

9 Debugging Processes

This chapter covers the following topics

- Turning on internal debug flags
- Instrumenting code with debug logs
- Using Chrome Devtools to debug Node
- Enhancing stack traces
- Improve stack trace presentation
- Tracing asynchronous operations
- Postmortems

Introduction

One to two paragraph intro to chapter

Debugging Node with Chrome Devtools

Node 6.3.0 onwards provides us with the `--inspect` flag which we can use to debug the Node runtime with Google Chrome's Devtools.

Debugging Legacy Node

This recipe can be followed with older versions of Node prior to Node 6.3.0 - it just requires a little more set up. To follow this recipe in a legacy Node version, jump to [Using `node-inspector` with Older Node Versions](#) in the [There's More](#) section of this recipe first.

In this recipe we're going to diagnose and solve a problem in a simple Express application.

Getting Ready

We're going to debug a small web server, so let's create that real quick.

On the command line we execute the following commands:

```
$ mkdir app
$ cd app
$ npm init -y
$ npm install --save express
$ touch index.js future.js past.js
```

Our `index.js` file should contain the following:

```
const express = require('express')
const app = express()
const past = require('./past')
const future = require('./future')

app.get('/:age', (req, res) => {
  res.send(past(req.params.age, 10) + future(req.params.future, 10))
})

app.listen(3000)
```

Our `past.js` should look like:

```
module.exports = (age, gap) => {
  return `${gap} years ago you were ${Number(age) - gap}<br>`
}
```

And our `future.js` file should be:

```
module.exports = (age, gap) => {
  return `In ${gap} years you will be ${Number(age) + gap}<br>`
```

Web Frameworks

We're only using express here as an example, to learn more about Express and other Frameworks see **Chapter 7 Working With Web Frameworks**

How to do it

When we run our server (which we created in the Getting Ready section) normally, and navigate our browser to (for instance) `http://localhost:3000/31` the output is as follows:

```
10 years ago you were 21
In 10 years you will be NaN
```

Looks like we have a Not a Number problem.

Let's start our server in inspection mode:

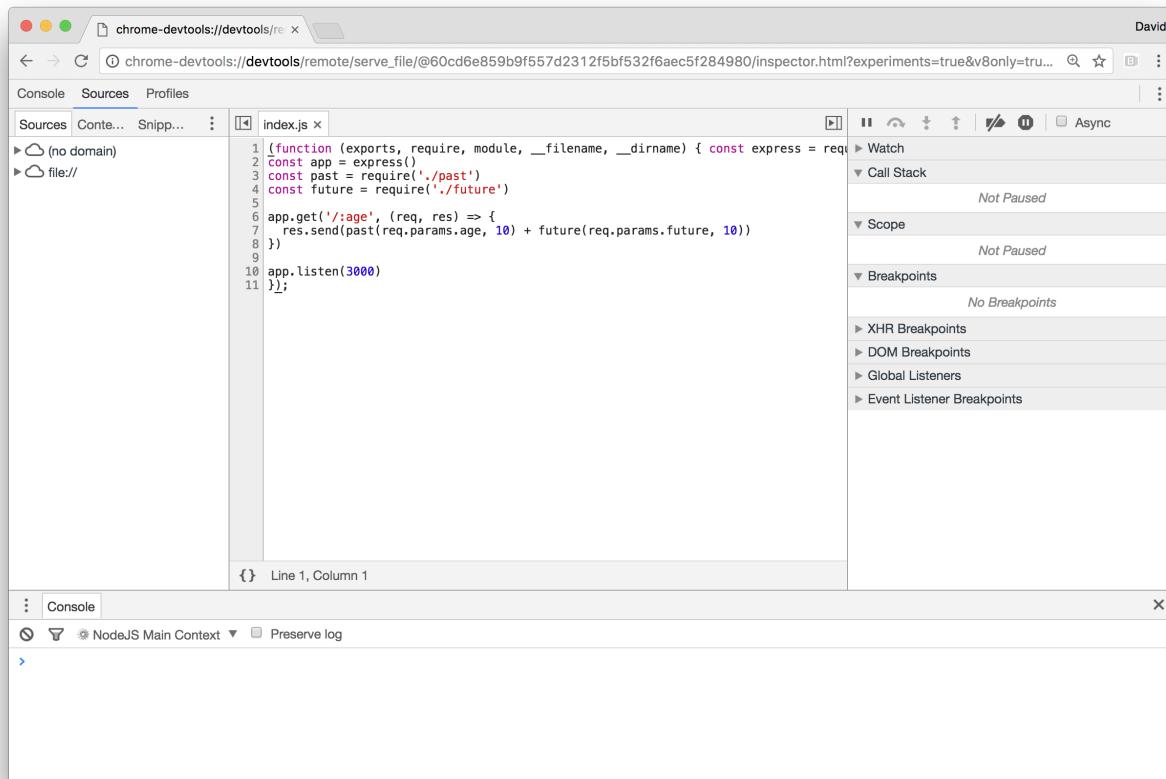
```
$ node --inspect index.js
```

We should see output that's something like the following:

```
Debugger listening on port 9229.
To start debugging, open the following URL in Chrome:
chrome-devtools://devtools/remote/serve_file/@60cd6e859b9f557d2312f5bf532f6aec5f284980/inspector.html?experiments=true&v8only=true
```

Let's copy and paste full the URL with the `chrome-devtools://` protocol, into you Chromes location bar.

We should then see something like the following:



Chrome Devtools Running in Chrome Browser

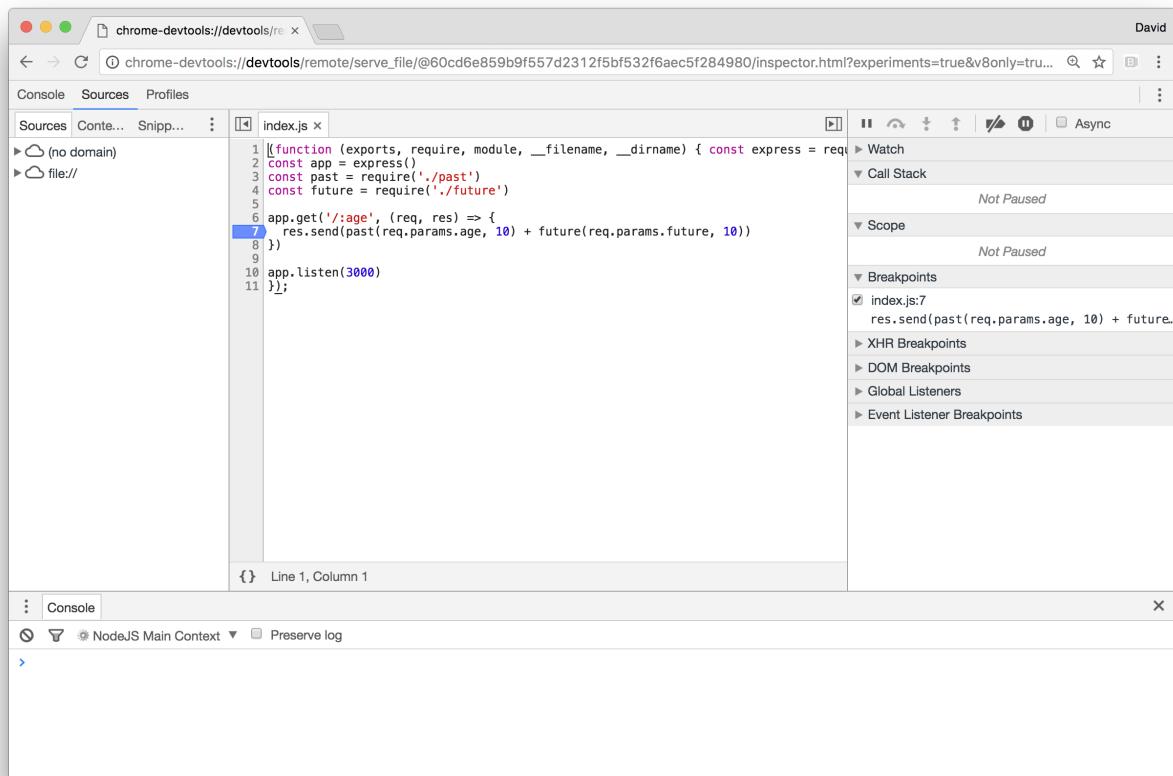
The Module Wrapper 

Notice that the Devtools Code section shows an additional outer function wrapping the code we placed into `index.js`. This outer function is added at run time to each code file loaded into the process (either by directly starting the file with `node` or by using `require` to load it). This outer function is the Module Wrapper, it's the mechanism Node uses to supply local references like `module` and `__filename` that are unique to our module without polluting global scope.

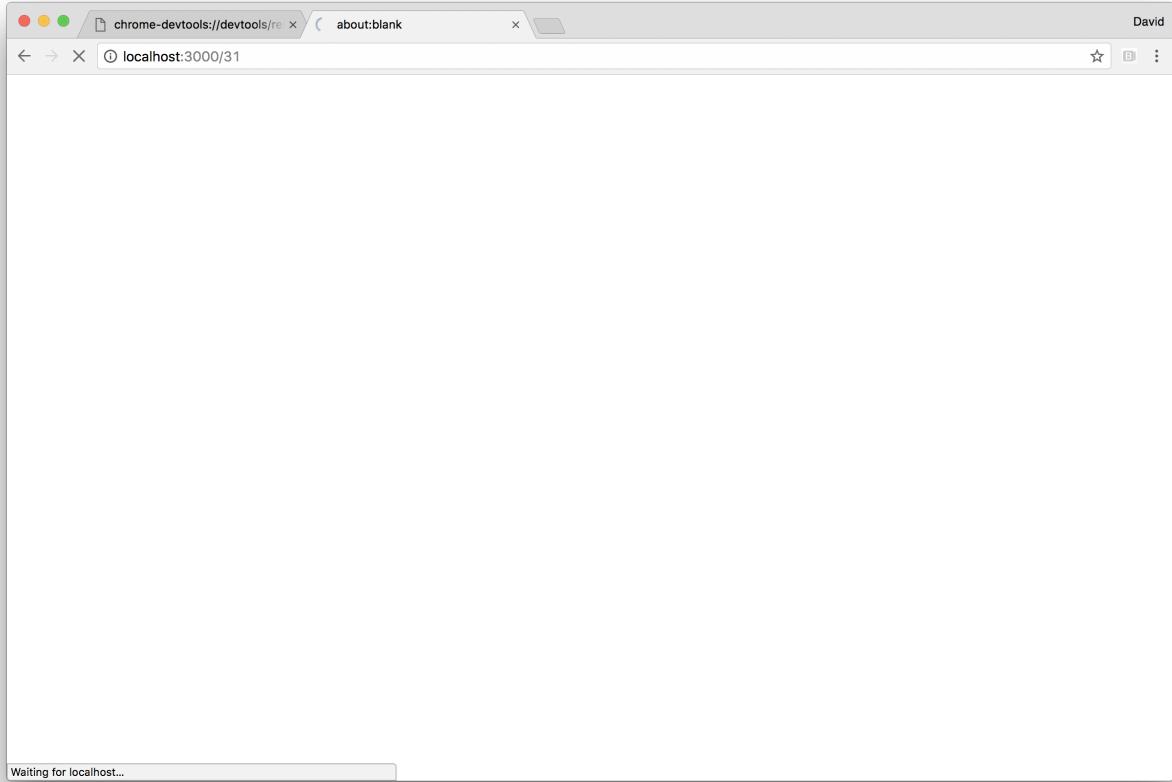
Now let's set a break inside our route handler, on line 7.

If we click the number 7 in the LOC column to the left of the code, it should an arrow like shape over and around the number, also making color the color of the 7 white. Over on the right hand column, in the Breakpoints pane we should also see a checkbox with "index.js:7" next to it, while beneath that is the code from the line we've set a breakpoint on.

In short, the Devtools GUI should now something look like the following:

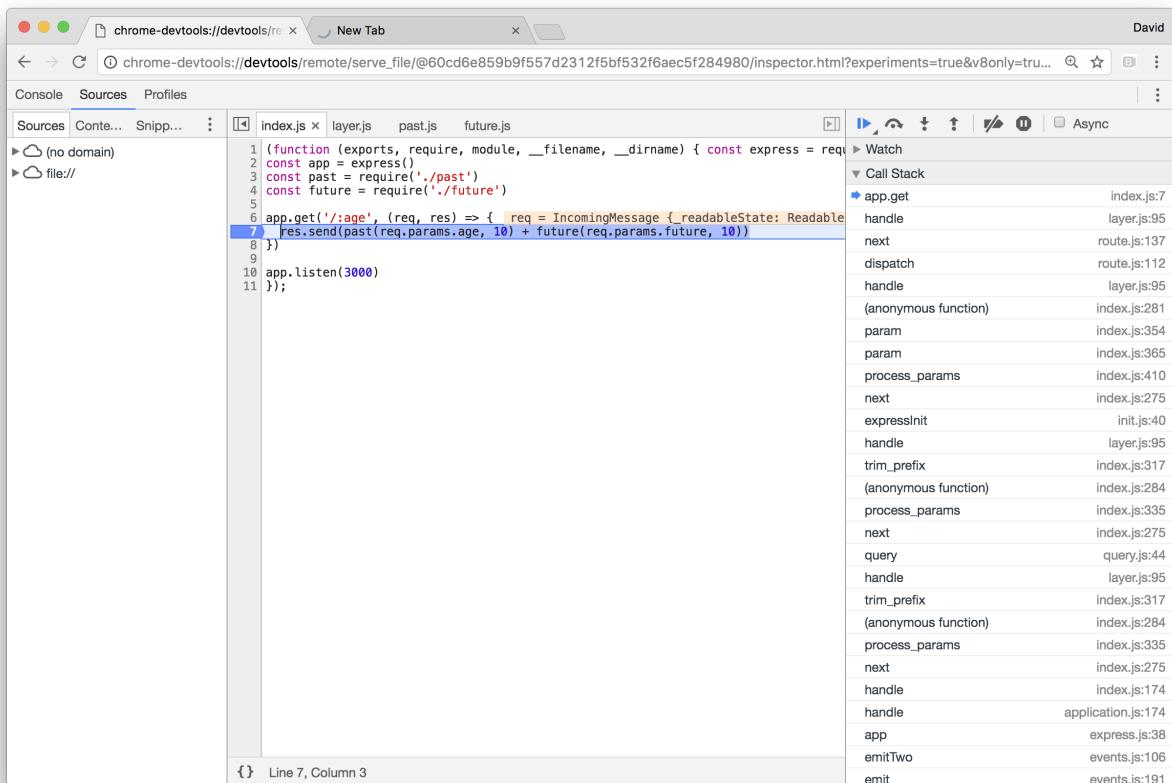


Now let's open a new tab, and navigate to `http://localhost:3000/31`



This will cause the breakpoint to trigger, and Devtools will immediately grab focus.

The next thing we see should look like the following:



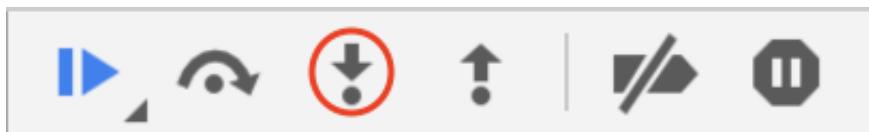
We can see line 7 is now highlighted, and there's a sort of tooltip showing us the

values of the `req` and `res` objects on the line above.

Over in the right column the Call Stack panel is full of Call Frames (the functions in the stack), and there's now a blue play button in the control icons at the top of the right column. If we scroll the right column, we'd also see the Scope pane is populated with references.

The debugger is waiting for us to allow execution to proceed, and we can choose whether to step over, in or out of the next instruction.

Let's try stepping in, this is the down arrow pointing to a dot, third icon from the left in the controls section:



When we press this, we step into the `past` function, which is in the `past.js` file, so Devtools will open a new tab in the above center code panel, and highlight the line which is about to execute (in our case, line 2).

The screenshot shows the Chrome DevTools interface with the following details:

- Sources Panel:** Shows tabs for `index.js` and `past.js`. The `past.js` tab is active, displaying the following code:

```
1 (function (exports, require, module, __filename, __dirname) { module.exports = (a  
2   return `${gap} years ago you were ${Number(age) - gap}`  
3 );  
4 });
```
- Call Stack Panel:** Shows a list of call frames:
 - module.exports (past.js:2)
 - app.get (index.js:7)
 - handle (layer.js:95)
 - next (route.js:137)
 - dispatch (route.js:112)
 - handle (layer.js:95)
 - (anonymous function) (index.js:281)
 - param (index.js:354)
 - param (index.js:365)
 - process_params (index.js:410)
 - next (index.js:275)
 - expressInit (init.js:40)
 - handle (layer.js:95)
 - trim_prefix (index.js:317)
 - (anonymous function) (index.js:284)
 - process_params (index.js:335)
 - next (index.js:275)
 - query (query.js:44)
- Console Panel:** Shows the message: "Line 2, Column 3".

So let's step out of the `past` function, by pressing the arrow pointing up and away from a dot, next to the step in icon:



The second line of the output seems to have the issue, which is our `future` function.

Now that we've stepped out, we can see that the call to `future` is highlighted in a darker shade of blue:

The screenshot shows the Chrome DevTools Sources tab for a Node.js application running at localhost:3000/31. The code in `index.js` is as follows:

```
1 (function(exports, require, module, __filename, __dirname) { const express = require('express');
2 const app = express();
3 const past = require('./past')
4 const future = require('./future')
5
6 app.get('/:age', (req, res) => {
7   req = IncomingMessage { readableState: ReadableStreamDefaultController {
8     res.send(past(req.params.age, 10) + future(req.params.future, 10))
9   })
10 })
11 app.listen(3000)
12});
```

The line `7 res.send(past(req.params.age, 10) + future(req.params.future, 10))` is highlighted in blue. To the right of the code, the Call Stack shows the execution path:

- Watch
- Call Stack
 - app.get
 - handle
 - next
 - dispatch
 - handle
 - (anonymous function)
 - param
 - param
 - process_params
 - next
 - expressInit
 - handle
 - trim_prefix
 - (anonymous function)
 - process_params
 - next
 - query
 - handle

The call stack also includes entries from `layer.js`, `route.js`, and `index.js`.

Then we'll press the step in icon again, which will take us into the `future` function in the `future.js` file:

The screenshot shows the Chrome DevTools Sources tab. In the code editor, there is a syntax error highlighted in red on line 2 of file 'future.js':

```
1 (function (exports, require, module, __filename, __dirname) { module.exports = (a
2 return `In ${gap} years you will be ${Number(age) + gap}`  
3 }
4 });

{} Line 2, Column 3
```

The right panel displays the Call Stack, listing the execution path from the error to the top of the application:

- ▶ Watch
- ▼ Call Stack
 - ▶ module.exports future.js:2
 - app.get index.js:7
 - handle layer.js:95
 - next route.js:112
 - dispatch route.js:112
 - handle layer.js:95
 - (anonymous function) index.js:281
 - param index.js:354
 - param index.js:365
 - process_params index.js:410
 - next index.js:275
 - expressInit init.js:40
 - handle layer.js:95
 - trim_prefix index.js:317
 - (anonymous function) index.js:284
 - process_params index.js:335
 - next index.js:275
 - query query.js:44

Okay, this is the function that generates that particular sentence with the `Nan` in it. A `Nan` can be generated for all sort of reasons, dividing zero by itself, subtracting `Infinity` from `Infinity` trying coerce a string to a number when the string does not hold a valid number, to name a few. At any rate, it's probably something to do with the values in our `future` function.

Let's hover over the `gap` variable, we should see:

The screenshot shows the Chrome DevTools Sources tab. In the code editor, there is a tooltip for the variable 'age' at line 2, column 3. The tooltip displays the value '10'. The code in the editor is:

```
1 (function (exports, require, module, __filename, __dirname) { module.exports = (a  
2 return `In ${gap} years you will be ${Number(age) + gap}`  
3 } );  
4 );
```

The right panel shows a call stack with various file names and line numbers. The bottom panel is a console window.

Seems fine, now let's hover over the `age` variable:

The screenshot shows the Chrome DevTools Sources tab. In the code editor, there is a tooltip for the variable 'age' at line 2, column 3. The tooltip displays the value 'undefined'. The code in the editor is identical to the previous screenshot.

Wait why does that say `undefined`, we vicariously passed `31` by navigating to the `http://localhost:3000/31`.

To be sure our eyes aren't deceiving us, we can double check by collapsing the Call Stack column (by clicking the small downwards arrow next to the C of Call Stack). We should see:

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. In the main pane, the 'future.js' file is open, showing the following code:

```
1 (function (exports, require, module, __filename, __dirname) { module.exports = (a  
2   return `In ${gap} years you will be ${Number(age) + gap}<br>`  
3 }  
4 );
```

The cursor is positioned at line 2, column 3, where the variable 'age' is defined. The 'Call Stack' section is collapsed. The 'Scope' section shows the following local variables:

- Local
 - age: undefined
 - gap: 10
 - this: global

The 'Breakpoints' section shows a checked entry for 'index.js:7':
res.send(past(req.params.age, 10) + future...
The 'Console' tab at the bottom is active.

Well `Number(undefined)` is `Nan`, and `Nan + 10` is also `Nan`.

Why is `age` set to `undefined`?

Let's open up the Call Stack bar again, and click the second row from the top (which says `app.get`).

We should be back in the `index.js` file again (but still frozen on line 2 of `future.js`). Like so:

The screenshot shows the Chrome DevTools interface. In the Sources panel, there are three files listed: index.js*, layer.js, and future.js. The index.js file is open, showing the following code:`1 (function (exports, require, module, __filename, __dirname) { const express = require('express')
2 const app = express()
3 const past = require('../past')
4 const future = require('../future')
5
6 app.get('/:age', (req, res) => {
7 res.send(past(req.params.age, 10) + future(req.params.age, 10))
8 })
9
10 app.listen(3000)
11 })`Line 7, Column 60

In the Call Stack panel, the stack trace is shown:

- Watch
- Call Stack
 - Not Paused
- Scope
 - Not Paused
- Breakpoints
 - No Breakpoints
- XHR Breakpoints
- DOM Breakpoints
- Global Listeners
- Event Listener Breakpoints

Now let's hover over the value we're passing in to `future`:

The screenshot shows the Chrome DevTools interface. In the Sources panel, there are three files listed: index.js*, past.js, and layer.js. The index.js file is open, showing the following code:`1 (function (exports, require, module, __filename, __dirname) { const express = require('express')
2 const app = express()
3 const past = require('../past')
4 const future = require('../future')
5
6 app.get('/:age', (req, res) => {
7 res.send(past(req.params.age, 10) + future(req.params.future, 10))
8 })
9
10 app.listen(3000)
11 })`Line 7, Column 39

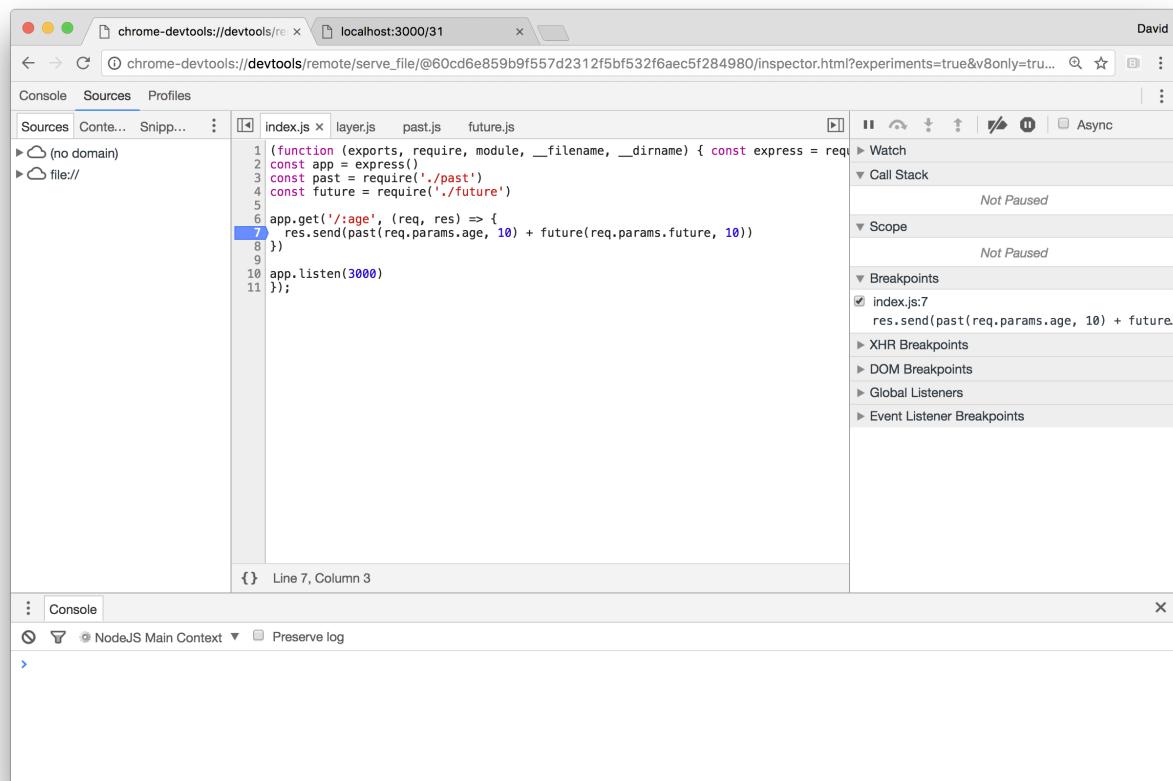
A tooltip "undefined" is displayed over the `req.params.future` parameter in the code. In the Call Stack panel, the stack trace is shown:

- Watch
- Call Stack
 - module.exports future.js:2
 - app.get index.js:7
 - handle layer.js:95
 - next route.js:137
 - dispatch route.js:112
 - handle layer.js:95
 - (anonymous function) index.js:281
 - param index.js:354
 - param index.js:365
 - process_params index.js:410
 - next index.js:275
 - expressInit init.js:40
 - handle layer.js:95
 - trim_prefix index.js:317
 - (anonymous function) index.js:284
 - process_params index.js:335
 - next index.js:275
 - query query.js:44

That's `undefined` too, why is it `undefined` ?!

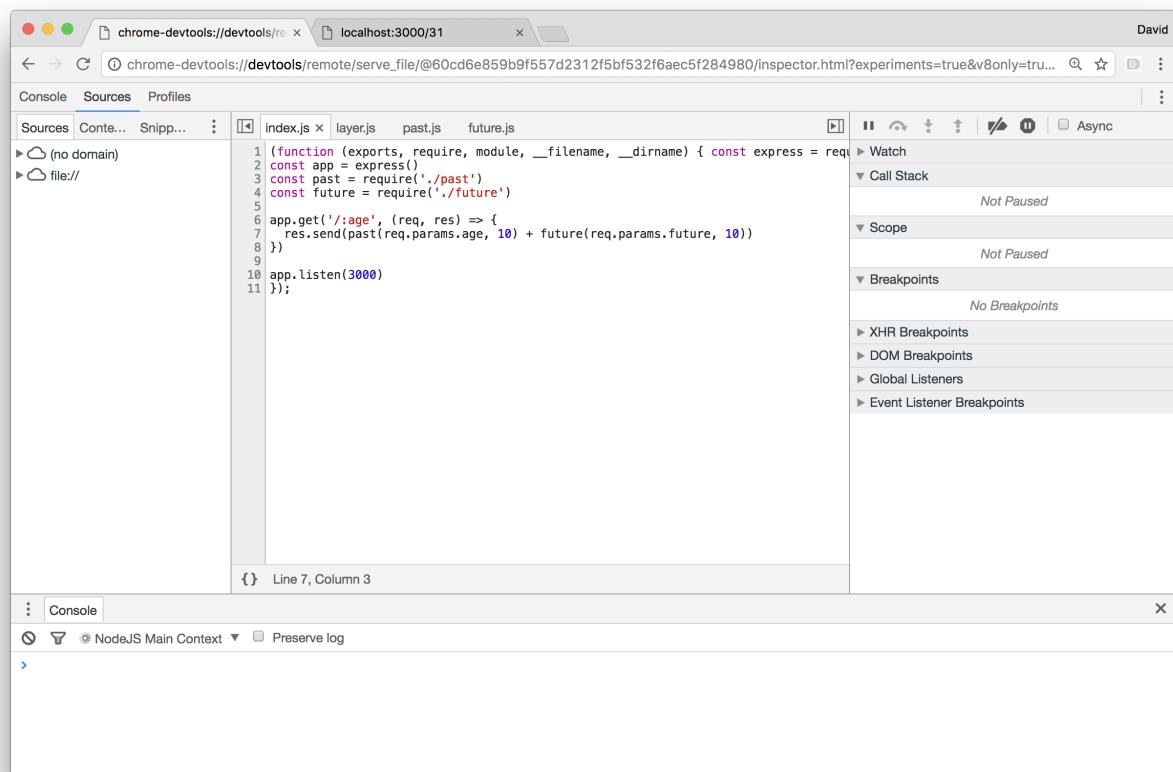
Oh. That should be `req.params.age` not `req.params.future`. Oops.

To be absolutely sure, let's fix it while the server is running, if we hit the blue play button twice we should see something like:



The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. The left pane displays the file structure: 'index.js x layer.js past.js future.js'. The right pane shows the code for 'index.js':1 (function (exports, require, module, __filename, __dirname) { const express = require('express')
2 const app = express()
3 const past = require('../past')
4 const future = require('../future')
5
6 app.get('/:age', (req, res) => {
7 res.send(past(req.params.age, 10) + future(req.params.future, 10))
8 }
9
10 app.listen(3000)
11});The line 7 is highlighted with a blue background. The right sidebar shows the 'Breakpoints' section with a checked checkbox for 'index.js:7'. Below the code editor, a status bar indicates 'Line 7, Column 3'.

Now let's click line 7 again to remove the breakpoint, we should be seeing:



The screenshot shows the same Chrome DevTools interface after removing the breakpoint from line 7. The right sidebar's 'Breakpoints' section now shows 'No Breakpoints'. The rest of the interface remains identical to the previous screenshot.

Now if we click immediately after the `e` in `req.params.future` we should get a blink cursor, we backspace out the word `future` and type the word `age`, making our code look like so:

localhost:3000/31

Sources | index.js | layer.js | past.js | future.js

```
1 (function (exports, require, module, __filename, __dirname) { const express = require('express')
2 const app = express()
3 const past = require('./past')
4 const future = require('./future')
5
6 app.get('/:age', (req, res) => {
7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
8 })
9
10 app.listen(3000)
11 })
```

{ } Line 7, Column 63

Watch

Call Stack

module.exports

future.js:2

app.get

index.js:7

handle

layer.js:95

next

route.js:137

dispatch

route.js:112

handle

layer.js:95

(anonymous function)

index.js:281

param

index.js:354

param

index.js:365

process_params

index.js:410

next

index.js:275

expressInit

init.js:40

handle

layer.js:95

trim_prefix

index.js:317

(anonymous function)

index.js:284

process_params

index.js:335

next

index.js:275

query

query.js:44

handle

layer.js:95

trim_prefix

index.js:317

(anonymous function)

index.js:284

process_params

index.js:335

next

index.js:275

handle

index.js:174

handle

express.js:38

app

express.js:106

emitTwo

events.js:106

Finally we can save those changes in our running server by pressing CMD + s on MacOS or Ctrl + s on Windows and Linux.

The code panel will then change background color, like so:

The screenshot shows the Chrome DevTools interface with the 'Sources' tab selected. The main pane displays the code for 'index.js' with a breakpoint set at line 7, column 60. The code is as follows:

```
1 (function (exports, require, module, __filename, __dirname) { const express = require('express')
2 const app = express()
3 const past = require('../past')
4 const future = require('../future')
5
6 app.get('/:age', (req, res) => {
7   res.send(past(req.params.age, 10) + future(req.params.age, 10))
8 })
9
10 app.listen(3000)
11 })
```

The right sidebar contains various developer tools like Watch, Call Stack, Scope, Breakpoints, XHR Breakpoints, DOM Breakpoints, Global Listeners, and Event Listener Breakpoints. The 'Scope' section shows 'Not Paused'. The 'Breakpoints' section shows 'No Breakpoints'.

Finally, let's check our route again:

The screenshot shows the browser window displaying the results of the route check. The output is:

```
10 years ago you were 21
In 10 years you will be 41
```

Ok we've definitely found the problem, and verified a solution.

How it works

We don't really need to know how debugging Node with devtools is made possible in order to avail of the tool, however, for the curious here's a high level overview.

Debugging ability is ultimately provided by v8, the JavaScript engine used by Node. When we run `node` with the `--inspect` flag the v8 inspector opens a port, that accepts WebSocket connections. Once a connection is established commands in the form of JSON packets are sent back and forth between the inspector and a client.

The `chrome-devtools://` URI is a special protocol recognized by the chrome browser that loads the Devtools UI (which is written in HTML, CSS and JavaScript, so can be loaded directly into a normal browser tab). The Devtools UI is loaded in a special mode (remote mode), where a WebSocket endpoint is supplied via the URL.

The WebSocket connection allows for bi-directional communication between the inspector and the client. The tiny Inspector WebSocket server is written entirely in C and runs on a separate thread so that when the process is paused, the inspector can continue to receive and send commands.

In order to achieve the level of control we're afforded in debug mode (ability to pause, step, inspect state, view callstack, live edit) v8 operations are instrumented throughout with Inspector C++ functions that can control the flow, and change state in place.

For instance, if we've set a breakpoint, once that line is encountered a condition will match in the C++ level that triggers a function which pauses the event loop (the JavaScript thread). The Inspector then sends a message to the client over the WebSocket connection telling it that the process is paused on a particular line and the client updates its state. Like wise, if the user chooses to "step into" a function, this command is sent to the Inspector, which can briefly unpause and repause the event loop in the appropriate place, then sends a message back with the new position and state.

There's more

Let's find out how to debug older versions of Node, make a process start with a paused runtime and learn to use the builtin command line debugging interface.

Using `node-inspector` with Older Node Versions

The `--inspect` flag and protocol were introduced in Node 6.3.0, primarily because the v8 engine had changed the debugging protocol. In Node 6.2.0 and down, there's a legacy debugging protocol enabled with the `--debug` flag but this isn't compatible with the native Chrome Devtools UI.

Instead we can use the `node-inspector` tool, as a client for the legacy protocol.

The `node-inspector` tool essentially wraps an older version of Devtools that interfaces with the legacy debug API, and then hosts it locally.

Let's install `node-inspector`:

```
$ npm i -g node-inspector
```

This will add a global executable called `node-debug` which we can use as shorthand to start our process in debug mode.

If we run our process like so

```
$ node-debug index.js
```

We should see output something like the following:

```
Node Inspector v0.12.10
Visit http://127.0.0.1:8080/?port=5858 to start debugging.
Debugging `index.js`

Debugger listening on [::]:5858
```

When we load the url <http://127.0.0.1:8080/?port=5858> in our browser we'll again see the familiar Devtools interface.

By default, the `node-debug` command starts our process in a paused state, after pressing run (the blue play button), we should now be able to follow the main recipe in its entirety using a legacy version of Node.

Immediately pausing a process on start

In many cases we want to debug a process from initialization, or we may want to set up breakpoints before anything can happen.

From Node 8 onwards we use the following to start Node in an immediately paused state:

```
$ node --inspect-brk index.js
```

In Node 6 (at time of writing 6.10.0), `--inspect` is supported but `--inspect-brk` isn't. Instead we can use the legacy `--debug-brk` flag in conjunction with `--inspect` like so:

```
$ node --debug-brk --inspect index.js
```

In Node v4 and lower, we'd simply use `--debug-brk` instead of `--debug` (in conjunction with another client, see [Using Node Inspector with Older Node Versions](#))

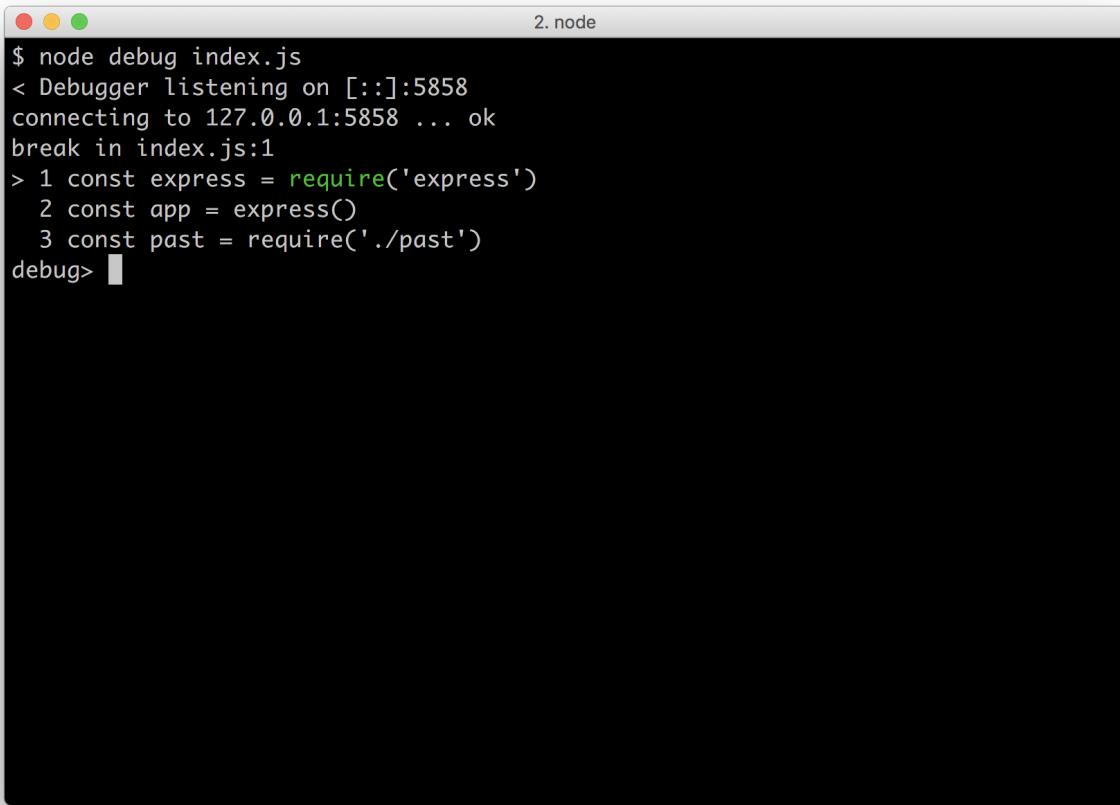
node debug

There may be rare occasions when don't have easy access to a GUI, in these scenarios command line abilities become paramount.

Let's take a look at Nodes built in command line debugging interface.

Let's run our app from the main recipe like so:

```
$ node debug index.js
```



```
$ node debug index.js
< Debugger listening on [::]:5858
connecting to 127.0.0.1:5858 ... ok
break in index.js:1
> 1 const express = require('express')
  2 const app = express()
  3 const past = require('./past')
debug> 
```

When we enter debug mode, we see the first three lines of our entry point (`index.js`).

Upon entering debug mode, the process is paused on the first line of the entry point. By default, when a breakpoint occurs the debugger shows 2 lines before and after the current line of code, since this is the first line we only see two lines after.

The debug mode provides several commands in the form of functions, or sometimes as magic getter/setters (we can view these commands by typing `help` and hitting enter)

Let's get a little context using the `list` function,

```
debug> list(10)
```

```
$ node debug index.js
< Debugger listening on [::]:5858
connecting to 127.0.0.1:5858 ... ok
break in index.js:1
> 1 const express = require('express')
  2 const app = express()
  3 const past = require('./past')
debug> list(10)
> 1 const express = require('express')
  2 const app = express()
  3 const past = require('./past')
  4 const future = require('./future')
  5
  6 app.get('/:age', (req, res) => {
  7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
  8 })
  9
10 app.listen(3000)
11 };
debug> 
```

This provides 10 lines after our current line (again it would also include 10 lines before, but we're on the first line so there's no prior lines to show).

We're interested in the 7th line, because this is the code that's executed when the server receives a request.

We can use the `sb` function (which stands for Set Breakpoint), to set a break point on line 7, like so:

```
debug> sb(7)
```

Now if we use `list(10)` again, we should see an asterisk (*) adjacent to line 7.

```
debug> list(10)
```

```
2. node
 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
 8 }
 9
10 app.listen(3000)
11 });
debug> sb(7)
> 1 const express = require('express')
 2 const app = express()
 3 const past = require('./past')
 4 const future = require('./future')
 5
 6 app.get('/:age', (req, res) => {
debug> list(10)
> 1 const express = require('express')
 2 const app = express()
 3 const past = require('./past')
 4 const future = require('./future')
 5
 6 app.get('/:age', (req, res) => {
* 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
 8 }
 9
10 app.listen(3000)
11 });
debug> █
```

Since our app began in paused mode, we need to tell the process to begin running as normal so we can send a request to it.

We use the `c` command to tell the process to continue, like so:

```
debug> c
```

Now let's make a request to our server, we could use a browser to do this, or if we have `curl` on our system, in another terminal we could run the following:

```
$ curl http://localhost:3000/31
```

This will cause the process to hit our breakpoint and the debugger console should print out "break in index.js:7" along with the line our code is currently paused on, with 2 lines of context before and after. We can see a right caret (`>`) indicating the current line.

```
2 const app = express()
3 const past = require('./past')
4 const future = require('./future')
5
6 app.get('/:age', (req, res) => {
debug> list(10)
> 1 const express = require('express')
2 const app = express()
3 const past = require('./past')
4 const future = require('./future')
5
6 app.get('/:age', (req, res) => {
* 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
8 })
9
10 app.listen(3000)
11 });
debug> c
break in index.js:7
  5
  6 app.get('/:age', (req, res) => {
> 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
  8 )
  9
debug> 
```

Now let's step in to the first function, to step in we use the `step` command:

```
debug> step
```

```
node      curl      2. node
> 1 const express = require('express')
  2 const app = express()
  3 const past = require('./past')
  4 const future = require('./future')
  5
  6 app.get('/:age', (req, res) => {
* 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
  8 })
  9
10 app.listen(3000)
11 });
debug> c
break in index.js:7
  5
  6 app.get('/:age', (req, res) => {
> 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
  8 )
  9
debug> step
break in past.js:2
  1 module.exports = (age, gap) => {
> 2   return `${gap} years ago you were ${Number(age) - gap}<br>`
  3 }
  4 );
debug> 
```

This enters our `past.js` file, with the current break on line 2.

We can print out references in the current debug scope using the `exec` command, let's print out the values of the `gap` and `age` arguments:

```
debug> exec gap
```

```
debug> exec age
```

```
2. node
node ⌘1 curl ⌘2
5
6 app.get('/:age', (req, res) => {
* 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
8 })
9
10 app.listen(3000)
11 });
debug> c
break in index.js:7
5
6 app.get('/:age', (req, res) => {
> 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
8 )
9
debug> step
break in past.js:2
1 module.exports = (age, gap) => {
> 2   return `${gap} years ago you were ${Number(age) - gap}<br>`
3 }
4 );
debug> exec gap
10
debug> exec age
'31'
debug>
```

Everything seems to be in order here.

Now let's step back out of the `past` function, we use the `out` command to do this, like so:

```
debug> out
```

```
2. node
node ⌘1 curl ⌘2
debug> c
break in index.js:7
  5
  6 app.get('/:age', (req, res) => {
> 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
  8 }
  9
debug> step
break in past.js:2
  1 module.exports = (age, gap) => {
> 2   return `${gap} years ago you were ${Number(age) - gap}<br>`
  3 }
  4 );
debug> exec gap
10
debug> exec age
'31'
debug> out
break in index.js:7
  5
  6 app.get('/:age', (req, res) => {
> 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
  8 }
  9
debug>
```

We should now see that the `future` function is a different color, indicating that this is the next function to be called. Let's step into the `future` function.

```
debug> step
```

```
node      2. node
          ⇧1   ⇧2
         ⌂ curl
9
debug> step
break in past.js:2
  1 module.exports = (age, gap) => {
> 2   return `${gap} years ago you were ${Number(age) - gap}<br>`
  3 }
  4 );
debug> exec gap
10
debug> exec age
'31'
debug> out
break in index.js:7
  5
  6 app.get('/:age', (req, res) => {
> 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
  8 }
  9
debug> step
break in future.js:2
  1 module.exports = (age, gap) => {
> 2   return `In ${gap} years you will be ${Number(age) + gap}<br>`
  3 }
  4 );
debug>
```

Now we're in our `future.js` file, again we can print out the `gap` and `age` arguments using `exec`:

```
debug> exec gap
```

```
debug> exec age
```

```
2. node
node  curl  curl
> 2  return `${gap} years ago you were ${Number(age) - gap}<br>`  
  3 }  
  4 );  
debug> exec gap  
10  
debug> exec age  
'31'  
debug> out  
break in index.js:7  
  5  
  6 app.get('/:age', (req, res) => {  
> 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))  
  8 })  
  9  
debug> step  
break in future.js:2  
  1 module.exports = (age, gap) => {  
> 2   return `In ${gap} years you will be ${Number(age) + gap}<br>`  
  3 }  
  4 );  
debug> exec gap  
10  
debug> exec age  
undefined  
debug> 
```

Aha, we can see that `age` is `undefined`, let's step back up into the router function using the `out` command:

```
debug> out
```

Let's inspect `req.params.future` and `req.params`:

```
debug> req.params.future
```

```
debug> req.params
```

```
2. node
node curl
> 7 res.send(past(req.params.age, 10) + future(req.params.future, 10))
 8 }
 9
debug> step
break in future.js:2
 1 module.exports = (age, gap) => {
> 2   return `In ${gap} years you will be ${Number(age) + gap}<br>`
 3 }
 4 );
debug> exec gap
10
debug> exec age
undefined
debug> out
break in index.js:7
 5
 6 app.get('/:age', (req, res) => {
> 7   res.send(past(req.params.age, 10) + future(req.params.future, 10))
 8 }
 9
debug> exec req.params.future
undefined
debug> exec req.params
{ age: '31' }
debug> 
```

It's now (again) obvious where the mistake lies, there is no `req.params.future`, that input should be `req.params.age`.

See also

Enhancing Stack Trace Output

[cute stack]

Getting Ready

How to do it

How it works

There's more

Increasing Stacksize

Asynchronous Stack Traces

`Error.prepareStackTrace`

See also

Enabling Debug Logs

Getting Ready

How to do it

How it works

There's more

`pino-debug`

####

See also

Enabling Core Debug Logs

It can be highly useful to understand what's going on in Node's core, there's a very easy way to get this information.

In this recipe we're going to use a special environment variable to enable various debugging flags that cause Node Core debug logging mechanisms to print to STDOUT.

Getting Ready

We're going to debug a small web server, so let's create that real quick.

On the command line we execute the following commands:

```
$ mkdir app
$ cd app
$ npm init -y
$ npm install --save express
$ touch index.js
```

Our `index.js` file should contain the following:

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

app.get('/', (req, res) => res.send('hey'))

setTimeout(function myTimeout() {
  console.log('I waited for you.')
}, 100)

app.listen(3000)
```

Web Frameworks

We're only using express here as an example, to learn more about Express and other Frameworks see [Chapter 7 Working With Web Frameworks](#)

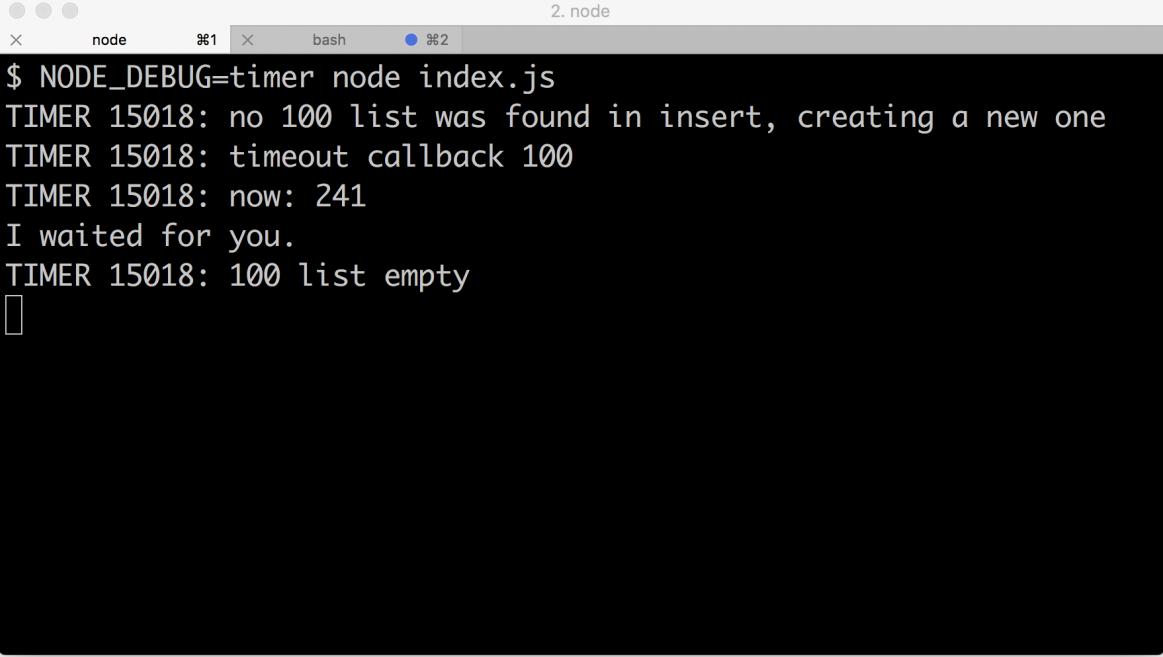
How to do it

We simply have to set the `NODE_DEBUG` environment variable to one or more of the supported flags.

Let's start with the `timer` flag by running our app like so:

```
$ NODE_DEBUG=timer node index.js
```

This should show something like the following figure:



A screenshot of a terminal window titled "2. node". The window has three tabs: "node", "⌘1", and "bash". The "⌘1" tab is active and contains the following text:

```
$ NODE_DEBUG=timer node index.js
TIMER 15018: no 100 list was found in insert, creating a new one
TIMER 15018: timeout callback 100
TIMER 15018: now: 241
I waited for you.
TIMER 15018: 100 list empty
[]
```

Core timer debug output

Let's try running the process again with both `timer` and `http` flags enabled:

```
$ NODE_DEBUG=timer,http node index.js
```

Now we need to trigger some HTTP operations to get any meaningful output, so let's send a request to the HTTP server using `curl` (our an alternative favoured method, such as navigating to `http://localhost:3000` in the browser).

```
$ curl http://localhost:3000
```

This should give output similar to the following image:

The screenshot shows two terminal windows. The left window, titled 'node' and '2. node', displays Node.js debug logs for the 'http' module. It includes logs for timers, HTTP connections, and various internal events like socket onClose and ParserExecute. The right window, titled 'bash', shows a command being run: '\$ curl http://localhost:3000'. The output of this command is 'hey\$'.

```
$ NODE_DEBUG=timer,http node index.js
TIMER 15393: no 100 list was found in insert, creating a new one
TIMER 15393: timeout callback 100
TIMER 15393: now: 242
I waited for you.
TIMER 15393: 100 list empty
HTTP 15393: SERVER new http connection
TIMER 15393: no 120000 list was found in insert, creating a new one
TIMER 15393: no 819 list was found in insert, creating a new one
HTTP 15393: write ret = true
HTTP 15393: outgoing message end.
HTTP 15393: SERVER socketOnParserExecute 78
HTTP 15393: server socket close
TIMER 15393: timeout callback 819
TIMER 15393: now: 4698
TIMER 15393: 819 list empty
```

Core timer and http debug output

How it works

The `NODE_DEBUG` environment variable can be set to any combination of the following flags:

- `http`
- `net`
- `tls`
- `stream`
- `module`
- `timer`
- `cluster`
- `child_process`
- `fs`

The `fs` debug flag

The quality of output varies for each flag. At time of writing, the `fs` flag in particular doesn't actually supply any debug log output, but when enabled will cause a useful stack trace to be added to any unhandled error events for asynchronous I/O calls. See

<https://github.com/nodejs/node/blob/cccc6d8545c0ebd83f934b9734f5605aaeb000f2/lib/fs.js#L76-L94> for context.

In our recipe we were able to enable core timer and HTTP debug logs, by setting the `NODE_DEBUG` environment variable to `timers` in the first instance and then `timers,http` in the second.

We use a comma to delimit the debug flags, however the delimiter can be any character.

Each line of output consists of the namespace, the process ID (PID), and the log message.

When we set `NODE_DEBUG` to `timer`, the first log message indicates that it's creating a list for `100`. Our code passes `100` as the second argument passed to `setTimeout`, internally the first argument (the timeout callback) is added to a queue of callbacks that should run after `100` millisecond. Next we see a message "timeout callback 100" which means every 100ms timeout callback will now be called. The following message (the "now" message) \ indicates the current "time" as the internal `timers` module sees it, this is milliseconds since the `timers` module was initialized. The "now" message can be useful to see the time drift between timeouts and intervals, because a timeout of 10ms will rarely (if ever) be exactly 10 ms. More like 14ms, because of 4ms of execution time for other code in a given tick (time around the event loop). While 4ms drift is acceptable, a 20ms drift would indicate potential performance problems - a simple `NODE_DEBUG=timer` prefix could be used for a quick check. The final debug message shows that the `100` list is now empty, meaning all callback functions set for that particular interval have now been called.

Most of the HTTP output is self explanatory, we can see when a new connection has been made to the server, when a message has ended and when a socket has closed. The remaining two cryptic messages are `write ret = true` and `SERVER socketOnParserExecute 78`. The `write ret = true` relates to when the server attempted to write to a socket, if the value was false it would mean the socket had closed and the (again internally) the server would begin to handle that scenario. As for the `socketOnParserExecute` message, this has to do with Nodes internal HTTP parser (written in C), the number (78) is the string length of the headers sent from the client to the server.

Combining multiple flags can be useful, we set `NODE_DEBUG` to `timer,http` we were able to how the `http` module interacts with the internal `timer` module. We can see after a the "SERVER new http connection" message, that two timers are set (based on the timeout lists being created), one for 120000 milliseconds (two minutes, the default socket timeout) and one (in the example case) for 819 milliseconds.

This second interval (819) is to do with an internal caching mechanism for the HTTP `Date` header. Since the smallest unit in the `Date` header is seconds, a timeout is

set for the amount of milliseconds left before the next second and the `Date` header is provided the same string for the remainder of that second.

Core Mechanics

For a deeper understanding of our discussion here, see the `There's More` section where we use debugging tools to step through code in Node core to show how to fully pick apart the log messages in this recipe.

There's more

Let's look at the way Node Core triggers the debug log messages, and see if we can use this knowledge to gain greater understanding of Node's internal workings.

Creating our own `NODE_DEBUG` flags

Core modules tend to use the `util` module's `debuglog` method to generate a logging function that defaults to a no-op (an empty function) but writes log messages to STDOUT when the relevant flag appears in the `NODE_DEBUG` environment variable.

We can use `util.debuglog` to create our own core like log messages

[how to etc]

[tip box recommending `debug` module instead]

Debugging Node Core Libraries

- run with `--node-inspect` and `--debug-brk` (point to previous recipe dealing with general usage of `node--inspect`)
- navigate to (no domain) files
- find `util.js`
- find `exports.debuglog`
- set breakpoint in function created by `debuglog`
- press run
- cycle through messages until message is odd number (like 819)
- use stack inspector to find the `utcDate` function
- understand `timers.enroll` and `unrefActive`
- understand what `utcDate` is doing

See also

- TBD

Tracing Asynchronous Operations

Getting Ready

How to do it

How it works

There's more

See also

Postmortem Debugging

Getting Ready

How to do it

How it works

There's more

See also