IGME 430 - Rich Media 2

Node Debugging Guide

This guide is a resource to aid in debugging node projects in various ways.

Node Debugger

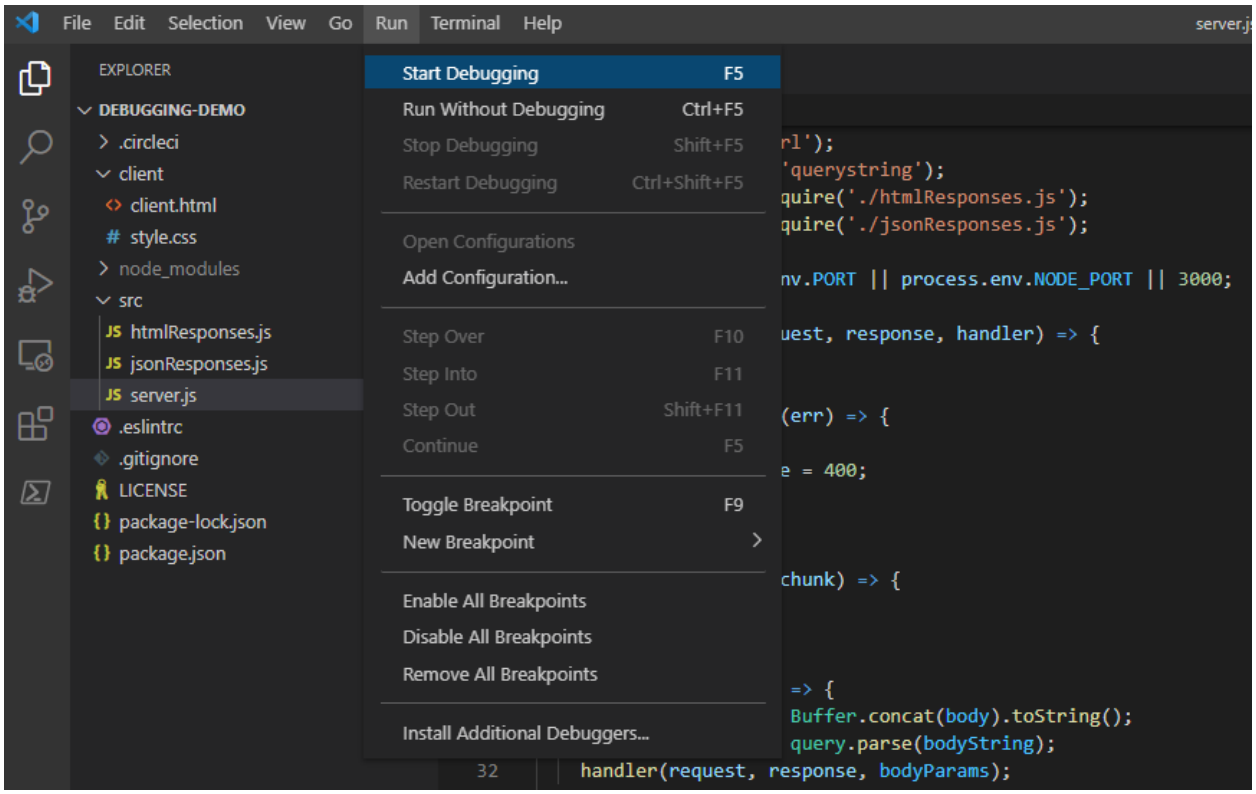
Just like programming in C# or C++ with Visual Studio or programming client-side JS and debugging it with the Chrome Inspector, there are comprehensive debugging tools that will make our lives far easier when trying to solve issues with our node code.

Node has some great documentation on all of these topics:
https://nodejs.org/en/docs/guides/debugging-getting-started/

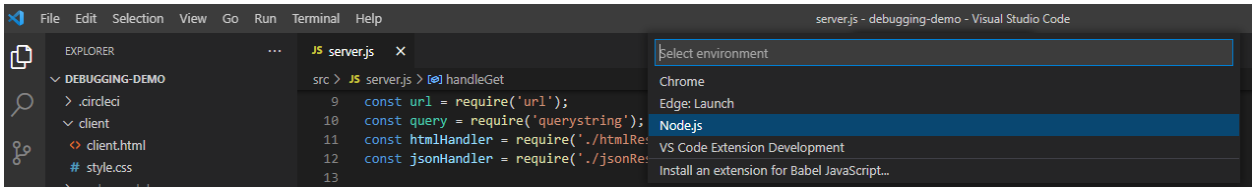
VS Code

Visual Studio Code (https://code.visualstudio.com/) is by no means the only text editor for Node development, nor is it “the best”. That being said, it has a lot of great features that set it apart from a fair bit of the competition. One of those features is it’s built in Node inspector/debugger.

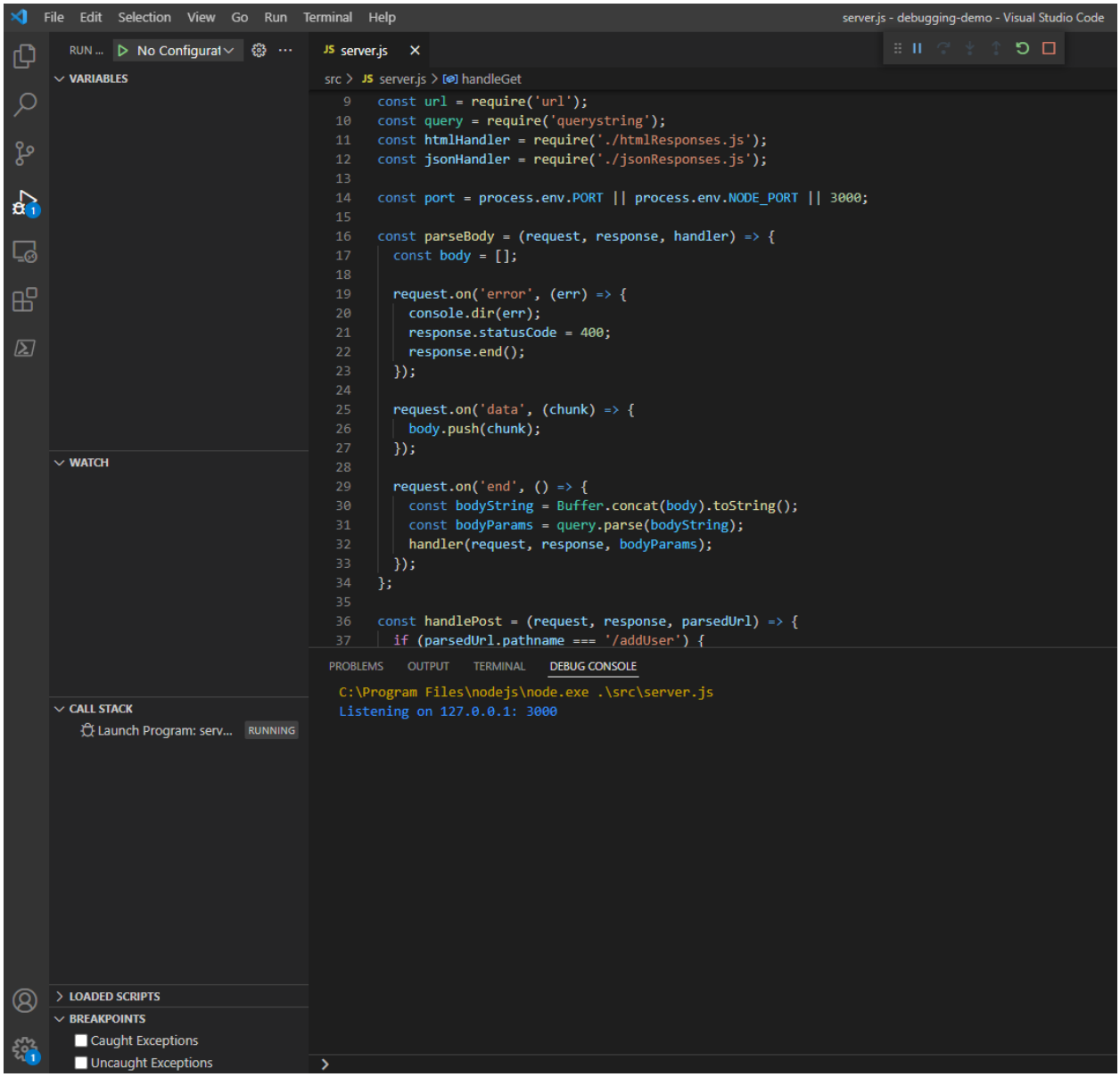
To begin debugging a Node project in VSCode, open the starting code file of your project (something like server.js). Then go to the “Run” menu at the top and select “Start Debugging”.



When the “Select Environment” dropdown appears, select “Node.js”.



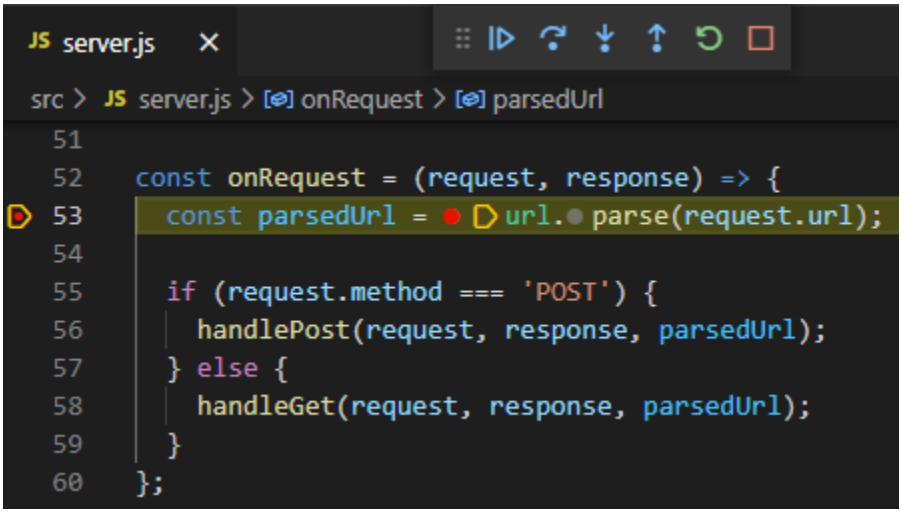
VSCode will then switch into the “Run and Debug” layout, where we have access to many familiar development tools such as the Variables window, the Watch window, and the Call Stack. Additionally, we can see the familiar “Pause”, “Step Over”, “Step Into”, and “Step Out” buttons. It is worth mentioning that at this point, VSCode has started our Node server for us by running the “node” command on the file we had open.



Now we can add things like breakpoints to my code by clicking just to the left of a specific line number.

```
52  const onRequest = (request, response) => {
53    const parsedUrl = url.parse(request.url);
54
55    if (request.method === 'POST') {
56      handlePost(request, response, parsedUrl);
57    } else {
58      handleGet(request, response, parsedUrl);
59    }
60  };
```

In this example, if we navigate to localhost:3000, the code will pause on the first line of onRequest. From there, we can use the step buttons to navigate through my code as we wish.



We can also see the values of every variable that exists on each line, and watch as they change and evolve over time as we move through the code. When we are ready to stop debugging, we can simply press the red square icon in the step function window.

Simply stepping through the code can oftentimes help you understand the flow of execution better than reading stack traces, etc.

Chrome / Chromium / Edge

As mentioned above, VSCode is not the only tool that exists for debugging Node. Node itself exposes debug tie-ins for other tools to use. Since browsers like Chrome and

Edge already have built in JavaScript debuggers, it was a natural leap for them to add support for Node.

To get started with them, we first need to add a debug script to our package.json.

```
"scripts": {
  "start": "node ./src/server.js",
  "pretest": "eslint ./src --fix",
  "test": "echo \"Tests complete\"",
  "debug": "node --inspect ./src/server.js"
},
```

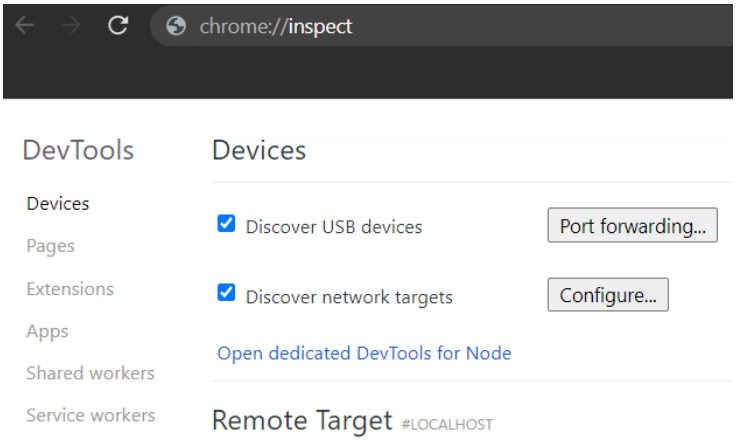
This debug script is very similar to the start script, but it includes the –inspect flag. This flag tells Node to allow external programs to monitor my server code execution. Now in Powershell we can run the command “npm run debug” to start my server in debug mode.

```
PS C:\Users\Austin\Downloads\debugging-demo> npm run debug

> simple_http@0.0.0 debug
> node --inspect ./src/server.js

Debugger listening on ws://127.0.0.1:9229/e00123c2-6811-4c81-9607-1bd519b1c85e
For help, see: https://nodejs.org/en/docs/inspector
Listening on 127.0.0.1: 3000
```

Now that my server is running in debug mode, we can attempt to hook in our browsers inspector. To do so, we can open the browser and navigate to the appropriate link.

Browser	**Link**
Chrome or Chromium	chrome://inspect
Edge	edge://inspect

That brings us to a screen that looks something like this:



On this screen we can click the “Open dedicated DevTools for Node” and select our node server. From that point, we can use the browsers development tools in the same way we would use the ones built into VSCode.

Server Console Window

Error Messages

The console window (Powershell, Terminal, etc) can provide us with quite a lot of information about what is going wrong when our server crashes or other exceptions are thrown by our server code.

For example, consider the following runtime error:

```
C:\Users\Austin\Downloads\debugging-demo\src\server.js:44
    htmlHandler.getCSS(request, response);
                ^

TypeError: htmlHandler.getCSS is not a function
    at handleGet (C:\Users\Austin\Downloads\debugging-demo\src\server.js:44:17)
    at Server.onRequest (C:\Users\Austin\Downloads\debugging-demo\src\server.js:58:5)
    at Server.emit (node:events:390:28)
    at parserOnIncoming (node:_http_server:951:12)
    at HTTPParser.parserOnHeadersComplete (node:_http_common:128:17)
```

Immediately upon reading this error message, we have a considerable amount of information to work with provided that we know what we are looking at.

The first line gives me a filepath, as well as a line number. In this case, the /src/server.js file at line 44.

```
src > JS server.js > ...
 44         htmlHandler.getCSS(request, response);
```

The second part of the error message tells us that we have a “TypeError”. A type error is a standard error type thrown by Node which tells us that we are trying to do something with the incorrect type. In this case, htmlHandler.getCSS() is apparently not a function.

Since we expect getCSS to be a function, we should look at where it is coming from. In this case, the htmlHandler is an import of the htmlResponses.js file. In that file we find the following.

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

module.exports = {
  getIndex,
};
```

getCSS is a function, but we never export it. That means that it does not exist within htmlHandler in our server.js file. We can resolve the issue by exporting the function.

Stack Traces

Oftentimes our server console will give us a cryptic error message and a large stack trace with which to work. Fortunately stack traces are incredibly useful, but they do require you to know what you are looking at.

Consider the following error and stack trace:

```
Error [ERR_STREAM_WRITE_AFTER_END]: write after end
    at new NodeError (node:internal/errors:371:5)
    at write_ (node:_http_outgoing:748:11)
    at ServerResponse.write (node:_http_outgoing:707:15)
    at respondJSON (C:\Users\Austin\Downloads\debugging-demo\src\jsonResponses.js:7:12)
    at addUser (C:\Users\Austin\Downloads\debugging-demo\src\jsonResponses.js:46:12)
    at IncomingMessage.<anonymous> (C:\Users\Austin\Downloads\debugging-demo\src\server.js:32:5)
    at IncomingMessage.emit (node:events:402:35)
    at endReadableNT (node:internal/streams/readable:1343:12)
    at processTicksAndRejections (node:internal/process/task_queues:83:21)
Emitted 'error' event on ServerResponse instance at:
    at emitErrorNt (node:_http_outgoing:726:9)
    at processTicksAndRejections (node:internal/process/task_queues:84:21) {
  code: 'ERR_STREAM_WRITE_AFTER_END'
}
```

The error, “write after end”, may be a little cryptic to the uninitiated. Similarly, a lot of information has been dumped to the console window here. So what does it all mean?

What we are looking at is a stack trace. It is a list of the code that has been recently run by the program that led us to the crash. The way we often want to read them is to begin at the top (the most recently run line of code) and look back at what events led us to the point of the error.

In this case, the first line says “new NodeError (node:internal/errors:371:5)”. This means that the error was fired by a NodeError object within the node:internals/errors file at line

371, column 5. Not overly helpful for us in our attempt to resolve the issue. So we continue reading back.

The next few lines might give us some insight into what is going wrong. For example, something related to the http_outgoing library is going wrong on the write_ and write functions.

However, where we really want to look is the first line that we wrote. In this case, Powershell has conveniently highlighted using white text instead of gray text. This first line points us to “src/jsonResponses.js:7:12”, or jsonResponses line 7 column 12. Let’s take a look there.

```
5   const respondJSON = (request, response, status, object) => {
6     response.writeHead(status, { 'Content-Type': 'application/json' });
7     response.write(JSON.stringify(object));
8     response.end();
9   };
```

This is our respondJSON function, which works for other responses in the server. It is unlikely that the error is happening here. Let’s continue looking. The next line of the stack trace points to jsonResponses.js line 46 column 12.

```
43    if (responseCode === 201) {
44      responseJSON.message = 'Created Successfully';
45      respondJSON(request, response, responseCode, responseJSON);
46      return respondJSON(request, response, responseCode, responseJSON);
47    }
```

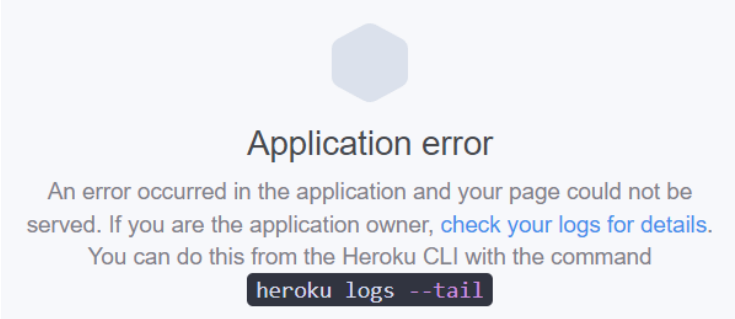
Here we find the culprit. We are calling respondJSON twice. Because each HTTP request can only receive one response, we get an error on line 46 because we try to send a response that has already been sent. Removing the code on line 45 will resolve the issue.

Heroku CLI

There are often occasions where your code works locally, but stops working when deployed to heroku.

For example if you are using a Windows machine and create a file called “style.css”, you can load that file at the paths “style.css” or “Style.css”. Realistically any case will work, because the Windows file system is not case sensitive. However, Heroku deploys your apps using Linux which is case sensitive. In that case, code that loads “style.css” as “Style.css” will error out.

To be able to troubleshoot these, you will want to use the Heroku CLI to view the full logs of your heroku application as if you were looking at the console window for the server.



To get the Heroku CLI (Command Line Interface), visit this link: https://devcenter.heroku.com/articles/heroku-cli

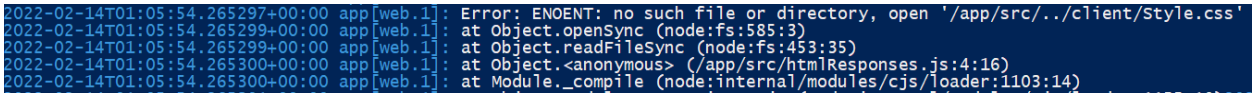
Once you have installed the Heroku CLI, you can access it through terminal windows like Powershell. Begin by running the “heroku login” command. Heroku will open a window in your default browser and authenticate your login through their website. Click the “Login” button, and return to the Powershell window.

You are now logged in to the Heroku CLI. That means you can run Heroku commands from this command window. For example, if you want to get the full log information related to the error on your server, run the following command with YOUR_APP_NAME replaced with the name of your heroku app.

```
heroku logs -a YOUR_APP_NAME
```

For example, if the app is called “debugging-node” you would run the command “heroku logs -a debugging-node”.

Once that command runs, you should get a full output of the logs from your app on heroku. For example, here is one showing the “style.css” vs “Style.css” error.



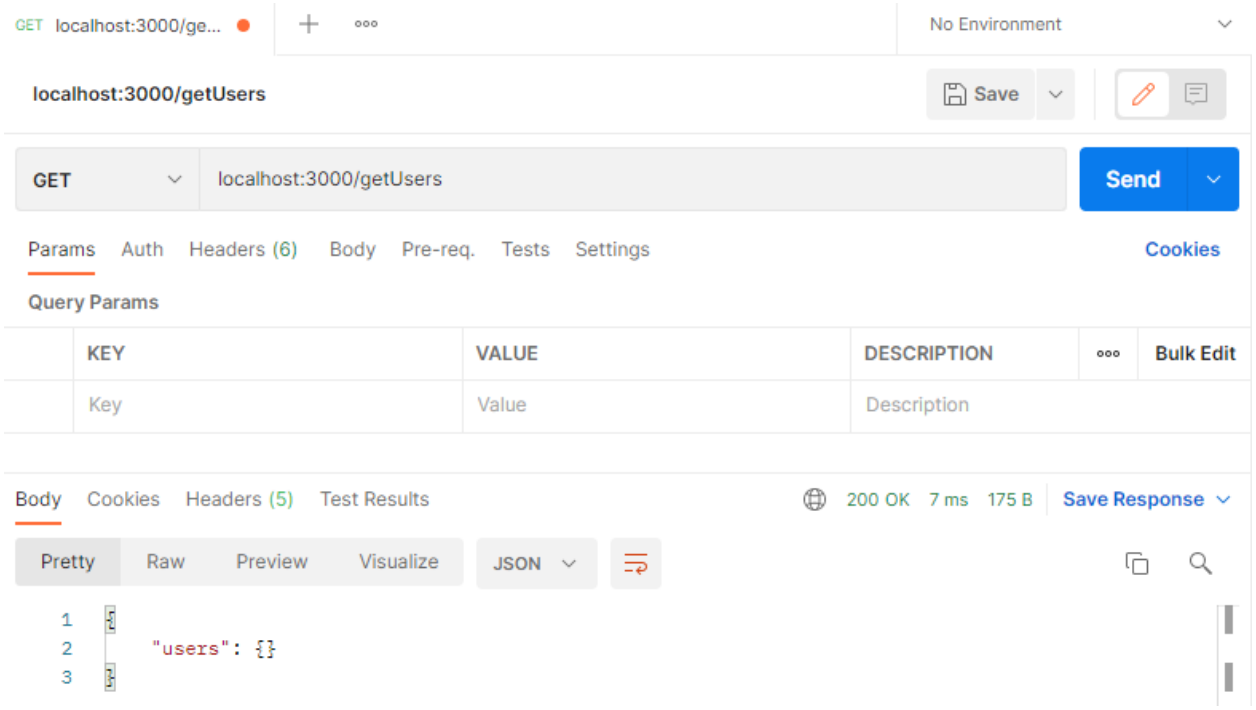
Postman

Although it is always possible to write a client application that can interact with your backend server, that is often overkill. If you are developing a data API that other developers can use, there is a good chance you don't want to waste time making a client web page to interact with it.

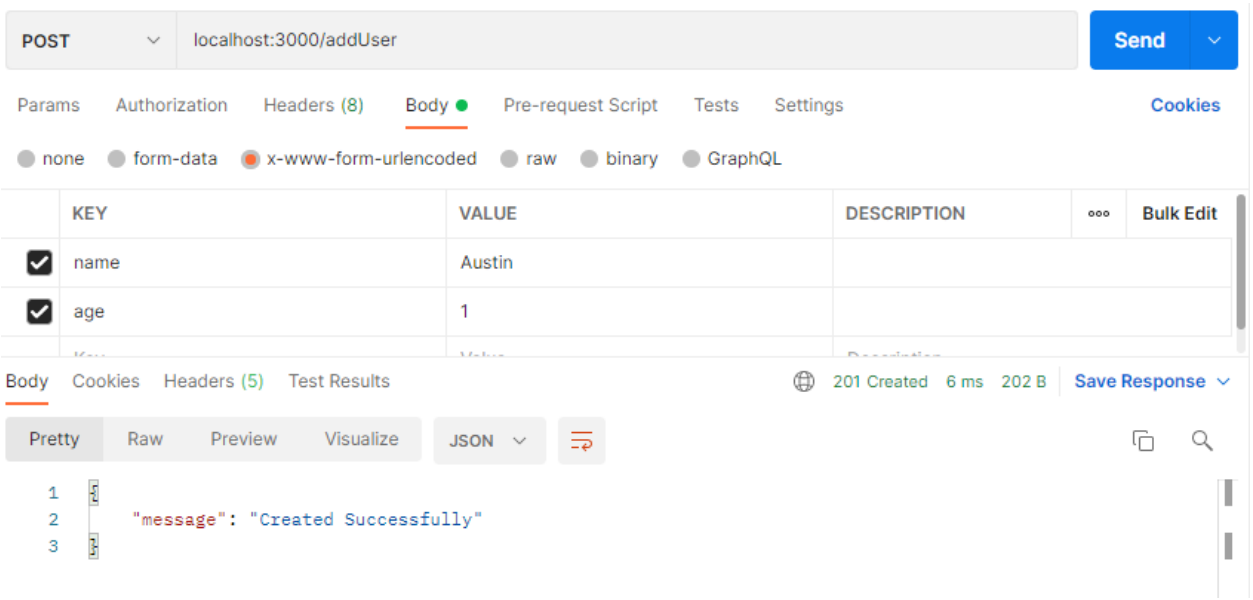
Luckily there are tools like Postman (https://www.postman.com/) that are designed to help you test your server's endpoints and validate that they work properly. Once you create an account and download Postman, you can begin using it to test your local server. You can also use it to test remote servers like your heroku apps.

The application has a built-in tutorial, so there will not be much detail in this document. However, below are a few examples.

GET requests:



POST requests:


POST
localhost:3000/addUser
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
KEY
VALUE
DESCRIPTION
Bulk Edit
name
Austin
age
1
Body
Cookies
Headers (5)
Test Results
201 Created
6 ms
202 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
1 {
2 "message": "Created Successfully"
3 }