

Node.js y Express: APIs REST

1. Introducción a Node.js

Node.js es un entorno de ejecución de JavaScript construido sobre el motor V8 de Chrome. Permite ejecutar JavaScript fuera del navegador, principalmente para crear aplicaciones del lado del servidor.

Características principales:

- Event-driven y non-blocking I/O: maneja múltiples conexiones sin bloquear
- Single-threaded con event loop: eficiente en memoria
- NPM: el gestor de paquetes más grande del mundo
- Cross-platform: funciona en Windows, macOS y Linux

Crear un servidor HTTP básico:

```
const http = require('http');
```

```
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('¡Hola Mundo!');
});

server.listen(3000, () => {
  console.log('Servidor en http://localhost:3000');
});
```

Node.js es ideal para:

- APIs REST y GraphQL
- Aplicaciones en tiempo real (chat, colaboración)
- Microservicios
- Herramientas CLI
- Streaming de datos

2. Módulos en Node.js

Node.js tiene un sistema de módulos para organizar y reutilizar código.

CommonJS (sistema tradicional):

```
// math.js - exportar
module.exports = {
  sumar: (a, b) => a + b,
  restar: (a, b) => a - b
};

// app.js - importar
const math = require('./math');
console.log(math.sumar(2, 3)); // 5
```

ES Modules (moderno, requiere "type": "module" en package.json):

```
// math.js - exportar
export const sumar = (a, b) => a + b;
export const restar = (a, b) => a - b;

// app.js - importar
import { sumar, restar } from './math.js';
```

Módulos built-in importantes:

- fs: sistema de archivos
- path: manejo de rutas
- http/https: servidores y clientes HTTP
- crypto: criptografía
- os: información del sistema operativo
- events: emisores de eventos
- stream: manejo de streams
- util: utilidades

Ejemplo con fs (File System):

```
const fs = require('fs').promises;
```

```
async function leerArchivo() {
  try {
    const contenido = await fs.readFile('archivo.txt', 'utf-8');
    console.log(contenido);
  } catch (error) {
    console.error('Error leyendo archivo:', error);
  }
}
```

3. Express.js Fundamentos

Express es el framework web más popular para Node.js. Simplifica la creación de servidores y APIs.

Instalación:

```
npm install express
```

Aplicación básica:

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

```
// Middleware para parsear JSON
app.use(express.json());
```

// Ruta GET

```
app.get('/', (req, res) => {
  res.json({ mensaje: 'Hola Mundo' });
});
```

// Ruta con parámetros

```
app.get('/usuarios/:id', (req, res) => {
  const { id } = req.params;
  res.json({ id, nombre: 'Usuario ' + id });
});
```

```
// Ruta POST
app.post('/usuarios', (req, res) => {
  const { nombre, email } = req.body;
  res.status(201).json({ id: 1, nombre, email });
});

// Iniciar servidor
app.listen(3000, () => {
  console.log('API en http://localhost:3000');
});
```

Métodos HTTP comunes:

- GET: obtener recursos
- POST: crear recursos
- PUT: actualizar recurso completo
- PATCH: actualizar parcialmente
- DELETE: eliminar recursos

Query parameters:

```
// GET /buscar?q=javascript&limit=10
app.get('/buscar', (req, res) => {
  const { q, limit } = req.query;
  res.json({ busqueda: q, limite: limit });
});
```

4. Middleware en Express

Los middleware son funciones que tienen acceso al objeto request (req), response (res), y la función next(). Se ejecutan en orden.

Estructura de un middleware:

```
function miMiddleware(req, res, next) {
  // Hacer algo
  console.log('Petición recibida:', req.method, req.url);
  next(); // Pasar al siguiente middleware
}
```

Tipos de middleware:

1. Middleware de aplicación:

```
app.use((req, res, next) => {
  req.requestTime = Date.now();
  next();
});
```

2. Middleware de ruta:

```
app.get('/admin', verificarAdmin, (req, res) => {
  res.json({ admin: true });
});
```

3. Middleware de manejo de errores (4 parámetros):

```
app.use((err, req, res, next) => {
```

```
console.error(err.stack);
res.status(500).json({ error: 'Error interno del servidor' });
});
```

Middleware comunes de terceros:

- cors: habilitar CORS
- helmet: headers de seguridad
- morgan: logging de peticiones
- compression: comprimir respuestas

Ejemplo con CORS y logging:

```
const cors = require('cors');
const morgan = require('morgan');
```

```
app.use(cors());
app.use(morgan('dev'));
app.use(express.json());
```

Middleware de autenticación:

```
function autenticar(req, res, next) {
  const token = req.headers.authorization;
  if (!token) {
    return res.status(401).json({ error: 'No autorizado' });
  }
  try {
    const decoded = jwt.verify(token.replace('Bearer ', ''), SECRET);
    req.usuario = decoded;
    next();
  } catch (error) {
    res.status(401).json({ error: 'Token inválido' });
  }
}
```

5. Estructura de Proyecto y Rutas

Una buena estructura facilita el mantenimiento y escalabilidad.

Estructura recomendada:

```
 proyecto/
   src/
     controllers/ # Lógica de negocio
     routes/      # Definición de rutas
     middleware/ # Middleware personalizado
     models/      # Modelos de datos
     services/    # Servicios externos, lógica reutilizable
     utils/       # Utilidades
     config/      # Configuración
     app.js       # Configuración de Express
     server.js    # Punto de entrada
```

Separar rutas con Router:

```
// routes/usuarios.js
const router = require('express').Router();
```

```

const usuariosController = require('../controllers/usuarios');

router.get('/', usuariosController.listar);
router.get('/:id', usuariosController.obtener);
router.post('/', usuariosController.crear);
router.put('/:id', usuariosController.actualizar);
router.delete('/:id', usuariosController.eliminar);

module.exports = router;

// app.js
const usuariosRoutes = require('./routes/usuarios');
app.use('/api/usuarios', usuariosRoutes);

Controladores:
// controllers/usuarios.js
const Usuario = require('../models/usuario');

exports.listar = async (req, res, next) => {
  try {
    const usuarios = await Usuario.find();
    res.json(usuarios);
  } catch (error) {
    next(error);
  }
};

exports.crear = async (req, res, next) => {
  try {
    const usuario = await Usuario.create(req.body);
    res.status(201).json(usuario);
  } catch (error) {
    next(error);
  }
};

```

6. Manejo de Errores y Validación

El manejo correcto de errores es crítico para APIs robustas.

Errores personalizados:

```

class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
    this.isOperational = true;
  }
}

// Uso
throw new AppError('Usuario no encontrado', 404);

```

Middleware de errores centralizado:

```
app.use((err, req, res, next) => {
  const statusCode = err.statusCode || 500;
  const message = err.isOperational ? err.message : 'Error interno';

  console.error('Error:', err);

  res.status(statusCode).json({
    success: false,
    error: message,
    ...(process.env.NODE_ENV === 'development' && { stack: err.stack })
  });
});
```

Wrapper para async/await:

```
const asyncHandler = (fn) => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};
```

// Uso

```
app.get('/usuarios', asyncHandler(async (req, res) => {
  const usuarios = await Usuario.find();
  res.json(usuarios);
}));
```

Validación con express-validator:

```
const { body, validationResult } = require('express-validator');
```

```
app.post('/usuarios',
  body('email').isEmail().withMessage('Email inválido'),
  body('password').isLength({ min: 6 }).withMessage('Mínimo 6 caracteres'),
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    // Crear usuario...
  }
);
```