

Pruebas unitarias y depuración

En este nuevo documento, describiremos las características que Node.js proporciona para probar y depurar código JavaScript. Comenzaremos demostrando el ejecutor de pruebas (*test runner*) integrado de Node.js, que facilita la definición y ejecución de pruebas unitarias. Continuaremos demostrando el depurador de Node.js, que está integrado en el entorno de ejecución de JavaScript, pero se utiliza a través de herramientas externas. La tabla 1 pone las pruebas y la depuración en contexto. La tabla 2 resume el documento.

Tabla 1: Poner las pruebas y la depuración en contexto.

Pregunta	Respuesta
¿Qué es?	Las pruebas unitarias son el proceso de definir y ejecutar pruebas que verifican el comportamiento del código. La depuración es el proceso de inspeccionar el estado de la aplicación a medida que se ejecuta para determinar la causa de un comportamiento inesperado o indeseable.
¿Por qué es útil?	Las pruebas y la depuración ayudan a identificar problemas en el código antes de que las aplicaciones se implementen para usuarios reales.
¿Cómo se utiliza?	Las pruebas unitarias se escriben en código JavaScript y se ejecutan utilizando el ejecutor de pruebas integrado de Node.js. El entorno de ejecución de Node.js incluye soporte para la depuración, que se utiliza a través de herramientas externas, incluidos los editores de código más populares.
¿Existen dificultades o limitaciones?	Las diferentes opiniones sobre cómo se debe probar el código suelen provocar tensión en los equipos de desarrollo. El esfuerzo que podría dedicarse a completar el proyecto se suele dedicar a discutir sobre las pruebas.
¿Existen alternativas?	Las pruebas y la depuración son actividades opcionales. Ambas pueden ayudar a producir código con menos defectos, pero ninguna es obligatoria.

Tabla 2: Resumen del documento.

Problema	Solución	Listado
Crear una prueba unitaria	Crear un archivo con el sufijo test.js y utilizar la función de prueba definida en el módulo node:test.	3
Ejecutar pruebas unitarias	Utilizar el argumento --test para iniciar Node.js en modo de prueba. Utilizar el argumento --watch para ejecutar pruebas automáticamente cuando se detecte un cambio.	4-8
Crear mocks	Utilizar los métodos fn, method, getter o setter.	9-10
Comprender cómo se utilizan los mocks	Usar las funciones de espionaje agregadas a los mocks.	11
Verificar el resultado de una prueba	Usar las funciones de aserción en el módulo assert.	12

Probar código asíncrono	Crear mocks asíncronos que produzcan datos de prueba.	13-16
Probar diferentes resultados	Utilizar subpruebas.	17
Activar el depurador	Usar la palabra clave debugger o establecer puntos de interrupción en el editor de código.	18-20
Depurar una aplicación	Usar las funciones proporcionadas por los editores de código o navegadores populares.	21-22

Preparación para este documento

En este documento, seguiremos usando el proyecto de aplicación web del documento previo. No se requieren cambios para prepararse para este nuevo documento. Abre un símbolo del sistema, navega hasta la carpeta de la aplicación web y ejecuta el comando que se muestra en el listado 1 para iniciar las herramientas de compilación.

Listado 1: Inicio de las herramientas de compilación.

```
npm start
```

Abre un navegador web, solicita `http://localhost:5000`, ingresa un mensaje en el campo de texto y haz clic en el botón Enviar mensaje. El código JavaScript del lado del cliente enviará el contenido del elemento de entrada al servidor backend, que lo enviará de regreso al navegador en la respuesta, como se muestra en la figura 1.

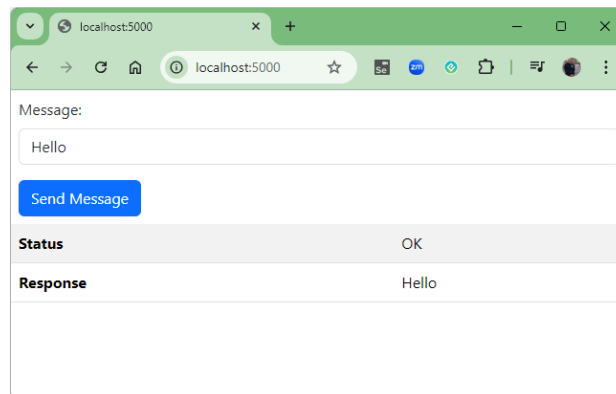


Figura 1: Ejecución de la aplicación de ejemplo.

Pruebas unitarias de aplicaciones Node.js

Node.js tiene un ejecutor de pruebas integrado, que es una forma conveniente de definir y ejecutar pruebas unitarias. Por mucho que recomendamos TypeScript para el desarrollo, las pruebas unitarias se escriben mejor en JavaScript puro. Las pruebas unitarias requieren un

uso extensivo de objetos falsos (conocidos como *mocks*) para aislar el código que se está probando del resto de la aplicación, y el proceso de creación de mocks (conocido como *mocking*) se basa en la creación de objetos que tengan la funcionalidad suficiente para ejecutar una prueba, lo que puede alterar el compilador TypeScript.

Para prepararse para las pruebas unitarias de JavaScript puro, el listado 2 cambia la configuración del compilador TypeScript.

Decidir si se deben realizar pruebas unitarias

Poder realizar pruebas unitarias fácilmente es uno de los beneficios de usar Node.js, pero no es para todos y no tenemos la intención de fingir lo contrario. En lo personal me gustan las pruebas unitarias y las suelo enseñar en mis cursos, pero no en todos, y no de manera tan constante como se podría esperar. Tiendo a concentrarme en escribir pruebas unitarias para características y funciones que sé que serán difíciles de escribir y probablemente serán la fuente de errores en la implementación. En estas situaciones, las pruebas unitarias ayudan a estructurar mis pensamientos sobre cómo implementar mejor lo que necesito. Creo que el solo hecho de pensar en lo que necesito probar me ayuda a generar ideas sobre problemas potenciales, y eso es antes de empezar a lidiar con errores y defectos reales.

Dicho esto, las pruebas unitarias son una herramienta y no una religión, y solo tu sabes cuántas pruebas necesitas. Si no te resultan útiles las pruebas unitarias o tienes una metodología diferente que se adapta mejor a ti, entonces no sientas que necesitas hacer pruebas unitarias solo porque están de moda. (Sin embargo, si no tienes una mejor metodología y no estás haciendo pruebas en absoluto, entonces probablemente estés dejando que los usuarios encuentren sus errores, lo que rara vez es ideal.

No tienes que hacer pruebas unitarias, pero realmente deberías considerar hacer algún tipo de prueba).

Si nunca has probado las pruebas unitarias antes, te animo a que las pruebes para ver cómo funcionan. Si no eres un fanático de las pruebas unitarias, puedes omitir esta sección y pasar a la sección depuración de código JavaScript, donde demostramos cómo usar las funciones de depuración de Node.js.

Listado 2: Adición de propiedades en el archivo tsconfig.json en la carpeta webapp.

```
{
  "extends": "@tsconfig/node20/tsconfig.json",
  "compilerOptions": {
    "rootDir": "src",
    "outDir": "dist",
    "allowJs": true
  },
  "include": ["src/**/*"]
}
```

Las nuevas propiedades de configuración le indican al compilador TypeScript que procese tanto archivos JavaScript como TypeScript y especifican que todos los archivos de código fuente están en la carpeta src.

Para comenzar con las pruebas, agrega un archivo llamado `readHandler.test.js` a la carpeta src con el contenido que se muestra en el listado 3.

Listado 3: El contenido del archivo `readHandler.test.js` en la carpeta src.

```
import { test } from "node:test";

test("my first test", () => {
  // do nothing - test will pass
});
```

La funcionalidad de prueba se proporciona en el módulo `node:test`, y la función más importante es `test`, que se utiliza para definir una prueba unitaria. La función `test` acepta un nombre para la prueba y una función, que se ejecuta para realizar la prueba.

Las pruebas se pueden ejecutar desde la línea de comandos. Abre un nuevo símbolo del sistema y ejecuta el comando que se muestra en el listado 4 en la carpeta `webapp`.

Listado 4: Ejecución de pruebas unitarias.

```
node --test dist
```

El argumento `--test` ejecuta el ejecutor de pruebas de Node.js. Los archivos de prueba se descubren automáticamente, ya sea porque el nombre del archivo contiene `test` o porque los archivos están en una carpeta llamada `test`. Seguimos la convención común de definir las pruebas para un módulo en un archivo que comparte el nombre del módulo pero con el sufijo `.test.js`.

El compilador TypeScript procesará el archivo JavaScript en la carpeta `src` y generará un archivo en la carpeta `dist` que contiene el código de prueba. El ejecutor de pruebas producirá el siguiente resultado, que puede incluir caracteres adicionales, como marcas de verificación, según su plataforma y línea de comandos (ver la figura en la siguiente página):

La prueba no hace nada todavía, pero el resultado muestra que el ejecutor de pruebas encontró el archivo y ejecutó la función que contiene.

El ejecutor de pruebas también se puede ejecutar en modo de observación, donde ejecutará pruebas automáticamente cuando haya un cambio de archivo. El listado 5 agrega un nuevo comando a la sección de scripts del archivo `package.json`.

```
PS C:\proyectosnode\webapp> node --test dist
✓my first test (1.1648ms)
|
| tests 1
| suites 0
| pass 1
| fail 0
| cancelled 0
| skipped 0
| todo 0
| duration_ms 108.9553
PS C:\proyectosnode\webapp> |
```

Uso de un paquete de pruebas

Hemos utilizado el ejecutor de pruebas integrado de Node.js en este documento porque es fácil de usar y hace todo lo que la mayoría de los proyectos requieren. Pero hay buenos paquetes de pruebas de código abierto disponibles; el más popular es Jest (<https://jestjs.io>). Un paquete de pruebas puede ser útil si tienes necesidades de pruebas especializadas o deseas utilizar el mismo paquete para probar el código JavaScript del lado del cliente y del servidor en tus proyectos.

Listado 5: Agregar un comando en el archivo package.json en la carpeta webapp.

```
...
"scripts": {
  "server": "tsc-watch --noClear --onSuccess \"node dist/server.js\"",
  "client": "webpack serve",
  "start": "npm-run-all --parallel server client",
  "test": "node --test --watch dist"
},
...
```

El argumento `--watch` pone al ejecutor de pruebas en modo de observación. Ejecuta el comando que se muestra en el listado 6 en la carpeta webapp para iniciar el comando definido en el listado 5.

Listado 6: Ejecución del ejecutor de pruebas en modo de observación.

```
npm run test
```

El ejecutor de pruebas se iniciará, descubrirá el archivo de prueba en el archivo `dist` y ejecutará la prueba que contiene, lo que producirá el siguiente resultado:

El listado 7 cambia el nombre dado a la prueba para confirmar que el modo de observación de pruebas está funcionando.

```

> webapp@1.0.0 test
> node --test --watch dist

✓my first test (1.1605ms)
  i tests 1
  i suites 0
  i pass 1
  i fail 0
  i cancelled 0
  i skipped 0
  i todo 0
  i duration_ms 137.5135

```

Listado 7: Cambiar el nombre en el archivo readHandler.test.js en la carpeta src.

```

import { test } from "node:test";

test("my new test name", () => {
  // do nothing - test will pass
});

```

El proceso de compilación principal detectará el cambio en el archivo JavaScript en la carpeta src y creará un archivo correspondiente en la carpeta dist. El ejecutor de pruebas de Node.js detectará el cambio en el archivo JavaScript puro y ejecutará su contenido, lo que producirá el siguiente resultado:

```

✓my first test (1.1605ms)
  i tests 1
  i suites 0
  i pass 1
  i fail 0
  i cancelled 0
  i skipped 0
  i todo 0
  i duration_ms 137.5135
✓my new test name (1.0658ms)
  i tests 1
  i suites 0
  i pass 1
  i fail 0
  i cancelled 0
  i skipped 0
  i todo 0
  i duration_ms 137.5135

```

El ejecutor de pruebas de Node.js considera que las pruebas han pasado si se completan sin generar una excepción, por lo que la prueba pasa, aunque no haga nada. El listado 8 modifica la prueba de muestra para que falle.

Listado 8: Creación de una prueba fallida en el archivo readHandler.test.js en la carpeta src.

```

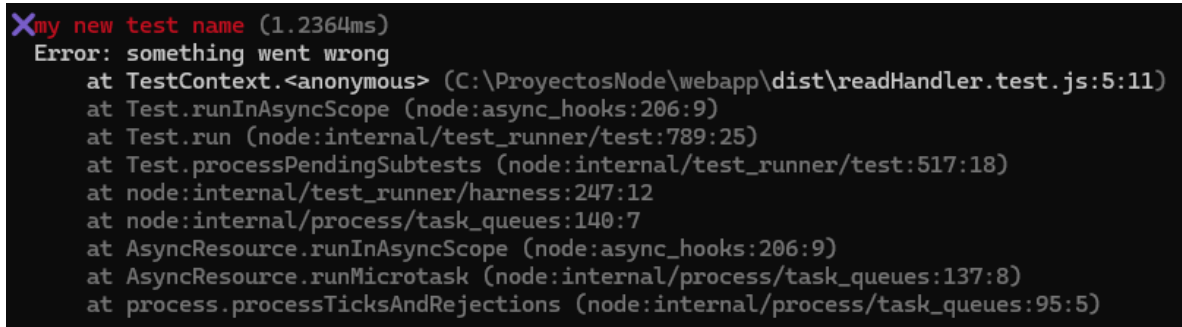
import { test } from "node:test";

test("my new test name", () => {
  throw new Error("something went wrong");
});

```

```
});
```

Cuando el ejecutor de pruebas ejecuta la prueba, se genera la excepción y el error se muestra en la salida de la consola, junto con algunos detalles sobre la excepción:



```
X my new test name (1.2364ms)
Error: something went wrong
    at TestContext.<anonymous> (C:\ProyectosNode\webapp\dist\readHandler.test.js:5:11)
    at Test.runInAsyncScope (node:async_hooks:206:9)
    at Test.run (node:internal/test_runner/test:789:25)
    at Test.processPendingSubtests (node:internal/test_runner/test:517:18)
    at node:internal/test_runner/harness:247:12
    at node:internal/process/task_queues:140:7
    at AsyncResource.runInAsyncScope (node:async_hooks:206:9)
    at AsyncResource.runMicrotask (node:internal/process/task_queues:137:8)
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
```

Escritura de pruebas unitarias

Un enfoque común para escribir pruebas unitarias es seguir el patrón organizar/actuar/afirmar (**arrange/act/assert**, A/A/A), que divide las pruebas unitarias en tres partes. Organizar se refiere a establecer las condiciones para la prueba, actuar se refiere a realizar la prueba y afirmar se refiere a verificar que el resultado fue el esperado.

Organización de una prueba

Para las aplicaciones web, la sección organizar de una prueba unitaria generalmente significa simular una solicitud y una respuesta HTTP para poder probar un controlador de solicitud. Como recordatorio, aquí está el `readHandler` del proyecto de ejemplo:

```
import { Request, Response } from "express";

export const readHandler = (req: Request, resp: Response) => {
  resp.cookie("sessionID", "mysecretcode");
  req.pipe(resp);
}
```

Este controlador hace dos cosas: establece una cookie e invoca el método `Request.pipe` para que el cuerpo de la respuesta se lea desde el cuerpo de la solicitud. Para probar esta funcionalidad, la prueba unitaria necesita una solicitud simulada (*mock Request*) que tenga un método `pipe` y una respuesta (*Response*) que tenga un método `cookie`. La prueba unitaria no necesita recrear la funcionalidad real de los métodos `pipe` y `cookie` porque estos están fuera del alcance del código que se está probando. El listado 9 utiliza las características proporcionadas por Node.js para crear objetos simulados.

Listado 9: Creación de objetos HTTP simulados en el archivo `readHandler.test.ts` en la carpeta `src`.

```
import { test } from "node:test";

test("readHandler tests", (testCtx) => {

  // Arrange - set up the test
  const req = {
    pipe: testCtx.mock.fn()
  };

  const resp = {
    cookie: testCtx.mock.fn()
  };

  // TODO - Act - perform the test

  // TODO - Assert - verify the results
});
```

Un buen objeto simulado contiene la funcionalidad suficiente para ejecutar la prueba, pero también debe admitir la inspección del resultado. Cuando el ejecutor de pruebas de Node.js invoca la función de prueba, proporciona un objeto `TestContext`, cuya propiedad simulada devuelve un objeto `MockTracker` que se puede usar para crear mocks, y cuyos métodos más útiles se describen en la tabla 3.

Tabla 3: Métodos útiles de `MockTracker`.

Nombre	Descripción
<code>fn(orig, impl)</code>	Este método crea una función simulada (mock). Los argumentos opcionales son la implementación original de la función y una nueva implementación. Si se omiten los argumentos, se devuelve una función sin operación.
<code>method(obj, name, impl, opts)</code>	Este método crea un método simulado. Los argumentos son un objeto y el nombre del método para simular. El argumento opcional es una implementación de reemplazo del método.
<code>getter(obj, name, impl, opts)</code>	Similar al método pero crea un getter.
<code>setter(obj, name, impl, opts)</code>	Similar al método pero crea un setter.

Los métodos descritos en la tabla 3 se utilizan para crear funciones o métodos que realizan un seguimiento de cómo se utilizan, lo que resulta útil en la parte de aserción de la prueba, que se describe en la sección afirmación de los resultados de la prueba.

Los métodos `method`, `getter` y `setter` pueden crear contenedores (*wrappers*) alrededor de la funcionalidad existente, como se demuestra en la sección prueba de código asíncrono. Es difícil encapsular los métodos y propiedades de solicitud y respuesta HTTP debido a la forma en que se crean y su dependencia de gran parte de la API de Node.js.

En cambio, el método `fn` se puede utilizar para crear una función que realiza un seguimiento de cómo se utiliza y proporciona un bloque de construcción simple para crear las características necesarias para probar el controlador. Las funciones de JavaScript pueden aceptar cualquier cantidad de argumentos, que es la forma en que la función devuelta por el método `fn` se puede utilizar en cualquier lugar. Esta es una de las razones por las que escribir pruebas en TypeScript puede ser tan difícil y por la que se debe utilizar JavaScript puro.

Realización de una prueba

En el caso de las pruebas unitarias en controladores (*handlers*) HTTP, la realización de la prueba suele ser la parte más sencilla del proceso, ya que implica invocar la función del controlador con los objetos de solicitud y respuesta HTTP simulados, como se muestra en el Listado 10.

Listado 10: Realización de la prueba en el archivo `readHandler.test.js` en la carpeta `src`.

```
import { test } from "node:test";
import { readHandler } from "../readHandler";

test("readHandler tests", (testCtx) => {

  // Arrange - set up the test
  const req = {
    pipe: testCtx.mock.fn()
  };

  const resp = {
    cookie: testCtx.mock.fn()
  };

  // Act - perform the test
  readHandler(req, resp);

  // TODO - Assert - verify the results
});
```

Afirmación de los resultados de la prueba

Los métodos de la tabla 3 producen resultados que tienen una propiedad simulada que se puede utilizar para conocer cómo se utilizó una función o un método cuando se realizó la prueba. La propiedad simulada devuelve un objeto `MockFunctionContext`, cuyas características más útiles se describen en la tabla 4.

Tabla 4: Características útiles de `MockFunctionContext`.

Nombre	Descripción
<code>callCount()</code>	Este método devuelve la cantidad de veces que se ha llamado a la función o al método.
<code>calls</code>	Este método devuelve un arreglo de objetos, donde cada elemento describe una llamada.

El resultado de la propiedad `calls` descrita en la tabla 4 contiene objetos con las propiedades descritas en la tabla 5.

Tabla 5: Propiedades útiles utilizadas para describir una llamada a un método o una función.

Nombre	Descripción
<code>arguments</code>	Esta propiedad devuelve un arreglo de argumentos que se pasaron a la función o al método.
<code>result</code>	Esta propiedad devuelve el resultado producido por la función o el método.
<code>error</code>	Esta propiedad devuelve un objeto si la función genera un error y <code>undefined</code> si no lo genera.
<code>stack</code>	Esta propiedad devuelve un objeto <code>Error</code> que se puede utilizar para determinar dónde se generó un error.

Las funciones y métodos simulados actúan como espías que informan sobre cómo se usaron durante la prueba, lo que permite inspeccionar y evaluar fácilmente el resultado, como se muestra en el listado 11.

Listado 11: Evaluación de los resultados del resto en el archivo `readHandler.test.js` en la carpeta `src`.

```
import { test } from "node:test";
import { readHandler } from "../readHandler";

test("readHandler tests", (testCtx) => {

  // Arrange - set up the test
  const req = {
    pipe: testCtx.mock.fn()
  };
});
```

```

const resp = {
  cookie: testCtx.mock.fn()
};

// Act - perform the test
readHandler(req, resp);

// Assert - verify the results
if (req.pipe.mock.callCount() !== 1 ||
    req.pipe.mock.calls[0].arguments[0] !== resp) {
  throw new Error("Request not piped");
}
if (resp.cookie.mock.callCount() === 1) {
  const [name, val] = resp.cookie.mock.calls[0].arguments;
  if (name !== "sessionID" || val !== "mysecretcode") {
    throw new Error("Cookie not set correctly");
  }
} else {
  throw new Error("cookie method not called once");
}
});

```

Las nuevas declaraciones usan las propiedades simuladas para confirmar que los métodos de canalización y cookie se han llamado una vez y han recibido los argumentos correctos.

La evaluación de los resultados de la prueba se puede simplificar mediante el uso de afirmaciones (*assertions*), que son métodos que realizan comparaciones y lanzan excepciones de manera más concisa. Node.js proporciona afirmaciones en el módulo `assert` y los métodos más útiles se describen en la tabla 6.

Tabla 6: Afirmaciones (*assertions*) útiles.

Nombre	Descripción
<code>assert(val)</code>	Este método genera un error si <code>val</code> no es verdadero
<code>equal(v1, v2)</code>	Este método genera un error si <code>v1</code> no es igual a <code>v2</code>
<code>notEqual(v1, v2)</code>	Este método genera un error si <code>v1</code> es igual a <code>v2</code> .
<code>deepStrictEqual(v1, v2)</code>	Este método realiza una comparación profunda de <code>v1</code> y <code>v2</code> y genera un error si no coinciden
<code>notDeepStrictEqual(v1, v2)</code>	Este método realiza una comparación profunda de <code>v1</code> y <code>v2</code> y genera un error si coinciden.
<code>match(str, regexp)</code>	Este método genera un error si <code>str</code> no coincide con la expresión regular especificada
<code>doesNotMatch(str, regexp)</code>	Este método genera un error si <code>str</code> coincide con la expresión regular especificada.

El listado 12 revisa la prueba unitaria para usar las afirmaciones para verificar los resultados.

Listado 12: Uso de afirmaciones en el archivo `readHandler.test.js` en la carpeta `src`.

```
import { test } from "node:test";
import { readHandler } from "../readHandler";
import { equal } from "assert";

test("readHandler tests", (testCtx) => {

  // Arrange - set up the test
  const req = {
    pipe: testCtx.mock.fn()
  };

  const resp = {
    cookie: testCtx.mock.fn()
  };

  // Act - perform the test
  readHandler(req, resp);

  // Assert - verify the results
  equal(req.pipe.mock.callCount(), 1);
  equal(req.pipe.mock.calls[0].arguments[0], resp);
  equal(resp.cookie.mock.callCount(), 1);
  equal(resp.cookie.mock.calls[0].arguments[0], "sessionID");
  equal(resp.cookie.mock.calls[0].arguments[1], "mysecretcode");
});
```

El método `equal` se utiliza para realizar una serie de comparaciones y arrojará un error que hará que la prueba falle si los valores no coinciden.

Prueba de código asíncrono

El ejecutor de pruebas de Node.js tiene soporte para probar código asíncrono. Para el código basado en promesas, la prueba falla si se rechaza la promesa. Para prepararte, el listado 13 cambia el controlador para que realices una lectura de archivo asíncrona y envíes el contenido del archivo al cliente.

Listado 13: Realización de una operación asíncrona en el archivo `readHandler.ts` en la carpeta `src`.

```
import { Request, Response } from "express";
import { readFile } from "fs";
```

```

export const readHandler = (req: Request, resp: Response) => {
  readFile("data.json", (err, data) => {
    if (err !== null) {
      resp.writeHead(500, err.message);
    } else {
      resp.setHeader("Content-Type", "application/json")
      resp.write(data);
    }
    resp.end();
  });
}

```

Ignore la salida del ejecutor de pruebas por el momento y verifica que el controlador funciona utilizando un navegador para solicitar `http://localhost:5000` y hacer clic en el botón Enviar mensaje. La respuesta contendrá los datos JSON leídos del archivo `data.json`, como se muestra en la figura 2.

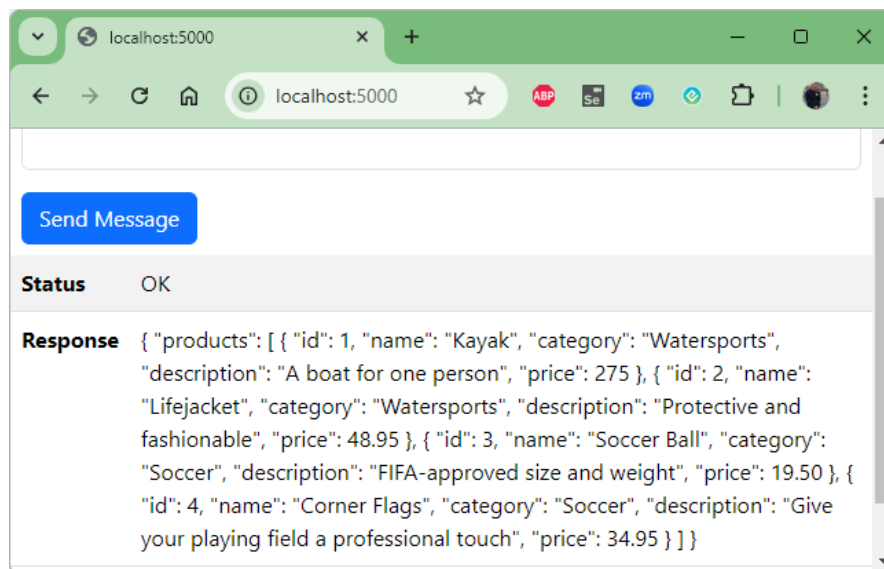


Figura 2: Prueba del controlador revisado.

Para escribir la prueba unitaria, se requiere un enfoque diferente para la simulación, como se muestra en el listado 14.

Listado 14: Prueba de un controlador asíncrono en el archivo `readHandler.test.js` en la carpeta `src`.

```

import { test } from "node:test";
import { readHandler } from "../readHandler";
import { equal } from "assert";

```

```
import fs from "fs";

test("readHandler tests", async (testCtx) => {

  // Arrange - set up the test
  const data = "json-data";
  testCtx.mock.method(fs, "readFile", (file, cb) => cb(undefined, data));
  const req = { };

  const resp = {
    setHeader: testCtx.mock.fn(),
    write: testCtx.mock.fn(),
    end: testCtx.mock.fn()
  };

  // Act - perform the test
  await readHandler(req, resp);

  // Assert - verify the results
  equal(resp.setHeader.mock.calls[0].arguments[0], "Content-Type");
  equal(resp.setHeader.mock.calls[0].arguments[1], "application/json");
  equal(resp.write.mock.calls[0].arguments[0], data);
  equal(resp.end.mock.callCount(), 1);
});
```

La clave de esta prueba es poder simular la función `readFile` en el módulo `fs`, lo que se hace con esta declaración:

```
...
testCtx.mock.method(fs, "readFile", (file, cb) => cb(undefined, data));
...
```

Esto es difícil de explicar porque el nombre y el resultado usan la misma palabra: el método llamado *method* simula (*mocks*) un método en un objeto. En este caso, el objeto es el módulo `fs` completo, que se importó de esta manera:

```
...
import fs from "fs";
...
```

Las funciones de nivel superior definidas por el módulo se presentan como métodos en un objeto llamado `fs`, lo que permite simularlas usando *method*. En este caso, la función `readFile` se ha reemplazado con una implementación simulada que invoca la función de devolución de llamada con datos de prueba, lo que hace posible realizar la prueba sin tener que leer desde el sistema de archivos. Las otras simulaciones en este ejemplo se crean con el método `fn` y

corresponden a los métodos de respuesta que son llamados por el controlador que se está probando.

Probar promesas

Probar código que usa promesas se hace de la misma manera, excepto que la simulación resuelve la promesa con datos de prueba. El listado 15 actualiza el controlador para usar la versión basada en promesas de la función `readFile`.

Listado 15: Uso de promesas en el archivo `readHandler.ts` en la carpeta `src`.

```
import { Request, Response } from "express";
import { readFile } from "fs/promises";

export const readHandler = async (req: Request, resp: Response) => {
  try {
    resp.setHeader("Content-Type", "application/json");
    resp.write(await readFile("data.json"));
  } catch (err) {
    resp.writeHead(500);
  }
  resp.end();
}
```

El listado 16 actualiza la prueba unitaria para que la prueba simulada resuelva una promesa.

Listado 16: Prueba de una promesa en el archivo `readHandler.test.js` en la carpeta `src`.

```
import { test } from "node:test";
import { readHandler } from "../readHandler";
import { equal } from "assert";
import fs from "fs/promises";

test("readHandler tests", async (testCtx) => {

  // Arrange - set up the test
  const data = "json-data";
  testCtx.mock.method(fs, "readFile", async () => data);
  const req = {};

  const resp = {
    setHeader: testCtx.mock.fn(),
    write: testCtx.mock.fn(),
    end: testCtx.mock.fn()
  };
}
```

```

    // Act - perform the test
    await readHandler(req, resp);

    // Assert - verify the results
    equal(resp.setHeader.mock.calls[0].arguments[0], "Content-Type");
    equal(resp.setHeader.mock.calls[0].arguments[1], "application/json");
    equal(resp.write.mock.calls[0].arguments[0], data);
    equal(resp.end.mock.callCount(), 1);
  });

```

La prueba simulada es una función asíncrona que produce los datos de prueba cuando se resuelve. El resto de la prueba unitaria no se modifica.

Creación de subpruebas

La prueba del listado 16 no prueba cómo responde el controlador cuando hay un problema al leer los datos del archivo. Se requiere un poco más de trabajo, como se muestra en el listado 17.

Listado 17: Prueba de múltiples resultados en el archivo `readHandler.test.js` en la carpeta `src`.

```

import { test } from "node:test";
import { readHandler } from "../readHandler";
import { equal } from "assert";
import fs from "fs/promises";

const createMockResponse = (testCtx) => ({
  writeHead: testCtx.mock.fn(),
  setHeader: testCtx.mock.fn(),
  write: testCtx.mock.fn(),
  end: testCtx.mock.fn()
});

test("readHandler tests", async (testCtx) => {

  // Arrange - set up the test
  const req = {};

  //const resp = {
  //  setHeader: testCtx.mock.fn(),
  //  write: testCtx.mock.fn(),
  //  end: testCtx.mock.fn()
  //};

```



```

    // Test the successful outcome
    await testCtx.test("Successfully reads file", async (innerCtx) => {

        // Arrange - set up the test
        const data = "json-data";
        innerCtx.mock.method(fs, "readFile", async () => data);
        const resp = createMockResponse(innerCtx);

        // Act - perform the test
        await readHandler(req, resp);

        // Assert - verify the results
        equal(resp.setHeader.mock.calls[0].arguments[0], "Content-Type");
        equal(resp.setHeader.mock.calls[0].arguments[1], "application/json");
        equal(resp.write.mock.calls[0].arguments[0], data);
        equal(resp.end.mock.callCount(), 1);
    });

    // Test the failure outcome
    await testCtx.test("Handles error reading file", async (innerCtx) => {

        // Arrange - set up the test
        innerCtx.mock.method(fs, "readFile",
            () => Promise.reject("file error"));
        const resp = createMockResponse(innerCtx);

        // Act - perform the test
        await readHandler(req, resp);

        // Assert - verify the results
        equal(resp.writeHead.mock.calls[0].arguments[0], 500);
        equal(resp.end.mock.callCount(), 1);
    });
})

```

La clase `TestContext` define un método de prueba que se puede utilizar para crear subpruebas. Las subpruebas reciben su propio objeto de contexto que se puede utilizar para crear pruebas simuladas específicas para esa subprueba y el listado 17 utiliza esta función para crear pruebas que utilizan diferentes implementaciones para la función `readFile` simulada.

Guarda los cambios y la salida del ejecutor de pruebas reflejará la adición de las subpruebas, de esta manera:

```
► readHandler tests
  ✓Successfully reads file (1.7016ms)
  ✓Handles error reading file (0.4723ms)
► readHandler tests (3.5171ms)
i tests 3
i suites 0
i pass 3
i fail 0
i cancelled 0
i skipped 0
i todo 0
i duration_ms 80.1705
```

Observa que las subpruebas son asíncronas y requieren la palabra clave `await`. Si no esperas las subpruebas, la prueba de nivel superior se completará antes y el ejecutor de pruebas informará un error.

Depuración de código javascript

La prueba unitaria es el proceso de confirmar que el código se comporta como debería; la depuración es el proceso de averiguar por qué no lo hace. Antes de comenzar, usa `Ctrl + C` para detener el proceso de compilación y el proceso de prueba unitaria. Una vez que los procesos se hayan detenido, ejecuta el comando que se muestra en el listado 18 para iniciar el servidor de desarrollo webpack por sí solo. El depurador se aplicará al servidor backend, que se iniciará por sí solo, pero depende de webpack para manejar las solicitudes de contenido del lado del cliente.

Listado 18: Inicio del servidor de desarrollo webpack.

```
npm run client
```

El siguiente paso es configurar el compilador TypeScript para que genere mapas de origen, como se muestra en el listado 19, que permite al depurador correlacionar el código JavaScript puro que ejecuta Node.js con el código TypeScript escrito por el desarrollador.

Listado 19: Habilidad de mapas de origen en el archivo `tsconfig.json` en la carpeta `src`.

```
{
  "extends": "@tsconfig/node20/tsconfig.json",
  "compilerOptions": {
    "rootDir": "src",
    "outDir": "dist",
    "allowJs": true,
    "sourceMap": true
  },
}
```

```
"include": ["src/**/*"]
}
```

Cuando guardes el archivo, el compilador comenzará a generar archivos con la extensión de archivo de mapa (*map*) en la carpeta dist.

Adición de puntos de interrupción de código

Los editores de código que tienen un buen soporte de TypeScript, como Visual Studio Code, permiten agregar puntos de interrupción a los archivos de código. La experiencia con esta función ha sido variada y sea encontrado que no son confiables, por lo que confiaremos en la palabra clave debugger de JavaScript, menos elegante pero más predecible.

Cuando se ejecuta una aplicación JavaScript a través de un depurador, la ejecución se detiene cuando se encuentra la palabra clave debugger y el control se transfiere al desarrollador. El listado 20 agrega la palabra clave debugger al archivo readHandler.ts.

Listado 20: Agregando la palabra clave debugger en el archivo readHandler.ts en la carpeta src.

```
import { Request, Response } from "express";
import { readFile } from "fs/promises";

export const readHandler = async (req: Request, resp: Response) => {
  try {
    resp.setHeader("Content-Type", "application/json")
    resp.write(await readFile("data.json"));
    debugger
  } catch (err) {
    resp.writeHead(500);
  }
  resp.end();
}
```

No habrá cambios en la salida cuando se ejecuta el código porque Node.js ignora la palabra clave debugger de manera predeterminada.

Uso de Visual Studio Code para depurar

La mayoría de los buenos editores de código tienen cierto grado de compatibilidad con la depuración de código TypeScript y JavaScript. En esta sección, te mostraremos cómo realizar la depuración con Visual Studio Code para darte una idea del proceso. Es posible que se

requieran diferentes pasos si utilizas otro editor, pero es probable que el enfoque básico sea similar.

Para establecer la configuración para la depuración, selecciona Agregar configuración en el menú Ejecutar y selecciona Node.js en la lista de entornos cuando se te solicite, como se muestra en la figura 3.

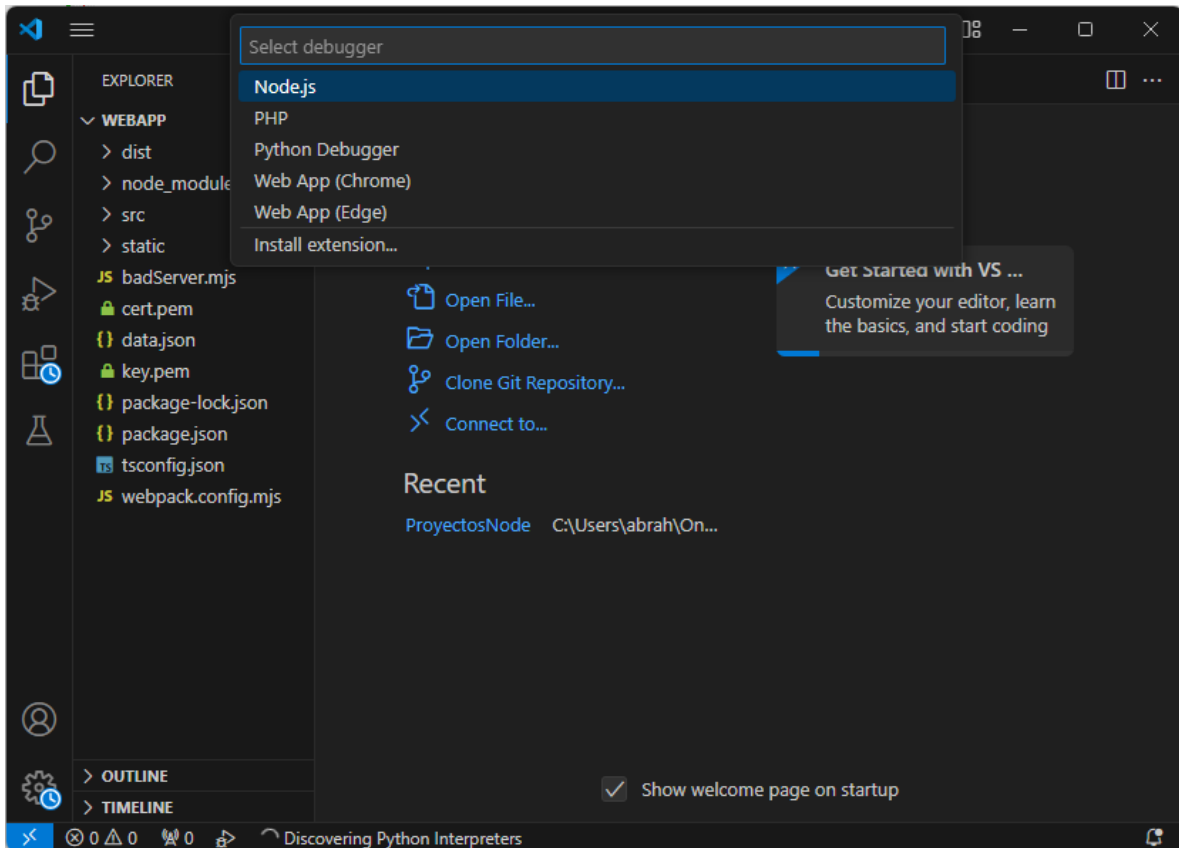


Figura 3: Selección del entorno de depuración.

Visual Studio Code creará una carpeta `.vscode` y un archivo llamado `launch.json`, que se utiliza para configurar el depurador. Cambia el valor de la propiedad `program` para que el depurador ejecute el código JavaScript en la carpeta `dist`, como se muestra en el listado 21.

Listado 21: Configuración del depurador en el archivo `launch.json` en la carpeta `.vscode`.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
```

```

    "skipFiles": [
      "<node_internals>/**"
    ],
    "program": "${workspaceFolder}/dist/server.js",
    "preLaunchTask": "tsc: build - tsconfig.json",
    "outFiles": [
      "${workspaceFolder}/dist/**/*.js"
    ]
  }
}
}

```

Guarda los cambios en el archivo launch.json y selecciona Iniciar depuración en el menú Ejecutar (run). Visual Studio Code iniciará Node.js y la ejecución continuará de forma normal hasta que se alcance la palabra clave debugger. Utiliza un navegador para solicitar `http://localhost:5000` y haz clic en el botón Enviar mensaje. La solicitud se pasará al controlador para su procesamiento y, cuando se alcance la palabra clave debugger, se detendrá la ejecución y el control se transferirá a la ventana emergente de depuración, como se muestra en la figura 4 (será en el archivo `readHandler.ts`).

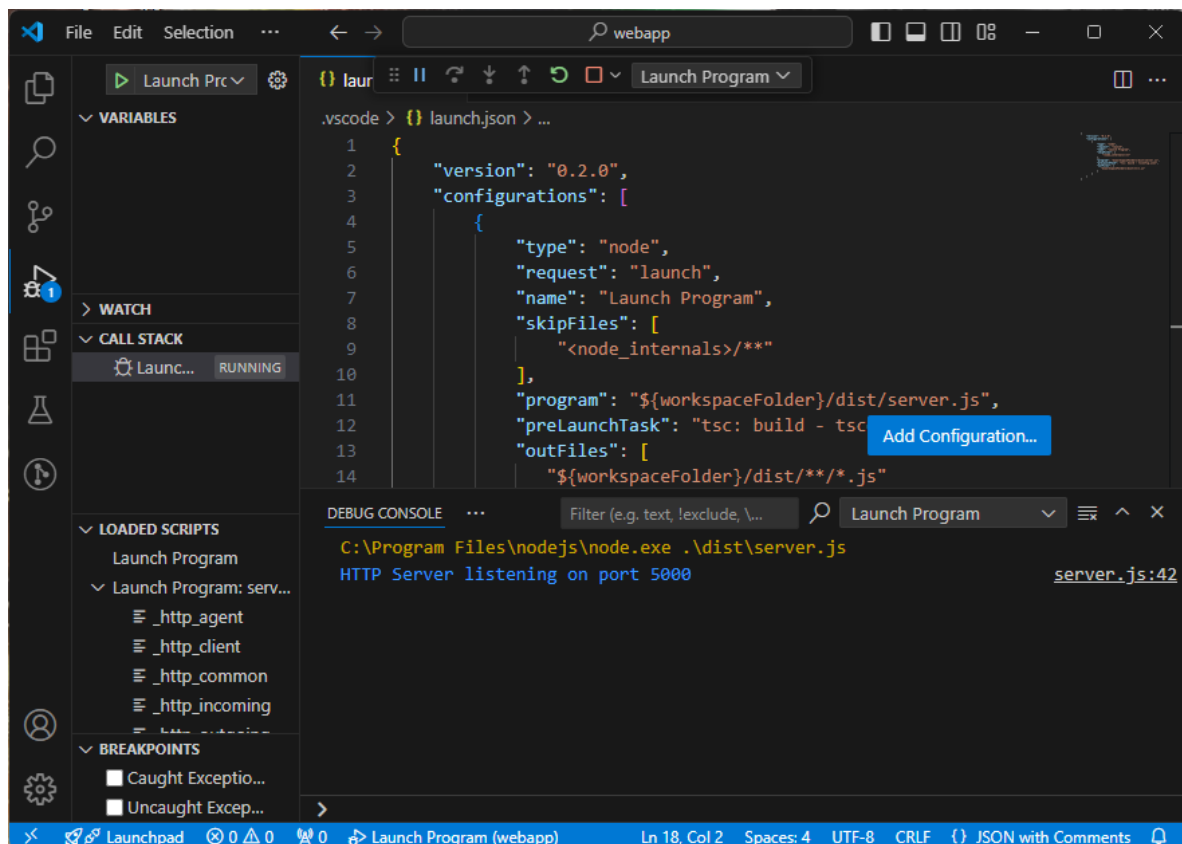


Figura 4: Depuración con Visual Studio Code.

El estado de la aplicación se muestra en la barra lateral, que muestra las variables que se establecieron en el punto en que se detuvo la ejecución. Hay disponibles funciones de depuración estándar, que incluyen la configuración de relojes, la ejecución de instrucciones y la reanudación de la ejecución.

La ventana de la consola de depuración permite ejecutar instrucciones de JavaScript en el contexto de la aplicación, de modo que, al ingresar un nombre de variable y presionar Retorno, por ejemplo, se devolverá el valor asignado a esa variable.

Uso del depurador remoto de Node.js

Si no deseas utilizar el editor de código para la depuración, Google Chrome ofrece una buena depuración integrada para Node.js, que utiliza las mismas funciones que se utilizan para depurar el código del lado del cliente.

Detén el depurador de Visual Studio Code de la sección anterior y ejecuta el comando que se muestra en el listado 22 en la carpeta webapp para iniciar Node.js en modo de depuración.

Listado 22: Inicio de Node.js en modo de depuración.

```
node --inspect dist/server.js
```

Cuando se inicia Node.js, generará mensajes como estos, que incluyen detalles de la URL en la que está listo para aceptar solicitudes de depuración:

```
Debugger listening on ws://127.0.0.1:9229/27d54598-f709-4f6d-a2c8-589bbd235f70
For help, see: https://nodejs.org/en/docs/inspector
HTTP Server listening on port 5000
Debugger attached.
```

Con Google Chrome, solicita `chrome://inspect` y haz clic en la opción Abrir herramientas de desarrollo dedicadas para Node y se abrirá la ventana de depuración, como se muestra en la figura 5.

Nota

Todos los navegadores que utilizan el motor Chromium admiten esta función, incluidos Brave, Opera y Edge. Utiliza el nombre del navegador para la URL que abre las herramientas de Node.js, como `brave://inspect` para el navegador Brave. Esto no funciona para Firefox, que tiene su propio motor de navegación.

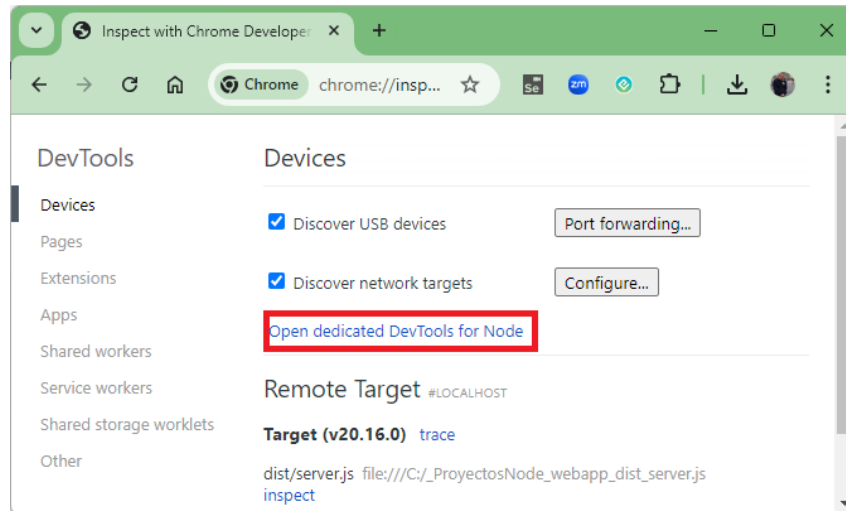


Figura 5: Uso de las funciones de depuración de Node.js en Chrome.

Abre una nueva ventana del navegador, solicita `http://localhost:5000` y haz clic en Enviar mensaje. Mientras se procesa la solicitud, Node.js llega a la palabra clave debugger. La ejecución se detiene y el control pasa a las herramientas para desarrolladores de Chrome, como se muestra en la figura 6.

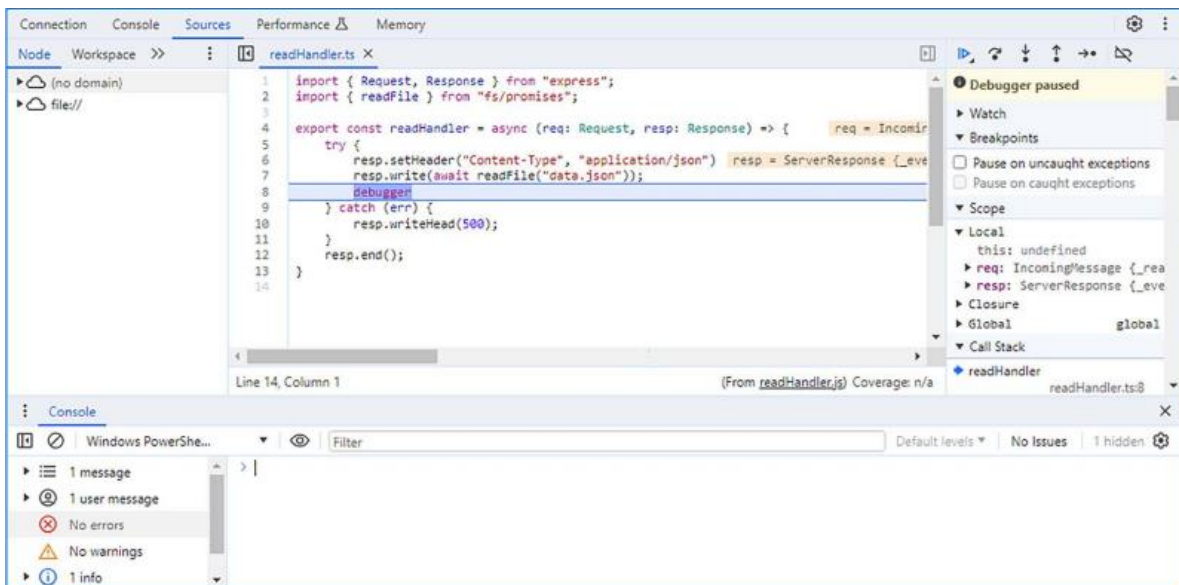


Figura 6: Las herramientas para desarrolladores de Chrome depuran Node.js.

Resumen

En este documento, describimos las funciones de Node.js para pruebas unitarias y depuración.

- Node.js incluye un ejecutor de pruebas integrado, con soporte para ejecutar pruebas y crear funciones y métodos simulados.
- Las pruebas unitarias para aplicaciones web se centran en el manejo de solicitudes y requieren simulacros de solicitudes y respuestas HTTP.
- Se pueden usar paquetes de terceros, como Jest, para proyectos que requieren las mismas herramientas de prueba para el código JavaScript del lado del cliente y del servidor.
- Node.js incluye soporte para depuración, que se puede realizar con muchos editores de código o con uno de los navegadores basados en Chromium, como Google Chrome.

En la siguiente parte de este curso, se mostrará cómo se puede utilizar Node.js para crear las funciones necesarias para las aplicaciones web, como generar contenido dinámico y autenticar usuarios.