**PROCESO DE GESTIÓN DE FORMACIÓN PROFESIONAL INTEGRAL**

**ADSO**

**ANEXO 5 -El Servidor de Nodelia**

## Introducción

¡Bienvenido, Arquitecto de Flujo! En este anexo construirás los cimientos de la comunicación entre los reinos de TechVerse: un servidor. Los servidores son la columna vertebral de Nodelia, permitiendo que los datos fluyan de manera eficiente y segura entre los diferentes sistemas.

# 1. El Servidor Básico de Nodelia

## 1.1 Creando un Servidor HTTP con Node.js Nativo

La forma más básica de crear un servidor web en Node.js es utilizando el módulo `http` integrado:

```
// Importamos el módulo http
const http = require('http');

// Definimos el puerto donde escuchará nuestro servidor
const PUERTO = 3000;

// Creamos el servidor
const servidor = http.createServer((req, res) => {
    // El callback se ejecuta cada vez que llega una petición

    // Configuramos la cabecera de respuesta con el estado y el tipo de contenido
    res.writeHead(200, {'Content-Type': 'text/plain'});

    // Enviamos el cuerpo de la respuesta
    res.end('¡Bienvenido a TechVerse! El servidor está funcionando correctamente.\n');
});
```

```
// Ponemos el servidor a escuchar en el puerto especificado
servidor.listen(PUERTO, () => {
    console.log(`El servidor está escuchando en el puerto ${PUERTO}`);
});
```

Para ejecutar este servidor, guarda el código en un archivo llamado `servidor-basico.js` y ejecútalo con `node servidor-basico.js`. Luego, abre tu navegador y visita `http://localhost:3000`.

## 1.2 Configurando Rutas Básicas

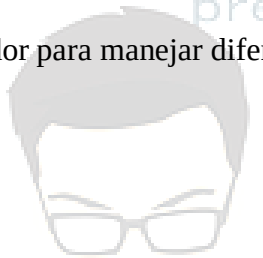
Ampliemos nuestro servidor para manejar diferentes rutas:

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

const PUERTO = 3000;

const servidor = http.createServer((req, res) => {
    // Parseamos la URL para obtener la ruta
    const parsedUrl = url.parse(req.url, true);
    const path = parsedUrl.pathname;

    // Configuramos la cabecera de respuesta para contenido HTML
    res.setHeader('Content-Type', 'text/html');

    // Manejamos diferentes rutas
    if (path === '/' || path === '/inicio') {
        res.writeHead(200);
        res.end('<h1>Bienvenido a TechVerse</h1><p>Estás en la página de inicio</p>');
    } else if (path === '/acerca') {
        res.writeHead(200);
        res.end('<h1>Acerca de TechVerse</h1><p>TechVerse es un multiverso digital donde la tecnología impulsa todos los aspectos de la vida.</p>');
```

```
    } else if (path === '/contacto') {
        res.writeHead(200);
        res.end('<h1>Contacto</h1><p>Envía un mensaje a los Arquitectos de Flujo</p>');
    } else {
        // Si la ruta no coincide con ninguna de las anteriores
        res.writeHead(404);
        res.end('<h1>Error 404: Página no encontrada</h1><p>La ruta que buscas no existe en TechVerse</p>');
    }
});

servidor.listen(PUERTO, () => {
    console.log(`El servidor está escuchando en el puerto ${PUERTO}`);
});
```

### 1.3 Manejando Diferentes Métodos HTTP

Ahora vamos a extender nuestro servidor para manejar diferentes métodos HTTP (GET, POST, etc.):



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

const PUERTO = 3000;

// Simulamos una base de datos de usuarios
let usuarios = [
    { id: 1, nombre: 'Ada Lovelace', rol: 'Arquitecta Pionera' },
    { id: 2, nombre: 'Alan Turing', rol: 'Arquitecto Lógico' }
];

const servidor = http.createServer((req, res) => {
    // Parseamos la URL para obtener la ruta y los parámetros
```

```
const parsedUrl = url.parse(req.url, true);
const path = parsedUrl.pathname;
const query = parsedUrl.query;

// Obtenemos el método de la solicitud
const metodo = req.method.toUpperCase();

// Configuramos la cabecera para respuestas JSON
res.setHeader('Content-Type', 'application/json');

// Ruta para obtener todos los usuarios
if (path === '/api/usuarios' && metodo === 'GET') {
    res.writeHead(200);
    res.end(JSON.stringify(usuarios));
}
// Ruta para obtener un usuario específico
else if (path.match(/^\/api\/usuarios\/\d+$/) && metodo === 'GET') {
    const id = parseInt(path.split('/')[3]);
    const usuario = usuarios.find(u => u.id === id);

    if (usuario) {
        res.writeHead(200);
        res.end(JSON.stringify(usuario));
    } else {
        res.writeHead(404);
        res.end(JSON.stringify({ error: 'Usuario no encontrado' }));
    }
}
// Ruta para crear un nuevo usuario
else if (path === '/api/usuarios' && metodo === 'POST') {
    let datos = '';

    // Recopilamos los datos de la solicitud
```

```
            req.on('data', chunk => {
                datos += chunk.toString();
            });

            // Cuando se han recibido todos los datos
            req.on('end', () => {
                try {
                    const nuevoUsuario = JSON.parse(datos);
                    nuevoUsuario.id = usuarios.length + 1;
                    usuarios.push(nuevoUsuario);
                    res.writeHead(201); // 201 Created
                    res.end(JSON.stringify(nuevoUsuario));
                } catch (error) {
                    res.writeHead(400); // 400 Bad Request
                    res.end(JSON.stringify({ error: 'Datos inválidos' }));
                }
            });
        }
        // Si la ruta no coincide con ninguna de las anteriores
        else {
            res.writeHead(404);
            res.end(JSON.stringify({ error: 'Ruta no encontrada' }));
        }
    });

    servidor.listen(PUERTO, () => {
        console.log(`El servidor API está escuchando en el puerto ${PUERTO}`);
    });
```

### 1.4 Implementando Manejo Básico de Errores

Es importante implementar un buen manejo de errores para que nuestro servidor sea robusto:

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

const PUERTO = 3000;

const servidor = http.createServer((req, res) => {
    try {
        // Simulamos un error aleatorio para demostración
        if (Math.random() > 0.7) {
            throw new Error('Error aleatorio para demostración');
        }

        res.writeHead(200, {'Content-Type': 'text/plain'});
        res.end('Operación exitosa');
    } catch (error) {
        console.error(`Error en la solicitud: ${error.message}`);
        res.writeHead(500, {'Content-Type': 'text/plain'});
        res.end('Error interno del servidor');
    }
});

// Manejamos errores a nivel del servidor
servidor.on('error', (error) => {
    console.error(`Error en el servidor: ${error.message}`);
});

// Manejar cierre inesperado
process.on('uncaughtException', (error) => {
    console.error(`Excepción no capturada: ${error.message}`);
    console.error('Cerrando servidor de forma segura...');
    servidor.close(() => {
        process.exit(1);
    });
```

```
    // Si el servidor no se cierra en 5 segundos, forzamos el cierre
    setTimeout(() => {
        console.error('Cierre forzado');
        process.exit(1);
    }, 5000);
});

servidor.listen(PUERTO, () => {
    console.log(`El servidor está escuchando en el puerto ${PUERTO}`);
});
```

# 2. El Guardián de Express

Express.js es un framework minimalista para Node.js que simplifica la creación de servidores web y APIs.

## 2.1 Migrando a Express.js

Primero, debes instalar Express.js:

```
bash
npm init -y  # Inicializa un proyecto de Node.js si aún no lo has hecho
npm install express
```

Ahora, creemos un servidor básico con Express:

```
// Importamos Express
const express = require('express');

// Creamos una instancia de la aplicación Express
const app = express();

// Definimos el puerto
const PUERTO = 3000;
```

```
// Configuramos una ruta básica
app.get('/', (req, res) => {
    res.send('¡Bienvenido a TechVerse! Esta es una aplicación Express.js');
});

// Iniciamos el servidor
app.listen(PUERTO, () => {
    console.log(`Servidor Express ejecutándose en el puerto ${PUERTO}`);
});
```

## 2.2 Implementando Middlewares

Los middlewares son funciones que tienen acceso al objeto de solicitud (req), al objeto de respuesta (res) y a la siguiente función de middleware en el ciclo de solicitud/respuesta.

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

// Middleware para parsear el cuerpo de las solicitudes JSON
app.use(express.json());

// Middleware para logging
app.use((req, res, next) => {
    const inicio = Date.now();
    console.log(`${req.method} ${req.url} - ${new Date().toISOString()}`);

    // Cuando la respuesta termine, loguear el tiempo de respuesta
    res.on('finish', () => {
        const duracion = Date.now() - inicio;
        console.log(`${req.method} ${req.url} - Completado en ${duracion}ms`);
    });

    // Llamar a next para continuar con el siguiente middleware
```

```
    next();
});

// Middleware de autenticación básica
function autenticarBasico(req, res, next) {
    const authHeader = req.headers.authorization;

    if (!authHeader) {
        return res.status(401).json({ mensaje: 'No se proporcionó autorización' });
    }

    // En un sistema real, verificaríamos credenciales adecuadamente
    // Esto es solo para demostración
    if (authHeader === 'Bearer token-secreto') {
        next(); // Usuario autenticado, continúa
    } else {
        res.status(403).json({ mensaje: 'Acceso denegado' });
    }
}

// Ruta pública
app.get('/', (req, res) => {
    res.send('API pública de TechVerse');
});

// Ruta protegida con autenticación
app.get('/admin', autenticarBasico, (req, res) => {
    res.json({ mensaje: 'Panel de administración', acceso: 'concedido' });
});

// Middleware para manejo de errores (debe ir al final)
app.use((err, req, res, next) => {
```

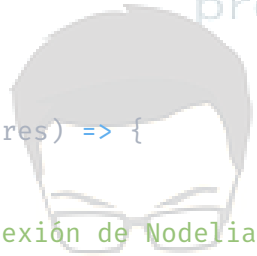
```
    console.error(`Error: ${err.message}`);
    res.status(500).json({ error: 'Error interno del servidor' });
});

// Middleware para rutas no encontradas
app.use((req, res) => {
    res.status(404).json({ error: 'Ruta no encontrada' });
});

app.listen(PUERTO, () => {
    console.log(`Servidor Express con middlewares ejecutándose en el puerto ${PUERTO}`);
});
```

### 2.3 Creando Rutas Organizadas con Router

Express.Router nos permite modularizar nuestras rutas:

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

// Configuraciones
app.use(express.json());

// Creamos un router para las rutas de usuarios
const usuariosRouter = express.Router();

// Rutas de usuarios
usuariosRouter.get('/', (req, res) => {
    res.json([
        { id: 1, nombre: 'Ada Lovelace', rol: 'Arquitecta Pionera' },
        { id: 2, nombre: 'Alan Turing', rol: 'Arquitecto Lógico' }
    ]);
```

```
});

usuariosRouter.get('/:id', (req, res) => {
    const id = parseInt(req.params.id);
    // En un caso real, buscaríamos el usuario en la base de datos
    res.json({ id, nombre: 'Usuario Ejemplo', rol: 'Arquitecto Novato' });
});

usuariosRouter.post('/', (req, res) => {
    const nuevoUsuario = req.body;
    // En un caso real, guardaríamos en la base de datos
    res.status(201).json({ ...nuevoUsuario, id: 3 });
});

// Creamos un router para las rutas de proyectos
const proyectosRouter = express.Router();

// Rutas de proyectos
proyectosRouter.get('/', (req, res) => {
    res.json([
        { id: 1, nombre: 'Reconexión de Nodelia', estado: 'En progreso' },
        { id: 2, nombre: 'Puentes de DataRealm', estado: 'Planificación' }
    ]);
});

proyectosRouter.get('/:id', (req, res) => {
    const id = parseInt(req.params.id);
    res.json({ id, nombre: 'Proyecto Ejemplo', estado: 'Activo' });
});

// Registramos los routers con sus prefijos
app.use('/api/usuarios', usuariosRouter);
app.use('/api/proyectos', proyectosRouter);
```



```
// Ruta principal
app.get('/', (req, res) => {
    res.send(`
        <h1>API de TechVerse</h1>
        <p>Endpoints disponibles:</p>
        <ul>
            <li>GET /api/usuarios</li>
            <li>GET /api/usuarios/:id</li>
            <li>POST /api/usuarios</li>
            <li>GET /api/proyectos</li>
            <li>GET /api/proyectos/:id</li>
        </ul>
    `);
});

app.listen(PUERTO, () => {
    console.log(`Servidor Express con routers ejecutándose en el puerto ${PUERTO}`);
});
```

### 2.4 Manejando Peticiones Asíncronas con Async/Await

Vamos a refactorizar nuestras rutas para utilizar funciones asíncronas:

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

app.use(express.json());

// Simulamos una base de datos con un retraso artificial
const baseDeDatos = {
    usuarios: [
```

```
        { id: 1, nombre: 'Ada Lovelace', rol: 'Arquitecta Pionera' },
        { id: 2, nombre: 'Alan Turing', rol: 'Arquitecto Lógico' }
    ],

    // Métodos asíncronos que simulan operaciones de BD
    async obtenerTodos() {
        // Simulamos un retraso de red/BD
        await new Promise(resolve => setTimeout(resolve, 100));
        return [...this.usuarios];
    },

    async obtenerPorId(id) {
        await new Promise(resolve => setTimeout(resolve, 50));
        return this.usuarios.find(u => u.id === id);
    },

    async crear(usuario) {
        await new Promise(resolve => setTimeout(resolve, 200));
        const nuevoUsuario = { ...usuario, id: this.usuarios.length + 1 };
        this.usuarios.push(nuevoUsuario);
        return nuevoUsuario;
    }
};

// Middleware para capturar errores en rutas async
const asyncHandler = fn => (req, res, next) => {
    Promise.resolve(fn(req, res, next)).catch(next);
};

// Rutas con manejo asíncrono
app.get('/api/usuarios', asyncHandler(async (req, res) => {
    const usuarios = await baseDeDatos.obtenerTodos();
    res.json(usuarios);
```

```
}));

app.get('/api/usuarios/:id', asyncHandler(async (req, res) => {
    const id = parseInt(req.params.id);
    const usuario = await baseDeDatos.obtenerPorId(id);

    if (!usuario) {
        return res.status(404).json({ error: 'Usuario no encontrado' });
    }

    res.json(usuario);
}));

app.post('/api/usuarios', asyncHandler(async (req, res) => {
    const datosUsuario = req.body;

    if (!datosUsuario.nombre || !datosUsuario.rol) {
        return res.status(400).json({ error: 'El nombre y rol son obligatorios'
});
    }

    const nuevoUsuario = await baseDeDatos.crear(datosUsuario);
    res.status(201).json(nuevoUsuario);
}));

// Middleware de manejo de errores global
app.use((err, req, res, next) => {
    console.error(`Error en ${req.method} ${req.path}: ${err.message}`);
    res.status(500).json({ error: 'Error interno del servidor' });
});

app.listen(PUERTO, () => {
```

```
    console.log(`Servidor Express asíncrono ejecutándose en el puerto
${PUERTO}`);
});
```

# 3. Optimizando el Servidor

Para completar este anexo, vamos a optimizar nuestro servidor para manejar múltiples conexiones concurrentes y mejorar su rendimiento.

## 3.1 Compresión de Respuestas

```
const express = require('express');
const compression = require('compression');  // npm install compression

const app = express();
const PUERTO = 3000;

// Aplicamos compresión a todas las respuestas
app.use(compression());

app.use(express.json());

app.get('/api/datos-grandes', (req, res) => {
    // Generamos datos grandes para demostrar la compresión
    const datoGrande = {
        titulo: 'Historia de TechVerse',
        contenido: 'Un largo texto...'.repeat(1000),
        metadata: {
            autor: 'Arquitecto de Flujo',
            fecha: new Date(),
            tags: ['historia', 'techverse', 'nodelia', 'datarealm']
        }
    };
```

```
    res.json(datoGrande);
});

app.listen(PUERTO, () => {
    console.log(`Servidor con compresión ejecutándose en el puerto ${PUERTO}`);
});
```

### 3.2 Implementando Cache de Respuestas

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

// Cache en memoria simple
const cache = new Map();

// Middleware de cache
function middlewareCache(duracionSegundos) {
    return (req, res, next) => {
        const key = req.originalUrl;
        const cachearEstaRuta = req.method === 'GET';  // Solo cacheamos GETs

        // Si la ruta está en cache y no ha expirado, devolvemos la respuesta cacheada
        if (cachearEstaRuta && cache.has(key)) {
            const { datos, expiracion } = cache.get(key);

            if (Date.now() < expiracion) {
                console.log(`Sirviendo desde cache: ${key}`);
                return res.json(datos);
            } else {
                // Expiró, eliminamos del cache
                cache.delete(key);
```

```
        }
    }

    // Sobreescribimos el método res.json para interceptar la respuesta
    const originalJson = res.json;
    res.json = function(datos) {
        if (cachearEstaRuta && res.statusCode === 200) {
            console.log(`Guardando en cache: ${key}`);
            cache.set(key, {
                datos,
                expiracion: Date.now() + (duracionSegundos * 1000)
            });
        }
        originalJson.call(this, datos);
    };

    next();
  };
}

// Aplicamos el middleware de cache a una ruta específica (10 segundos)
app.get('/api/datos', middlewareCache(10), (req, res) => {
    // Simulamos una operación costosa
    console.log('Generando datos frescos...');
    const datos = {
        mensaje: 'Datos de TechVerse',
        timestamp: new Date().toISOString()
    };

    res.json(datos);
});

app.listen(PUERTO, () => {
```

```
    console.log(`Servidor con cache ejecutándose en el puerto ${PUERTO}`);
});
```

### 3.3 Manejo de Conexiones Concurrentes

Node.js está diseñado para manejar eficientemente conexiones concurrentes gracias a su naturaleza no bloqueante. Sin embargo, hay ciertas prácticas que podemos implementar para optimizarlo aún más:

```
const express = require('express');
const cluster = require('cluster');
const os = require('os');

// Número de CPUs disponibles
const numCPUs = os.cpus().length;

// Si es el proceso maestro
if (cluster.isMaster) {
    console.log(`Proceso maestro ${process.pid} está ejecutándose`);

    // Crear un worker por cada CPU disponible
    for (let i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    // Log cuando un worker se desconecta
    cluster.on('exit', (worker, code, signal) => {
        console.log(`Worker ${worker.process.pid} murió`);
        // Reemplazamos el worker muerto
        cluster.fork();
    });
} else {
    // Los workers comparten el mismo puerto
    const app = express();
```

```
const PUERTO = 3000;

app.get('/', (req, res) => {
    res.send(`Hola desde el worker ${process.pid}`);
});

// Ruta que simula una operación intensiva en CPU
app.get('/intensivo', (req, res) => {
    let resultado = 0;
    // Simulamos una operación intensiva
    for (let i = 0; i < 10000000; i++) {
        resultado += i;
    }
    res.json({ resultado, worker: process.pid });
});

app.listen(PUERTO, () => {
    console.log(`Worker ${process.pid} ejecutándose en puerto ${PUERTO}`);
});
}
```

# 4. Ejercicios Prácticos

## Ejercicio 1: Servidor de Archivos Estáticos

Crea un servidor que pueda servir archivos estáticos (HTML, CSS, JS, imágenes) desde una carpeta específica.

## Ejercicio 2: API RESTful Completa

Implementa una API RESTful para gestionar una colección de recursos (por ejemplo, libros, tareas, etc.) con todas las operaciones CRUD utilizando Express.js y almacenamiento en memoria.

## Ejercicio 3: Chat en Tiempo Real

Desarrolla un servidor de chat simple utilizando Socket.io para comunicación en tiempo real entre clientes.

# Conclusión

¡Felicidades, Arquitecto de Flujo! Has aprendido a construir servidores robustos y eficientes con Node.js, sentando las bases para restaurar la comunicación en TechVerse. Estos conocimientos te permitirán:

1. Crear servidores HTTP básicos con Node.js nativo
2. Implementar aplicaciones web completas con Express.js
3. Utilizar middlewares para añadir funcionalidades como autenticación y logging
4. Estructurar tu código utilizando routers
5. Optimizar el rendimiento con técnicas como compresión y caché
6. Manejar múltiples conexiones concurrentes

Recuerda que la comunicación entre los reinos de TechVerse depende de la solidez de tus servidores. ¡Continúa tu viaje y restaura el equilibrio en el multiverso digital!

**Próximo paso**: Explora "Conectando con DataRealm" para aprender a integrar bases de datos con tus servidores y completar la restauración del flujo de información.

# Introducción

Bienvenido, Arquitecto de Flujo, a tu primera misión crítica en la restauración de TechVerse. Este taller te guiará paso a paso para reconstruir el servidor central de Nodelia, estableciendo así el primer punto de comunicación entre los reinos.

# Objetivos de Aprendizaje

- Comprender los fundamentos de los servidores HTTP en Node.js
- Implementar un servidor básico utilizando el módulo HTTP nativo
- Migrar a Express.js y comprender sus ventajas
- Implementar rutas y middlewares en Express
- Manejar errores y optimizar un servidor web

# PARTE 1: NIVEL 1 - EL SERVIDOR BÁSICO

### Ejercicio 1: Creando tu primer servidor HTTP

1. Crea una carpeta llamada `nodelia-server`
2. Inicializa un proyecto Node.js con `npm init -y`
3. Crea un archivo llamado `server.js` con el siguiente código:

```
// Importamos el módulo HTTP nativo de Node.js
const http = require('http');

// Definimos el puerto donde escuchará nuestro servidor
const PORT = 3000;

// Creamos el servidor
const server = http.createServer((req, res) => {
  // Configuramos los headers de la respuesta
  res.setHeader('Content-Type', 'text/plain');

  // Escribimos un mensaje de respuesta
  res.end('¡Bienvenido a Nodelia, Arquitecto de Flujo!');
});

// El servidor comienza a escuchar en el puerto especificado
server.listen(PORT, () => {
  console.log(`Servidor de Nodelia activo en el puerto ${PORT}`);
});
```

4. Ejecuta tu servidor con `node server.js`
5. Abre tu navegador y visita `http://localhost:3000`

## Ejercicio 2: Configurando rutas básicas

Modifica tu `server.js` para manejar diferentes rutas:

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

const PORT = 3000;

const server = http.createServer((req, res) => {
  // Parsear la URL para obtener la ruta
```

```
  const parsedUrl = url.parse(req.url, true);
  const path = parsedUrl.pathname;

  // Configurar headers de respuesta
  res.setHeader('Content-Type', 'text/plain');

  // Manejar diferentes rutas
  if (path === '/') {
    res.statusCode = 200;
    res.end('Bienvenido al Portal Central de Nodelia');
  }
  else if (path === '/status') {
    res.statusCode = 200;
    res.end('Estado del Servidor: Activo');
  }
  else if (path === '/info') {
    res.statusCode = 200;
    res.end('Información del Reino de Nodelia: El núcleo de TechVerse');
  }
  else {
    // Ruta no encontrada
    res.statusCode = 404;
    res.end('Error 404: La ruta solicitada no existe en Nodelia');
  }
});

server.listen(PORT, () => {
  console.log(`Servidor de Nodelia activo en el puerto ${PORT}`);
});
```

### Ejercicio 3: Manejando métodos HTTP

Expande tu servidor para manejar diferentes métodos HTTP:

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

const PORT = 3000;

// Base de datos simulada (en memoria)
const nodeliaDB = {
  resources: [
    { id: 1, name: 'Crystals', quantity: 100 },
    { id: 2, name: 'Code Fragments', quantity: 250 },
    { id: 3, name: 'Energy Cores', quantity: 50 }
  ]
};

const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url, true);
  const path = parsedUrl.pathname;
  const method = req.method;

  // Ruta para manejar recursos
  if (path === '/api/resources') {
    // Configurar header para respuestas JSON
    res.setHeader('Content-Type', 'application/json');

    // GET: Listar todos los recursos
    if (method === 'GET') {
      res.statusCode = 200;
      res.end(JSON.stringify(nodeliaDB.resources));
    }
    // POST: Crear un nuevo recurso (simulado)
    else if (method === 'POST') {
      res.statusCode = 201;
```

```
    res.end(JSON.stringify({ message: 'Recurso creado con éxito' }));
  }
  // Método no soportado
  else {
    res.statusCode = 405;
    res.end(JSON.stringify({ error: 'Método no permitido' }));
  }
}
// Ruta para manejar un recurso específico
else if (path.match(/^\/api\/resources\/[0-9]+$/)) {
  const id = parseInt(path.split('/')[3]);
  const resource = nodeliaDB.resources.find(r => r.id === id);

  res.setHeader('Content-Type', 'application/json');

  // Recurso no encontrado
  if (!resource) {
    res.statusCode = 404;
    res.end(JSON.stringify({ error: 'Recurso no encontrado' }));
    return;
  }

  // GET: Obtener un recurso específico
  if (method === 'GET') {
    res.statusCode = 200;
    res.end(JSON.stringify(resource));
  }
  // PUT: Actualizar un recurso (simulado)
  else if (method === 'PUT') {
    res.statusCode = 200;
    res.end(JSON.stringify({ message: `Recurso ${id} actualizado` }));
  }
  // DELETE: Eliminar un recurso (simulado)
```

```
    else if (method === 'DELETE') {
      res.statusCode = 200;
      res.end(JSON.stringify({ message: `Recurso ${id} eliminado` }));
    }
    // Método no soportado
    else {
      res.statusCode = 405;
      res.end(JSON.stringify({ error: 'Método no permitido' }));
    }
  }
  // Ruta por defecto
  else {
    res.setHeader('Content-Type', 'text/plain');
    res.statusCode = 404;
    res.end('Error 404: La ruta solicitada no existe en Nodelia');
  }
});

server.listen(PORT, () => {
  console.log(`Servidor de Nodelia activo en el puerto ${PORT}`);
});
```

## Ejercicio 4: Manejo básico de errores

Implementa un sistema de manejo de errores en tu servidor:

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

const PORT = 3000;

const server = http.createServer((req, res) => {
  // Envolver toda la lógica del servidor en un try-catch
```


try {
const parsedUrl = url.parse(req.url, true);
const path = parsedUrl.pathname;
// Simulamos un error en una ruta específica
if (path === '/error') {
throw new Error('¡Error simulado en Nodelia!');
}
// Procesamiento normal
res.setHeader('Content-Type', 'text/plain');
res.statusCode = 200;
res.end('Servidor funcionando correctamente');
} catch (error) {
// Manejo centralizado de errores
console.error(`[ERROR]: ${error.message}`);
res.setHeader('Content-Type', 'application/json');
res.statusCode = 500;
res.end(JSON.stringify({
error: 'Error interno del servidor',
message: 'Ha ocurrido un problema en Nodelia'
}));
}
});
// Manejo de errores a nivel de servidor
server.on('error', (error) => {
console.error(`[SERVER ERROR]: ${error.message}`);
// Si el puerto está en uso, intentamos con otro
if (error.code === 'EADDRINUSE') {

```
    console.log(`El puerto ${PORT} está en uso, intentando con el puerto ${PORT + 1}...`);
    setTimeout(() => {
      server.close();
      server.listen(PORT + 1);
    }, 1000);
  }
});

server.listen(PORT, () => {
  console.log(`Servidor de Nodelia activo en el puerto ${PORT}`);
});
```

### Ejercicio 5: Optimización para conexiones concurrentes (BONUS)

Mejora tu servidor para manejar múltiples conexiones:

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

const PORT = 3000;

// Número de núcleos de CPU
const numCPUs = os.cpus().length;

if (cluster.isMaster) {
  console.log(`Maestro ${process.pid} iniciando...`);
  console.log(`Iniciando ${numCPUs} trabajadores...`);

  // Crear un trabajador por cada núcleo disponible
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
```

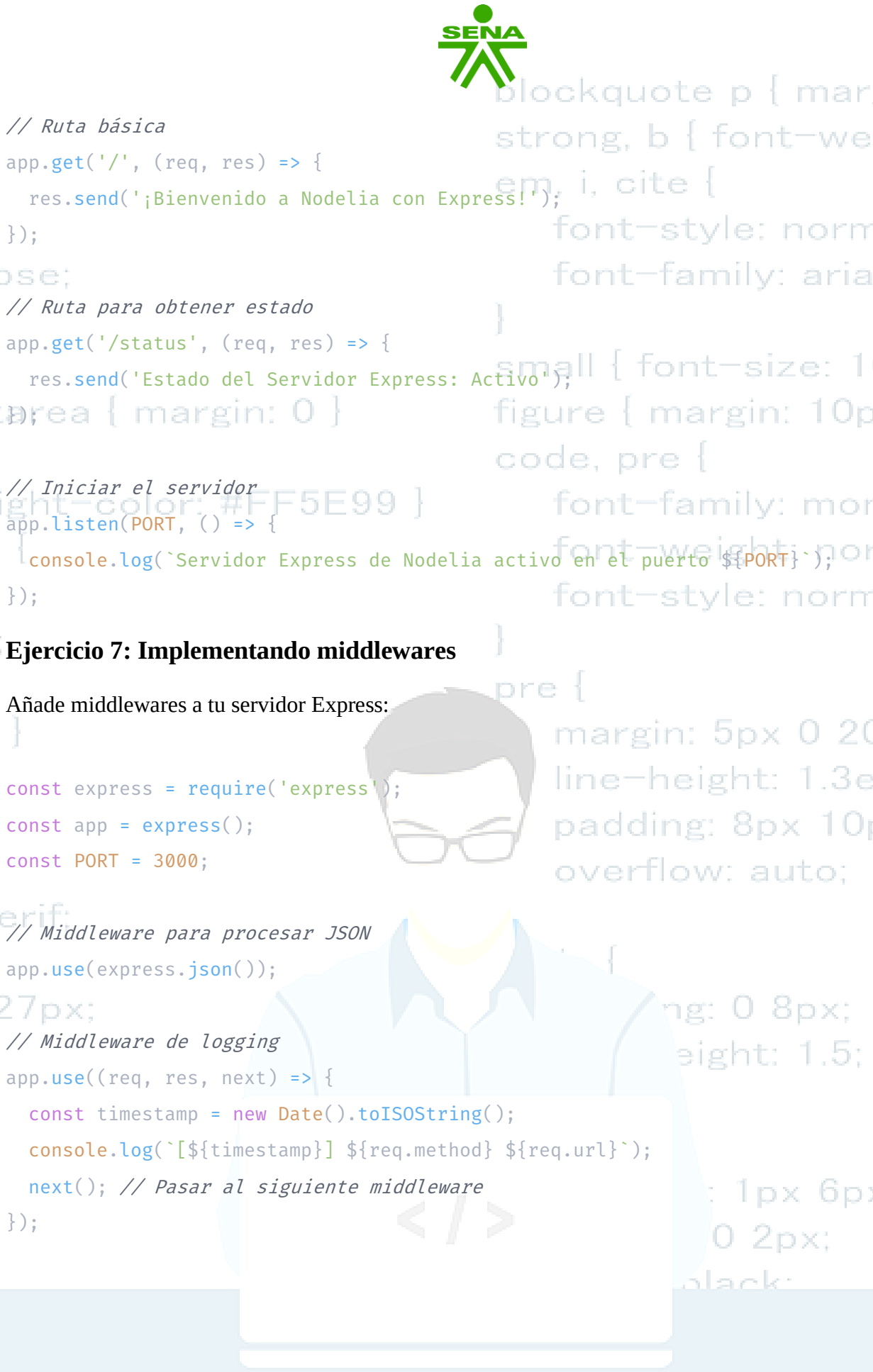
```
  // Si un trabajador muere, creamos uno nuevo
  cluster.on('exit', (worker, code, signal) => {
    console.log(`Trabajador ${worker.process.pid} ha caído. Iniciando nuevo
trabajador...`);
    cluster.fork();
  });
} else {
  // Código del servidor que ejecutará cada trabajador
  const server = http.createServer((req, res) => {
    res.setHeader('Content-Type', 'text/plain');
    res.end(`Servidor de Nodelia respondiendo desde el trabajador
${process.pid}`);
  });

  server.listen(PORT, () => {
    console.log(`Trabajador ${process.pid} escuchando en el puerto ${PORT}`);
  });
}
```

# PARTE 2: NIVEL 2 - EL GUARDIÁN DE EXPRESS

## Ejercicio 6: Migrando a Express.js

1. Instala Express.js: `npm install express`
2. Crea un nuevo archivo llamado `express-server.js`:

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

// Configuración para procesar JSON en las peticiones
app.use(express.json());
```



```
// Ruta básica
app.get('/', (req, res) => {
  res.send('¡Bienvenido a Nodelia con Express!');
});

// Ruta para obtener estado
app.get('/status', (req, res) => {
  res.send('Estado del Servidor Express: Activo');
});

// Iniciar el servidor
app.listen(PORT, () => {
  console.log(`Servidor Express de Nodelia activo en el puerto ${PORT}`);
});
```

## Ejercicio 7: Implementando middlewares

Añade middlewares a tu servidor Express:

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

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

// Middleware de logging
app.use((req, res, next) => {
  const timestamp = new Date().toISOString();
  console.log(`[${timestamp}] ${req.method} ${req.url}`);
  next(); // Pasar al siguiente middleware
});
```

```
// Middleware para medir tiempo de respuesta
app.use((req, res, next) => {
  const start = Date.now();

  // Una vez que la respuesta se envíe, calculamos el tiempo
  res.on('finish', () => {
    const duration = Date.now() - start;
    console.log(`[${req.method}] ${req.url} - ${duration}ms`);
  });

  next();
});

// Middleware de autenticación básica (simulación)
const authenticate = (req, res, next) => {
  const authHeader = req.headers.authorization;

  if (!authHeader) {
    return res.status(401).json({ error: 'No autorizado - Token no proporcionado' });
  }

  // En un caso real, verificaríamos el token
  // Aquí simplemente verificamos que exista "NODELIA" en el header
  if (!authHeader.includes('NODELIA')) {
    return res.status(403).json({ error: 'Prohibido - Token inválido' });
  }

  // Si la autenticación es exitosa, continuamos
  next();
};

// Rutas públicas
```

```
app.get('/', (req, res) => {
  res.send('Portal público de Nodelia');
});

// Rutas protegidas
app.get('/admin', authenticate, (req, res) => {
  res.send('Portal administrativo de Nodelia - Acceso autorizado');
});

// Manejador de errores para rutas no encontradas
app.use((req, res, next) => {
  res.status(404).send('Error 404: Ruta no encontrada en el Reino de Nodelia');
});

// Manejador de errores general
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Error interno en el servidor de Nodelia');
});

app.listen(PORT, () => {
  console.log(`Servidor Express de Nodelia activo en el puerto ${PORT}`);
});
```

### Ejercicio 8: Organizando rutas con Router

Organiza tu API usando Express Router:

1. Crea una carpeta llamada `routes`
2. Crea un archivo `routes/resources.js`:

```
const express = require('express');
const router = express.Router();
```

```
// Base de datos simulada (en memoria)
const resourcesDB = [
  { id: 1, name: 'Crystals', quantity: 100 },
  { id: 2, name: 'Code Fragments', quantity: 250 },
  { id: 3, name: 'Energy Cores', quantity: 50 }
];

// GET: Obtener todos los recursos
router.get('/', (req, res) => {
  res.json(resourcesDB);
});

// GET: Obtener un recurso específico
router.get('/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const resource = resourcesDB.find(r => r.id === id);

  if (!resource) {
    return res.status(404).json({ error: 'Recurso no encontrado' });
  }

  res.json(resource);
});

// POST: Crear un nuevo recurso
router.post('/', (req, res) => {
  const { name, quantity } = req.body;

  // Validación básica
  if (!name || !quantity) {
    return res.status(400).json({ error: 'Se requieren nombre y cantidad' });
  }
```

```
  // Generamos un nuevo ID (en producción usaríamos una mejor estrategia)
  const newId = resourcesDB.length > 0 ? Math.max(...resourcesDB.map(r => r.id))
+ 1 : 1;

  // Creamos el nuevo recurso
  const newResource = {
    id: newId,
    name,
    quantity
  };

  // Lo añadimos a la "base de datos"
  resourcesDB.push(newResource);

  // Respondemos con el recurso creado
  res.status(201).json(newResource);
});

// PUT: Actualizar un recurso
router.put('/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const { name, quantity } = req.body;

  // Buscar el índice del recurso
  const resourceIndex = resourcesDB.findIndex(r => r.id === id);

  if (resourceIndex === -1) {
    return res.status(404).json({ error: 'Recurso no encontrado' });
  }

  // Actualizar el recurso
  resourcesDB[resourceIndex] = {
    ...resourcesDB[resourceIndex],
```

```
    name: name || resourcesDB[resourceIndex].name,
    quantity: quantity !== undefined ? quantity :
resourcesDB[resourceIndex].quantity
  };

  res.json(resourcesDB[resourceIndex]);
});

// DELETE: Eliminar un recurso
router.delete('/:id', (req, res) => {
  const id = parseInt(req.params.id);

  // Buscar el índice del recurso
  const resourceIndex = resourcesDB.findIndex(r => r.id === id);

  if (resourceIndex === -1) {
    return res.status(404).json({ error: 'Recurso no encontrado' });
  }

  // Eliminar el recurso
  const deletedResource = resourcesDB.splice(resourceIndex, 1)[0];

  res.json({ message: `Recurso ${id} eliminado`, resource: deletedResource });
});

module.exports = router;
```

3. Crea un archivo `routes/connections.js`:

```
const express = require('express');
const router = express.Router();

// Base de datos simulada para las conexiones entre reinos
```

```
const connectionsDB = [
  { id: 1, source: 'Nodelia', target: 'Expressland', status: 'active' },
  { id: 2, source: 'Nodelia', target: 'DataRealm', status: 'inactive' },
  { id: 3, source: 'Nodelia', target: 'Webonia', status: 'degraded' }
];

// GET: Obtener todas las conexiones
router.get('/', (req, res) => {
  res.json(connectionsDB);
});

// GET: Obtener una conexión específica
router.get('/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const connection = connectionsDB.find(c => c.id === id);

  if (!connection) {
    return res.status(404).json({ error: 'Conexión no encontrada' });
  }

  res.json(connection);
});

// POST: Establecer una nueva conexión
router.post('/', (req, res) => {
  const { source, target } = req.body;

  // Validación básica
  if (!source || !target) {
    return res.status(400).json({ error: 'Se requieren origen y destino' });
  }

  // Generamos un nuevo ID
```

```
  const newId = connectionsDB.length > 0 ? Math.max(...connectionsDB.map(c =>
c.id)) + 1 : 1;

  // Creamos la nueva conexión
  const newConnection = {
    id: newId,
    source,
    target,
    status: 'pending'
  };

  // La añadimos a la "base de datos"
  connectionsDB.push(newConnection);

  // Respondemos con la conexión creada
  res.status(201).json(newConnection);
});

// PATCH: Actualizar el estado de una conexión
router.patch('/:id/status', (req, res) => {
  const id = parseInt(req.params.id);
  const { status } = req.body;

  // Validación
  if (!status || !['active', 'inactive', 'degraded',
'pending'].includes(status)) {
    return res.status(400).json({
      error: 'Estado inválido',
      validStatus: ['active', 'inactive', 'degraded', 'pending']
    });
  }

  // Buscar la conexión
```

```
  const connectionIndex = connectionsDB.findIndex(c => c.id === id);

  if (connectionIndex === -1) {
    return res.status(404).json({ error: 'Conexión no encontrada' });
  }

  // Actualizar el estado
  connectionsDB[connectionIndex].status = status;

  res.json(connectionsDB[connectionIndex]);
});

module.exports = router;
```

4. Ahora actualiza tu `express-server.js` para usar estos routers:

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

// Importamos los routers
const resourcesRouter = require('./routes/resources');
const connectionsRouter = require('./routes/connections');

// Middlewares
app.use(express.json());

// Middleware de logging
app.use((req, res, next) => {
  const timestamp = new Date().toISOString();
  console.log(`[${timestamp}] ${req.method} ${req.url}`);
  next();
});
```

```
// Rutas básicas
app.get('/', (req, res) => {
  res.send('¡Bienvenido al Portal Central de Nodelia!');
});

// Uso de los routers
app.use('/api/resources', resourcesRouter);
app.use('/api/connections', connectionsRouter);

// Manejador para rutas no encontradas
app.use((req, res, next) => {
  res.status(404).send('Error 404: Ruta no encontrada en el Reino de Nodelia');
});

// Manejador de errores
app.use((err, req, res, next) => {
  console.error(`[ERROR] ${err.stack}`);
  res.status(500).json({
    error: 'Error interno del servidor',
    message: 'Ha ocurrido un problema en Nodelia'
  });
});

// Iniciamos el servidor
app.listen(PORT, () => {
  console.log(`Servidor Express de Nodelia activo en el puerto ${PORT}`);
});
```