

These are instructions for debugging a node project (in most cases).

Starting the Debugger

Once you have a node project that you can start, then you can attach the debugger.

1. Open git bash (or terminal) in the folder of your node project.
2. You'll type the node command and the main file to execute, but using the `--inspect` flag. This will start the node process as well as the debugger.

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

3. You should see the debugger start up in the console and your app start (if it's a server, it should start listening on the correct port).
4. Open chrome and go to `chrome://inspect`

`chrome://inspect`

You should see a list under remote targets of all of your apps running the debugger. Just click on the 'inspect' line for any you want to debug.

Devices

☒ Discover USB devices

Port forwarding...


☒ Discover network targets

Configure...

[Open dedicated DevTools for Node](#)

Remote Target #LOCALHOST

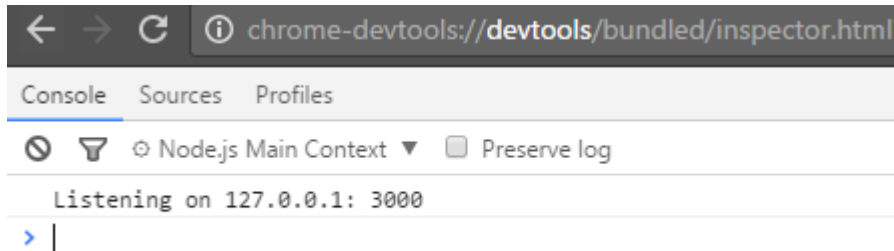
Target

 ./src/app.js file:///C:/_Users_CVandDeMark_Desktop_Courses_RMIA:
inspect

DEBUGGING NODE

5. You should get the chrome debug console. Here you can use the normal console.

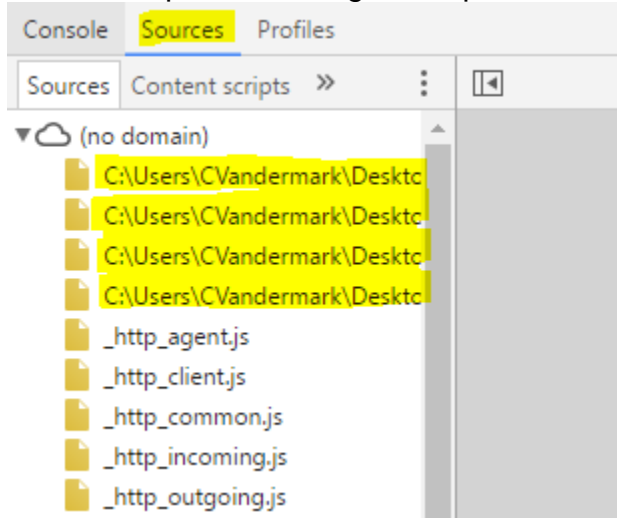
Console



Debugger

If you go to sources, you will get the normal chrome debugger but for all the files running in the modules. Your files will be at the top.

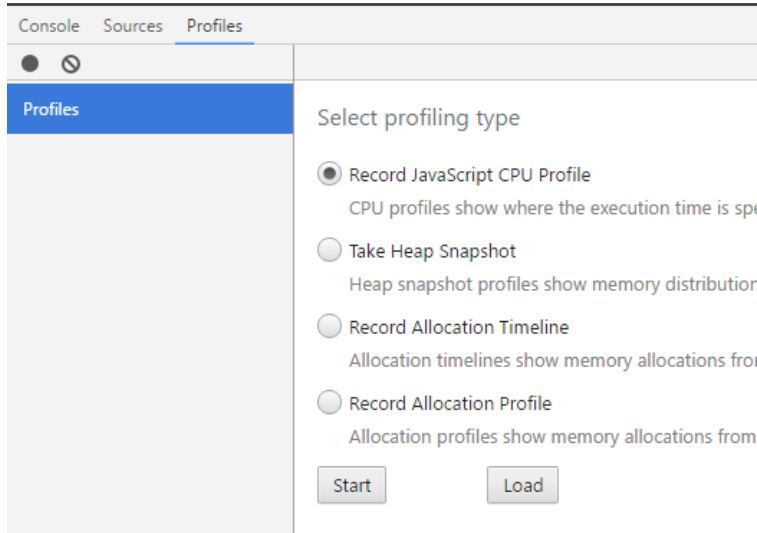
Note: You may need to restart the debugger command after setting a breakpoint, if the breakpoint is during start up and memory allocation.



DEBUGGING NODE

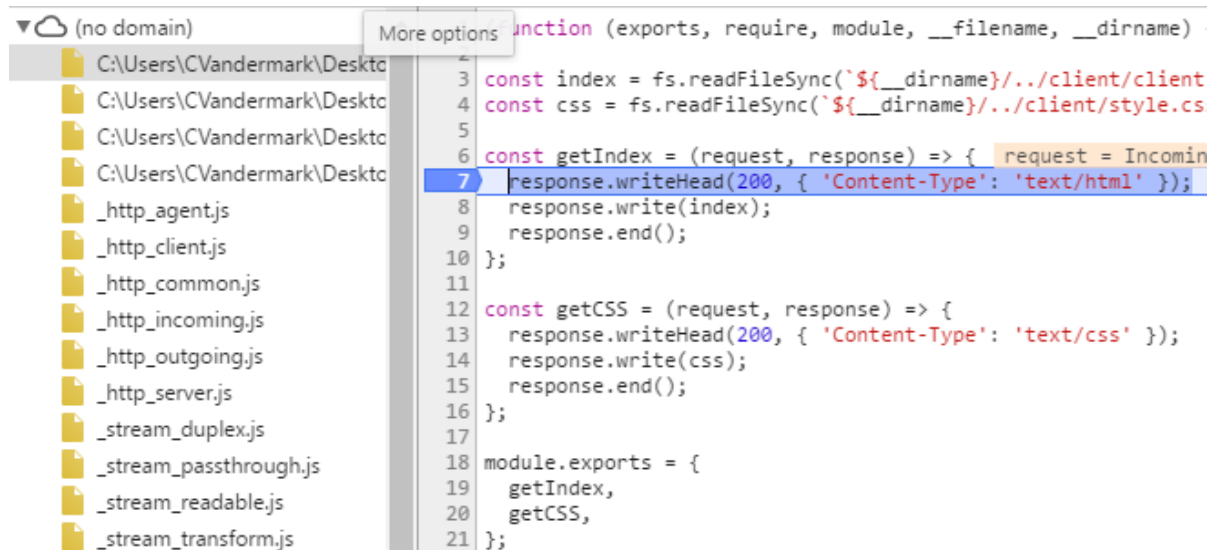
Profiler

You also have access to the profiler if you need it.



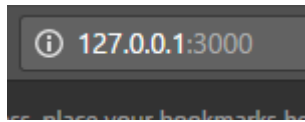
Starting the Debugger

1. Set a breakpoint where you want to debug by clicking the line number of that code.



DEBUGGING NODE

2. Test your app (in the case of a web server like this by going to the page)



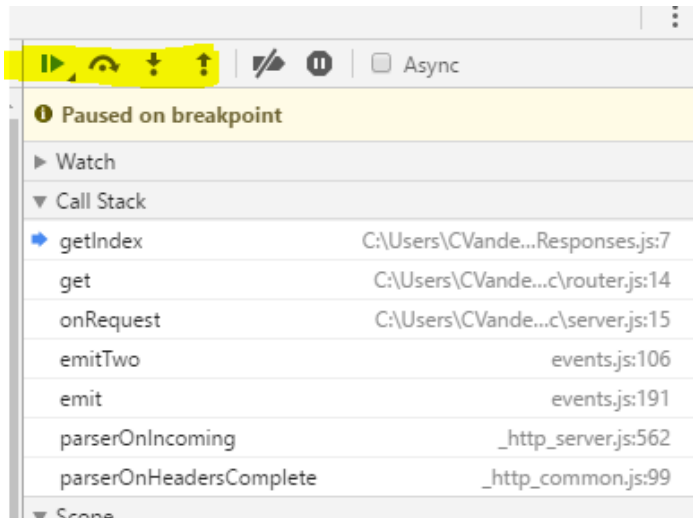
3. Debugger should pause on the breakpoint and allow you to check/change variables.

```
1 (function (exports, require, module, __filename, __dirname) { const fs
in)2
3 const index = fs.readFileSync(`${__dirname}/../client/client.html`);
4 const css = fs.readFileSync(`${__dirname}/../client/style.css`);
5
6 const getIndex = (request, response) => { request = IncomingMessage {
7   response.writeHead(200, { 'Content-Type': 'text/html' });
8   response.write(index);
9   response.end();
10 };
11
12 const getCSS = (request, response) => {
13   response.writeHead(200, { 'Content-Type': 'text/css' });
14   response.write(css);
15   response.end();
16 };
```

A screenshot of the Node.js debugger interface. At the top, there is a toolbar with icons for running, stepping, and other debugging actions, along with an 'Async' checkbox. Below the toolbar, a yellow banner reads 'Paused on breakpoint'. Underneath, there are sections for 'Watch' and 'Call Stack'. The 'Call Stack' section is expanded, showing a list of function calls: 'getIndex' (C:\Users\CVande...Responses.js:7), 'get' (C:\Users\CVande...c\router.js:14), 'onRequest' (C:\Users\CVande...c\server.js:15), 'emitTwo' (events.js:106), 'emit' (events.js:191), 'parserOnIncoming' (_http_server.js:562), and 'parserOnHeadersComplete' (_http_common.js:99). Below the call stack is the 'Scope' section, which is also expanded. It shows 'Local' variables: 'request' (IncomingMessage), 'response' (ServerResponse), and 'this' (Object). It also shows 'Closure' and 'Global' scopes. The 'Global' scope shows the 'global' object. At the bottom, there is a 'Breakpoints' section.

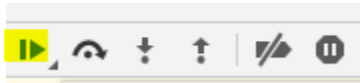
DEBUGGING NODE

4. Use the controls for moving through code.



Play/Pause

The play/pause button will continue code running until the next breakpoint is fired.



Step Over

This will *step over* the next function call (as in stay in the current function instead of following the line to a new function). The function is still called, but the debugger does not follow it.



In the example above, this would skip over the `writeHead` function instead of going into it. The `writeHead` function is still called, but our debugger lets it run and then jumps to the next line.

```
6 const getIndex = (request, response) => {  
7   response.writeHead(200, { 'Content-Type'  
8     response.write(index);  
9   response.end();  
10 }
```

DEBUGGING NODE

Step Into

The *step into* button will go into a function to start debugging that function. This will follow the function call of the current line (if it is a function), leaving the current function and jumping into the new function.



Using our example above, this would cause the debugger to move into the write head function.

```
const getIndex = (request, response) => { request = IncomingMessage;
response.writeHead(200, { 'Content-Type': 'text/html' });
response.write(index);
response.end();
};
```

Jumps into the writeHead function from the function shown above. Now we would be able to start stepping through the writeHead function.

```
ServerResponse.prototype.writeHead = writeHead;
function writeHead(statusCode, reason, obj) { statusCode = 200, reason = 'OK';
var headers; headers = undefined;

if (typeof reason === 'string') {
// writeHead(statusCode, reasonPhrase[, headers])
this.statusMessage = reason;
} else {
// writeHead(statusCode[, headers])
this.statusMessage =
this.statusMessage || STATUS_CODES[statusCode] || 'unknown';
obj = reason;
}
this.statusCode = statusCode;
```

DEBUGGING NODE

Step Out

The *step out* button will leave the current function and jump back to the previous function that called it (if there was one).



Using our example above, this would cause the debugger to move out of the `getIndex` function (completing this function) and going back to the line that called `getIndex`.

```
const getIndex = (request, response) => { request = IncomingMessage;
response.writeHead(200, { 'Content-Type': 'text/html' });
response.write(index);
response.end();
};
```

In this case it jumps to this parent function that called it, so we can start debugging this function.

```
const get = (request, response) => { request = IncomingMessage;
const parsedUrl = url.parse(request.url);

if (parsedUrl.pathname === '/') { parsedUrl = { pathname: '/', search: '' };
htmlHandler.getIndex(request, response);
} else if (parsedUrl.pathname === '/style.css') {
htmlHandler.getCSS(request, response);
} else if (parsedUrl.pathname === '/getUsers') {
jsonHandler.getUsers(request, response);
} else {
jsonHandler.notFound(request, response);
}
```

Toggle All Breakpoints

This allows us to temporarily toggle all breakpoints on or off for testing. This can be good if you want to test results of a change/fix but not remove all the breakpoints manually yet. You can turn them off, hit play, test and turn them back on.



Pause on Exceptions

This allows us to automatically jump to debugging when an exception occurs. This is very useful when trying to test when an exception occurs.

