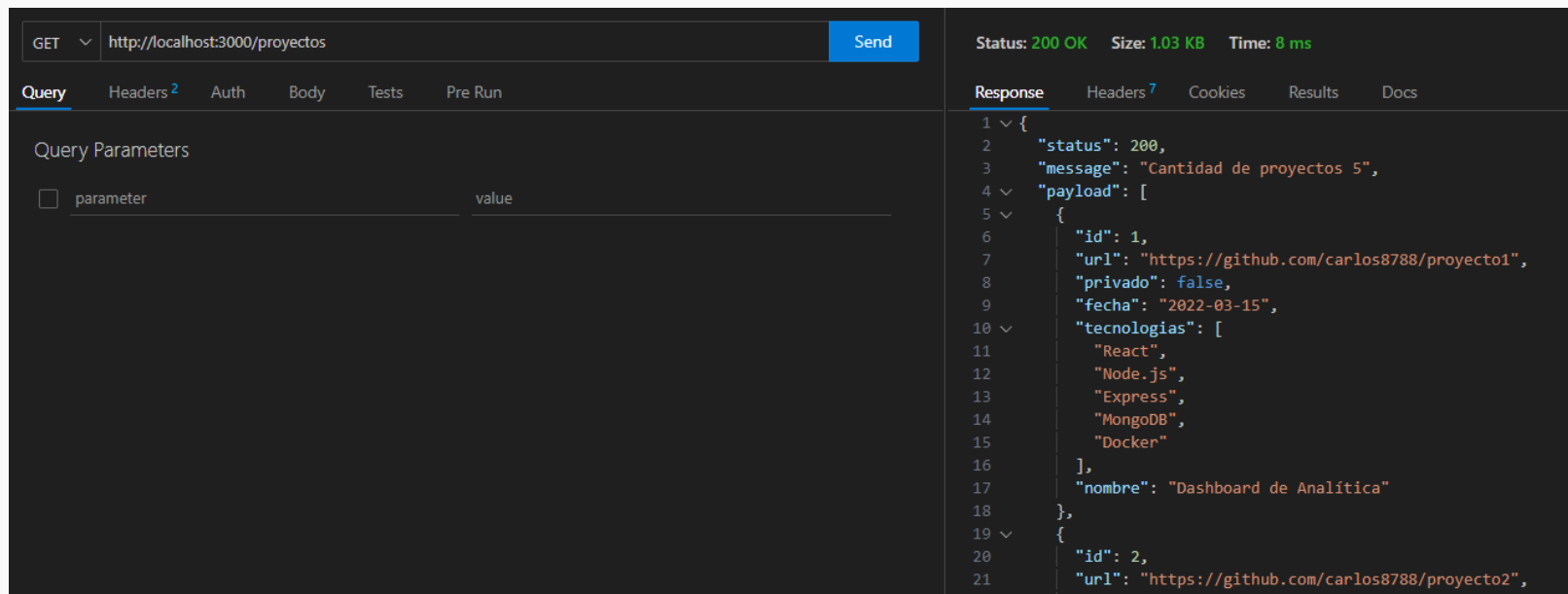
Este es el ícono que nos debe aparecer en la barra

Una vez presionado el ícono debemos ver la interfaz



En la barra de direcciones que está redondeada con rojo podremos colocar la url para empezar a probar nuestro proyecto, además tenemos el menú para definir que tipo de solicitud haremos GET, PUT, DELETE o POST

Resultado de un GET a <http://localhost:3000/proyectos>



The screenshot displays a web client interface with the following details:

- Request:** Method: GET, URL: <http://localhost:3000/proyectos>. A "Send" button is visible.
- Response Status:** 200 OK, Size: 1.03 KB, Time: 8 ms.
- Response Body (JSON):**

```
1 {
2   "status": 200,
3   "message": "Cantidad de proyectos 5",
4   "payload": [
5     {
6       "id": 1,
7       "url": "https://github.com/carlos8788/proyecto1",
8       "privado": false,
9       "fecha": "2022-03-15",
10      "tecnologias": [
11        "React",
12        "Node.js",
13        "Express",
14        "MongoDB",
15        "Docker"
16      ],
17      "nombre": "Dashboard de Analítica"
18    },
19    {
20      "id": 2,
21      "url": "https://github.com/carlos8788/proyecto2",
```

En el costado derecho vemos el resultado de la solicitud GET a esa URL

¿Qué es req.body?

req.body es un objeto que contiene los datos enviados en el cuerpo de una solicitud POST, PUT, PATCH, o similar. En un servidor Express, este objeto no está disponible por defecto en las solicitudes entrantes; necesita ser parseado y agregado a la solicitud mediante un middleware como `express.json()` para JSON o `express.urlencoded()` para datos codificados en URL.

```
const { url, privado, fecha, tecnologias, nombre } = req.body;
```

Esta línea desestructura el objeto req.body para extraer las propiedades url, privado, fecha, tecnologias, y nombre. Cada una de estas propiedades representa un valor que el cliente puede enviar para actualizar el proyecto. Por ejemplo, un cliente podría enviar una solicitud PUT con un cuerpo JSON como este:

```
{
  "url": "https://github.com/nuevoUsuario/proyectoActualizado",
  "privado": true,
  "fecha": "2024-08-01",
  "tecnologias": ["Node.js", "React"],
  "nombre": "Nuevo Nombre del Proyecto"
}
```

Veamos como manejar las solicitudes de POST y UPDATE enviando un json por el body

Crear un proyecto

```
export const crearProyecto = (req, res) => {
  const { url, privado, fecha, tecnologias, nombre } = req.body;

  // Suponiendo que tenemos una forma de generar un ID único, puede ser un contador o UUID
  const nuevoId = proyectos.length + 1; // Esto es solo un ejemplo simplista

  const nuevoProyecto = {
    id: nuevoId,
    url,
    privado,
    fecha,
    tecnologias,
    nombre
  };

  // Agregamos el nuevo proyecto al array de proyectos
  proyectos.push(nuevoProyecto);

  // Enviamos una respuesta indicando que el proyecto fue creado exitosamente
  res.status(201).json({
    status: 201,
    message: 'Proyecto creado exitosamente',
    payload: nuevoProyecto
  });
}
```

Actualizar un proyecto por ID

```
export const updateProyecto = (req, res) => {
  const { id } = req.params;
  const { url, privado, fecha, tecnologias, nombre } = req.body;
  const proyecto = proyectos.find(p => p.id === parseInt(id));
  if (proyecto) {
    proyecto.url = url || proyecto.url;
    proyecto.privado = privado === undefined ? proyecto.privado : privado;
    proyecto.fecha = fecha || proyecto.fecha;
    proyecto.tecnologias = tecnologias || proyecto.tecnologias;
    proyecto.nombre = nombre || proyecto.nombre;
    res.status(200).json({ status: 200, message: 'Proyecto actualizado',
payload: proyecto})
  } else {
    res.status(404).json({ status: 404, message: 'Proyecto no encontrado'})
  }
}
```


Conclusión de la clase

Hasta este punto, hemos abordado cómo estructurar una aplicación Express.js de manera modular, separando controladores, rutas y datos para mantener un código limpio, mantenible y escalable. Estos conceptos son fundamentales para construir aplicaciones más complejas y robustas, ya que permiten que cada parte del sistema tenga responsabilidades claras y sea fácilmente adaptable.

La modularización no solo facilita el desarrollo, sino que también permite que las aplicaciones crezcan sin volverse inmanejables. A lo largo de este proyecto, hemos trabajado con rutas y controladores que simulan la interacción con datos, lo que prepara el terreno para trabajar con bases de datos reales en las próximas clases.

Actividad optativa

Como actividad optativa, podés implementar una ruta adicional en tu proyecto que permita **actualizar** los datos de un proyecto. Usa `req.body` para obtener los nuevos valores de un proyecto existente y asegurate de hacer las validaciones correspondientes (por ejemplo, que el ID del proyecto exista antes de intentar actualizarlo). Esta actividad te ayudará a consolidar los conceptos que vimos en esta clase.