INTRODUCCIÓN AL DESARROLLO BACKEND

CON NODEJS

2023

ANDÉN Centro de Innovación y Emprendimientos Tecnológicos

SECRETARÍA DE EXTENSIÓN UNIVERSITARIA UTN - FRC SEU

UTN Facultad Regional Córdoba

Agencia CÓRDOBA JOVEN

CÓRDOBA entre todos

Pruebas y Depuración de una API con Node.js

- Objetivos de la Clase
- ¿Por qué Pruebas y Depuración?
- Tipos de Pruebas
- Herramientas de Pruebas en Node.js
- Pruebas Unitarias con Jest
- Pruebas de Integración con Supertest
- Depuración de Aplicaciones
- Depuración con el Depurador VSCode

Objetivos de la Clase

- Comprender la importancia de las pruebas y la depuración en el desarrollo de una API.
- Aprender a realizar pruebas unitarias y de integración en Node.js.
- Familiarizarse con las herramientas de depuración disponibles en Node.js.

¿Por qué Pruebas y Depuración?

Ejemplo:

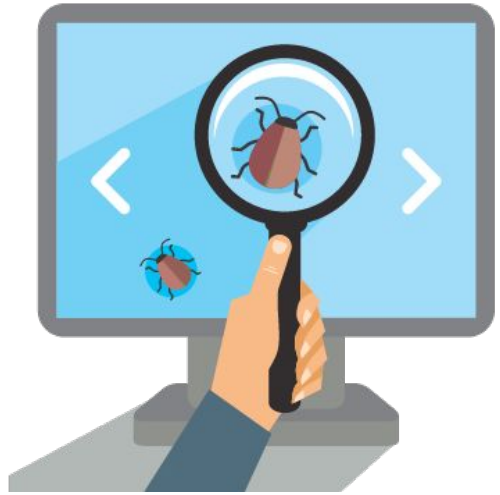
Si nuestra API no se prueba adecuadamente, podríamos enfrentar problemas de seguridad, mal rendimiento o comportamiento incorrecto. La depuración nos permite detectar y solucionar rápidamente errores como fallos en el código o comportamientos inesperados.

- Las pruebas y la depuración son prácticas fundamentales en el desarrollo de software. Aquí algunas razones por las que son importantes:
  - Las pruebas garantizan la calidad y fiabilidad del código al verificar que los componentes funcionen como se espera.
  - La depuración nos ayuda a identificar y solucionar errores en el código, asegurando un funcionamiento correcto.
  - Ambas prácticas son cruciales para desarrollar aplicaciones robustas y eficientes.

Tipos de Pruebas

Existen diferentes tipos de pruebas que se pueden realizar en una API:

- **Pruebas Unitarias:** Se centran en evaluar componentes individuales del código, como funciones o módulos, de manera aislada.
- **Pruebas de Integración:** Verifican la interacción entre diferentes componentes de la API y cómo se comportan juntos.
- **Pruebas de Aceptación:** Validan que la API cumpla con los requisitos y expectativas del cliente.



Ejemplo:

- En una prueba unitaria, podríamos evaluar una función de nuestra API que calcula el precio total de un carrito de compras.
- En una prueba de integración, podríamos verificar cómo interactúan varios módulos de nuestra API para procesar y entregar un pedido completo.

Herramientas de Pruebas en Node.js

Ejemplo:

Podríamos usar **Mocha** y **Chai** para escribir pruebas unitarias para una función de nuestra API y Supertest para realizar pruebas de integración para verificar las respuestas HTTP de nuestra API.

Existen varias herramientas populares para realizar pruebas en Node.js:

- **Mocha:** Un marco de pruebas flexible y potente que proporciona una estructura para organizar y ejecutar pruebas.
- **Chai:** Una biblioteca de aserciones que nos permite realizar afirmaciones claras y concisas en nuestras pruebas.
- **Jest:** Un marco de pruebas con una configuración fácil de usar, sintaxis sencilla y características integradas, como aserciones y cobertura de código.
- **Supertest:** Una biblioteca que facilita la realización de pruebas de integración HTTP para nuestra API.
- **Sinon:** Una biblioteca que nos permite crear espías, falsificaciones y stubs para simular comportamientos en nuestras pruebas.

Jest: Framework de Pruebas para Node.js

Jest es un framework de pruebas popular y potente para proyectos de Node.js. Proporciona una sintaxis sencilla y características integradas que facilitan la escritura y ejecución de pruebas.

Características clave de Jest:

- **Sintaxis sencilla:** Jest ofrece una sintaxis intuitiva y fácil de usar para definir pruebas y aserciones claras.
- **Configuración lista para usar:** Con Jest, puedes comenzar a escribir pruebas rápidamente sin tener que configurar mucho.
- **Aserciones integradas:** Jest proporciona métodos de aserción integrados como expect para realizar afirmaciones y verificar los resultados esperados.
- **Cobertura de código:** Jest puede generar informes de cobertura de código para evaluar cuánto de tu código está cubierto por las pruebas.
- **Ejecución rápida:** Jest utiliza un motor de ejecución paralela para acelerar la ejecución de pruebas y brindar resultados rápidos.

Jest: Ejemplo

```
test('sumar devuelve la suma correcta de dos números', () => {
  const a = 2;
  const b = 3;
  const resultadoEsperado = 5;

  const resultado = sumar(a, b);

  expect(resultado).toBe(resultadoEsperado);
});
```

En este ejemplo, hemos definido una prueba utilizando la función test de Jest.

Estamos verificando que la función sumar devuelve la suma correcta de dos números.

Utilizamos **expect** y el método **toBe** para realizar la aserción y verificar el resultado esperado.

Cómo utilizar Jest en tus proyectos de Node.js

1. **Instalación:** Asegúrate de tener Jest instalado como una dependencia de desarrollo en tu proyecto. Puedes instalar Jest utilizando npm con el siguiente comando:

```
> npm install --save-dev jest
```

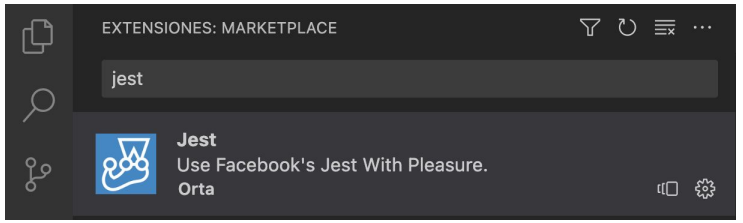
2. **Estructura de archivos:** Crea una estructura de archivos para tus pruebas. Por convención, los archivos de prueba deben tener nombres que terminen en .test.js o .spec.js. Por ejemplo:
- proyecto/
  |- sum.js
  |- sum.test.js
  |- package.json

3. **Escribir pruebas:**
   - En tu archivo de prueba, importa el código que deseas probar.
   - Utiliza la función test de Jest para definir una prueba.
   - Dentro de la prueba, utiliza **expect** y los métodos **toBe** de aserción de Jest para verificar el resultado esperado.

4. **Ejecutar pruebas:** Ejecuta tus pruebas utilizando el comando jest en la terminal. Jest buscará automáticamente los archivos de prueba y ejecutará las pruebas definidas en ellos. Por ejemplo:

```
> jest
```

Extensión VSCode



Instalación Global de Jest (Solo una vez)

```
> npm install -g jest
```

Jest: Cobertura de Pruebas

La cobertura de pruebas es una métrica que mide el grado en que el código fuente de tu aplicación está cubierto por pruebas automatizadas.

Jest, el marco de pruebas para Node.js, proporciona funcionalidades integradas para generar informes de cobertura de código.

Beneficios de la cobertura de pruebas:

- **Identificación de áreas no probadas:** La cobertura de pruebas ayuda a identificar áreas de tu código que no están siendo probadas, lo que te permite tomar medidas para aumentar la cobertura y mejorar la calidad del software.
- **Validación de la calidad del código:** Una alta cobertura de pruebas generalmente indica que tu código ha sido exhaustivamente probado, lo que puede aumentar la confianza en su calidad y robustez.
- **Detección de código inalcanzable:** La cobertura de pruebas puede ayudar a identificar partes del código que son inalcanzables durante la ejecución normal, lo que puede ser un indicador de problemas potenciales o código muerto.

```
jest --coverage
```

Scripts en el archivo package.json

```
{
  "name": "proyecto",
  "version": "1.0.0",
  "scripts": {
    "start": "node app.js",
    "test": "jest",
    "lint": "eslint src",
    "build": "webpack"
  },
  "devDependencies": {
    "jest": "^27.0.6",
    "eslint": "^7.32.0",
    "webpack": "^5.50.0"
  }
}
```

El archivo **package.json** es una parte fundamental de cualquier proyecto de Node.js y contiene información importante sobre el proyecto, incluidas las dependencias y los scripts.

Los **scripts** en el archivo **package.json** nos permiten automatizar tareas comunes y ejecutar comandos específicos del proyecto.

En este ejemplo, tenemos varios scripts definidos en el objeto "scripts":

"start": Ejecuta el archivo index.js utilizando el comando node.
"test": Ejecuta las pruebas utilizando Jest.
"lint": Ejecuta el linter (en este caso, ESLint) en la carpeta src para analizar y verificar el código en busca de errores o inconsistencias.
"build": Ejecuta el empaquetador (en este caso, Webpack) para generar una versión optimizada de la aplicación.

Estos scripts pueden ser ejecutados desde la terminal utilizando el comando **npm seguido del nombre del script**. Por ejemplo, **npm test** ejecutaría las pruebas utilizando Jest.

Desarrollo Basado en Pruebas (Test-Driven Development, TDD)

El Desarrollo Basado en Pruebas (TDD) es una práctica de desarrollo de software en la cual las pruebas unitarias se escriben antes de implementar la funcionalidad.

Principios clave del TDD:

- **Red-Green-Refactor:** El ciclo de trabajo en TDD sigue el patrón "red-green-refactor":
  - Red (Rojo): Escribe una prueba que falle. Esto define el comportamiento esperado de la funcionalidad.
  - Green (Verde): Implementa la funcionalidad mínima necesaria para que la prueba pase.
  - Refactor: Mejora el código sin cambiar su comportamiento, manteniendo las pruebas verdes.
- **Pruebas como guía:** Las pruebas se utilizan como una guía para desarrollar el código. Las pruebas definen los requisitos y comportamiento esperado de la funcionalidad.
- **Incremental y iterativo:** El desarrollo se realiza de manera incremental e iterativa, escribiendo pruebas y código en pequeños pasos, asegurando que cada cambio realizado esté respaldado por pruebas unitarias.

TDD: Beneficios

- **Mejora la calidad del código:** Al escribir pruebas antes de implementar la funcionalidad, se garantiza que el código esté bien estructurado y cumpla con los requisitos.
- **Permite una mejor refactorización:** Las pruebas actúan como una red de seguridad, lo que permite refactorizar y mejorar el código con confianza, asegurando que las pruebas sigan pasando.
- **Fomenta el diseño modular:** Al escribir pruebas antes de la implementación, se fomenta un diseño más modular y desacoplado, lo que facilita el mantenimiento y la extensibilidad del código.

Mockups: Simulando Comportamientos en Pruebas

Los mockups son objetos simulados o falsos que se utilizan en pruebas para representar el comportamiento de componentes externos o dependencias. Estos objetos permiten controlar y predecir el comportamiento durante las pruebas unitarias.

Beneficios de los mockups:

- **Aislamiento de dependencias:** Los mockups permiten aislar el código bajo prueba de sus dependencias reales, asegurando que las pruebas se centren en la lógica específica.
- **Control del comportamiento:** Con los mockups, puedes simular diferentes escenarios y respuestas de las dependencias para probar diferentes caminos de ejecución.
- **Eliminación de dependencias externas:** Al utilizar mockups en lugar de dependencias reales, se evita la necesidad de acceder a recursos externos, como bases de datos o servicios web, durante las pruebas unitarias.

Mockups: Simulando Comportamientos en Pruebas

Los mockups son objetos simulados o falsos que se utilizan en pruebas para representar el comportamiento de componentes externos o dependencias. Estos objetos permiten controlar y predecir el comportamiento durante las pruebas unitarias.

Beneficios de los mockups:

- **Aislamiento de dependencias:** Los mockups permiten aislar el código bajo prueba de sus dependencias reales, asegurando que las pruebas se centren en la lógica específica.
- **Control del comportamiento:** Con los mockups, puedes simular diferentes escenarios y respuestas de las dependencias para probar diferentes caminos de ejecución.
- **Eliminación de dependencias externas:** Al utilizar mockups en lugar de dependencias reales, se evita la necesidad de acceder a recursos externos, como bases de datos o servicios web, durante las pruebas unitarias.

Supertest: Pruebas de Integración Simplificadas

Supertest es una biblioteca popular en Node.js que facilita la realización de pruebas de integración para API y servicios web.

Características de Supertest:

- **Sintaxis sencilla:** Supertest ofrece una sintaxis fácil de usar para realizar solicitudes HTTP a tu API y verificar las respuestas.
- **Interfaz familiar:** Está diseñado para ser utilizado en combinación con marcos de pruebas como Jest o Mocha, por lo que su interfaz es familiar y se integra bien con los flujos de trabajo de pruebas existentes.
- **Simulación de solicitudes:** Permite simular solicitudes HTTP como GET, POST, PUT, DELETE, entre otras, a tus rutas definidas en la API.
- **Verificación de respuestas:** Proporciona métodos para verificar el código de estado, los encabezados y el contenido de las respuestas recibidas de la API.

Ejemplo SuperTest

```
const request = require('supertest');
const app = require('./app');

test('Obtener lista de productos', async () => {
  const response = await request(app).get('/api/productos');

  expect(response.statusCode).toBe(200);
  expect(response.body).toHaveLength(3);
});

test('Crear un nuevo producto', async () => {
  const nuevoProducto = { nombre: 'Producto X', precio: 10 };

  const response = await request(app)
    .post('/api/productos')
    .send(nuevoProducto);

  expect(response.statusCode).toBe(201);
  expect(response.body.nombre).toBe('Producto X');
});
```

- En este ejemplo, utilizamos **Supertest** para realizar solicitudes HTTP a nuestra API.
- Primero, realizamos una solicitud GET para obtener la lista de productos y verificamos el código de estado y la longitud de la respuesta. Luego, realizamos una solicitud POST para crear un nuevo producto y verificamos el código de estado y el nombre del producto creado.
- Supertest simplifica las pruebas de integración al proporcionar una interfaz intuitiva y herramientas útiles para verificar las respuestas de la API.

Depuración en Node.js

- La depuración nos permite identificar y solucionar errores en nuestro código.
- Console.log es una técnica común para mostrar información relevante durante la ejecución.
- El depurador de Node.js nos proporciona herramientas avanzadas para inspeccionar y controlar la ejecución del programa.

Configuración de la Depuración

1. Abrir el proyecto de la API en VSCode.
2. Asegurarse de tener un archivo de configuración de depuración válido (por ejemplo, launch.json).
3. Dentro de la configuración de depuración, especifica el punto de entrada de la aplicación y otras opciones necesarias.
4. Ejemplo de archivo launch.json

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Depurar API",
      "type": "node",
      "request": "launch",
      "program": "${workspaceFolder}/app.js",
      "env": {
        "NODE_ENV": "development"
      }
    }
  ]
}
```

Depuración con Visual Studio Code

- Nos permite establecer puntos de interrupción, inspeccionar variables y seguir la ejecución paso a paso.
- Ideal para depurar nuestras API en Node.js de manera eficiente.

Ejemplo:

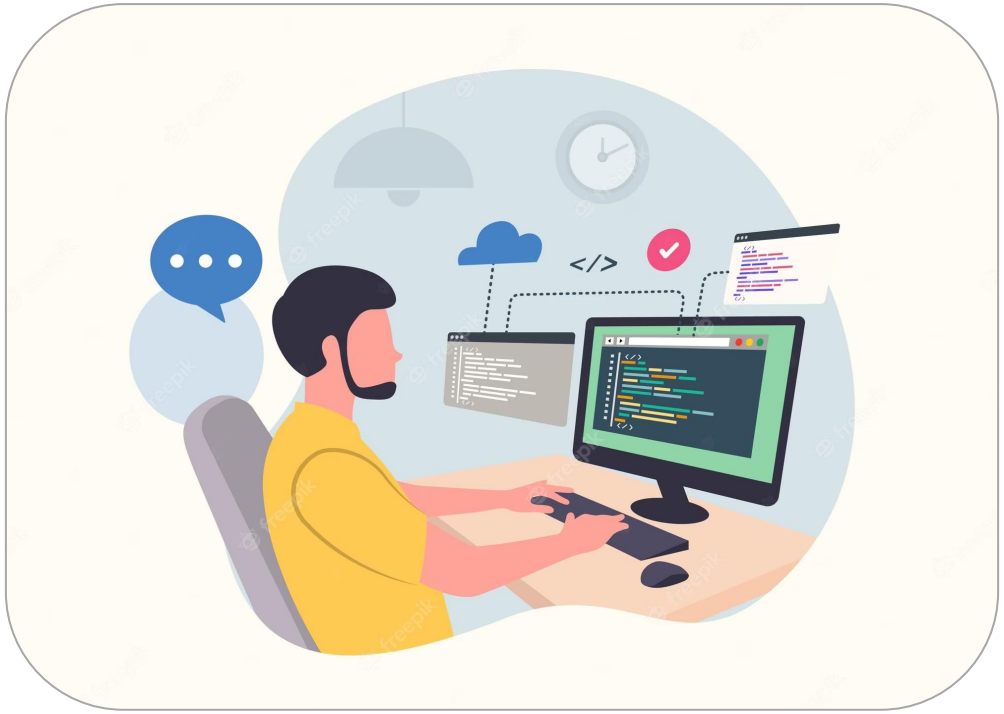
1. Abrir el proyecto de la API en VSCode.
2. Configurar los puntos de interrupción en el código en las áreas relevantes.
3. Iniciar el modo de depuración seleccionando la opción adecuada en la barra lateral o usando el atajo de teclado (por ejemplo, F5).
4. Ejecutar la API en modo de depuración.
5. La ejecución se detendrá en los puntos de interrupción y podrás inspeccionar variables y evaluar expresiones en la pestaña de Depuración.
6. Utiliza los controles de depuración (por ejemplo, continuar, paso a paso, pausa) para avanzar en la ejecución y examinar el comportamiento de la API.

Bueno, Vamo a Codea!!!

1. Prueba Unitaria: con Jest
2. Prueba Unitaria: Mockup con Jest
3. Prueba de Integración: SuperTest
4. Depuración

Actividad 7: Paso a Paso Pruebas Unitarias

- Seguir las instrucciones de la actividad publicada en la UVE.



MUCHAS GRACIAS

ANDÉN Centro de Innovación y Emprendimientos Tecnológicos

SECRETARÍA DE EXTENSIÓN UNIVERSITARIA UTN - FRC SEU

UTN Facultad Regional Córdoba

Agencia CÓRDOBA JOVEN

CÓRDOBA entre todos