# Servers and ExpressJS!

The way that we regularly interact with the internet is through user interfaces, however, a vast majority of programs using the internet do it entirely through sending pure data back and forth! Truthfully, HTML is just more data, however, our web browsing software (e.g. Chrome, Safari, Edge, etc.) understands the language that's sent back, and renders it in a way that's meaningful to us.

## TCP/Telnet:

Transmission Control Protocol (TCP) is a protocol that establishes how applications can connect and speak to each other over the internet. We won't dig too much into this (not a networking class). By using a tcp client, we can see see exactly how data are sent back and forth via the command line. The tool we'll use for that is telnet, which uses the TELNET protocol over TCP. For all intents and purposes, we're just making network calls with our command line.

For linux users, it already exists on your machine in your terminal. If you do not have it on your machine, you can install it with:

```
apt-get install telnet
```

For windows users, you don't need to install it specifically, but you'll need to enable it. To do so, follow these instructions. For mac users, you'll likely need to install it via a package manager like homebrew:

```
brew install telnet
```

#### Establishing a Connection:

Once you have telnet installed, you can start to establish a connection. You'll need to write:

```
telnet HOSTNAME/IP_ADDRESS PORT_NUMBER
```

Let's visit my githubio page with telnet (And use port 80):

```
telnet lanematthewj.github.io 80
```

Once we open this, notice that nothing happens. That's because we haven't actually said exactly what we want to do (other than establish a connection). Now we need to provide it with the call we'd like to make:

```
GET / HTTP/1.1
Host: lanematthewj.github.io
```

What we're doing above is saying we'd like to:

- `GET` - whatever http method we'd like to call.
- `/` - the path of the resource (think of this as the url path)
- `HTTP/1.1` - the version of http we want to specify.
- `Host: lanematthewj.github.io` - an IP address can host multiple domains. We need to specify which host we want!

When you make this request, you won't see anything yet. Hit enter a couple times to signify that you're finished with your headers / body of your **request**, then you should see a **response** something akin to:

```
HTTP/1.1 200 OK
Server: GitHub.com
Content-Type: text/html; charset=utf-8
Last-Modified: Wed, 22 Jun 2016 22:35:10 GMT
ETag: "576b129e-1cf5"
Access-Control-Allow-Origin: *
Expires: Mon, 30 Mar 2020 02:26:37 GMT
Cache-Control: max-age=600
X-Proxy-Cache: MISS
X-GitHub-Request-Id: 37BC:4B65:450016:5722BA:5E815683
Accept-Ranges: bytes
Date: Mon, 30 Mar 2020 02:40:09 GMT
Via: 1.1 varnish, 1.1 localhost01.localdomain
Age: 0
X-Served-By: cache-dal21228-DAL
X-Cache: MISS
X-Cache-Hits: 0
X-Timer: S1585536009.370735,VS0,VE41
Vary: Accept-Encoding
X-Fastly-Request-ID: ac77ff48fe8ee78b1b1fabddd2d0f192be0d2dfb
Content-Length: 7413
Connection: keep-alive

<!DOCTYPE html>
<html lang="en">
  <!-- Beautiful Jekyll | MIT license | Copyright Dean Attali 2016 -->
```

```
  <head>
  <meta charset="utf-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0">

  .... boring stuff ...

</html>
```

In the response, we get a whole ton of information back. Because we're in the command line, we don't see a beautifully rendered website. We instead see a bunch of HTML and a bunch of response headers (those lines above the HTML). The HTML we see is the response body (typically what we see as a rendered website)!

### Calling an API

However, since much of the internet is just data transferring back and forth, let's try calling an API that isn't going to return HTML. This time, though, let's use a route.

```
telnet ron-swanson-quotes.herokuapp.com 80
GET /v2/quotes HTTP/1.1
Host: ron-swanson-quotes.herokuapp.com
```

Notice above, instead of writing a `/` for our route to the index, we specified that we'd like to retrieve something from `/v2/quotes`. Because this endpoint returns a different quote every time, your response won't look exactly like this, but will be close enough:

```
HTTP/1.1 200 OK
Server: Cowboy
X-Powered-By: Express
Access-Control-Allow-Origin: *
Content-Type: application/json; charset=utf-8
Etag: W/"33-b65a0711"
Date: Mon, 30 Mar 2020 02:14:56 GMT
Via: 1.1 vegur, 1.1 localhost01.localdomain
Content-Length: 51
Age: 2250
Connection: keep-alive

["Never half-ass two things. Whole-ass one thing."]
```

Notice that in this response, we didn't receive any HTML. We only received data (in the form of a single element string array).

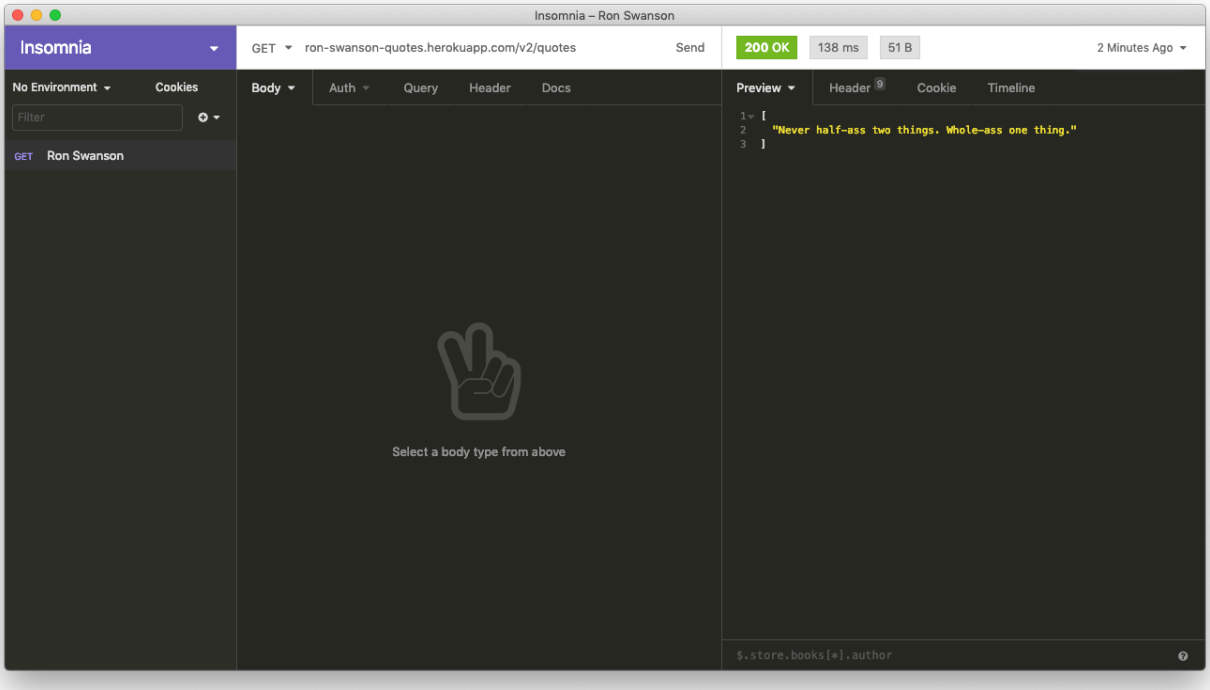
## Applications for Not Telnet:

Using telnet is great to understand exactly how data are sent back and forth, however, it's older and not really that intuitive to deal with. There are newer applications that do the exact same thing, such as Postman or Insomnia. Both are great. Use whichever you feel more comfortable with, since they ultimately do mostly the same things!

### Making a Call

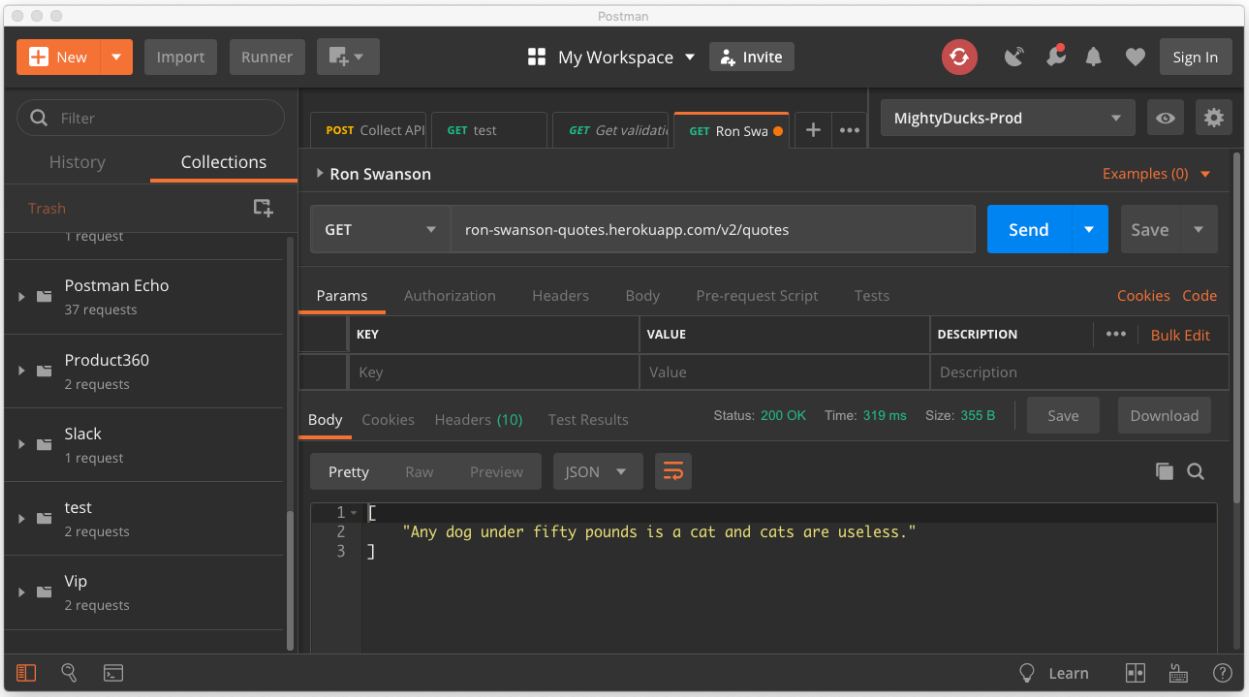
To make a call similar to how we did with telnet, we can simply just create a new call with `ctrl + n` or `cmd + n`. Each application has a space to enter in your requested URL. You can enter the path directly in, so you don't need to specify a hostname with an additional path to the desired resource. Additionally, right next to the url on both clients is a button specifying which request method you would like to use. Take. a look at how each application does calls `ron-swanson-quotes`:

**Insomnia:**




Insomnia has its request data on the left, and its response data on the right. You can view the response body as a "preview" or even as "raw data".

**Postman**




Postman's UI is a bit more arcane in that it takes a moment to get used to, however, it's pretty much the exact same as insomnia, except for that the request is on the top, and the response is below it.

By using these applications for network calls, you can test api's that you're developing and save your calls instead of typing everything out via telnet.

For the sake of the notes, you'll be seeing insomnia (since it has better screenshot capabilities).

## HTTPServer:

Having the tools to test an api is great, but knowing how to code one is why we're here! Let's start with an incredibly simple server (not using express). First, let's create a new node project (without a starter kit).

```
mkdir simple-server && cd simple-server
npm init -y
```

Notice that in our package json, our main entry point is `index.js`. Let's add that file and write some code:

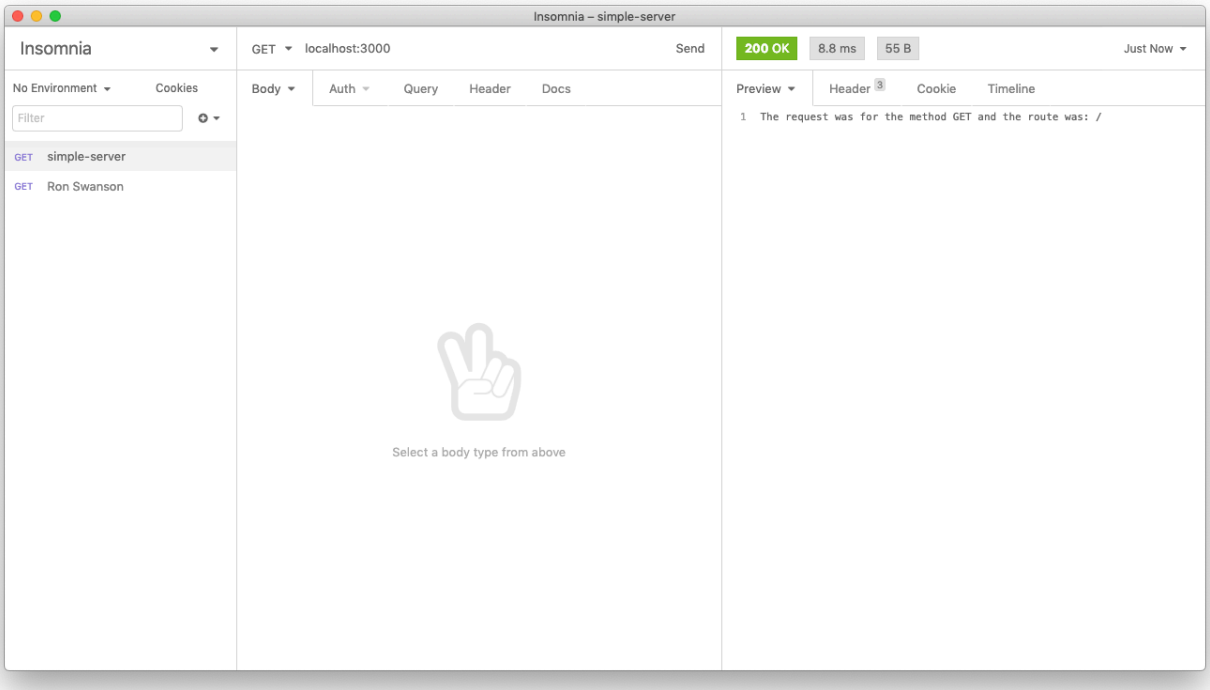
```
const http = require("http");

const server = http.createServer((req, res) => {
  console.log("The whole request was: " + req)

  const method = req.method;
  const route = req.url;

  res.end(
    "The request was for the method " + method + " and the rout was: " +
route
  );
});


const port = 3000;
server.listen(port);
console.log("Now listening on port " + port);
```

Now, let's start the server. We can start our server simply by typing `node index.js`:

```
(base) ➜  simple-server node index.js
Now listening on port 3000
```

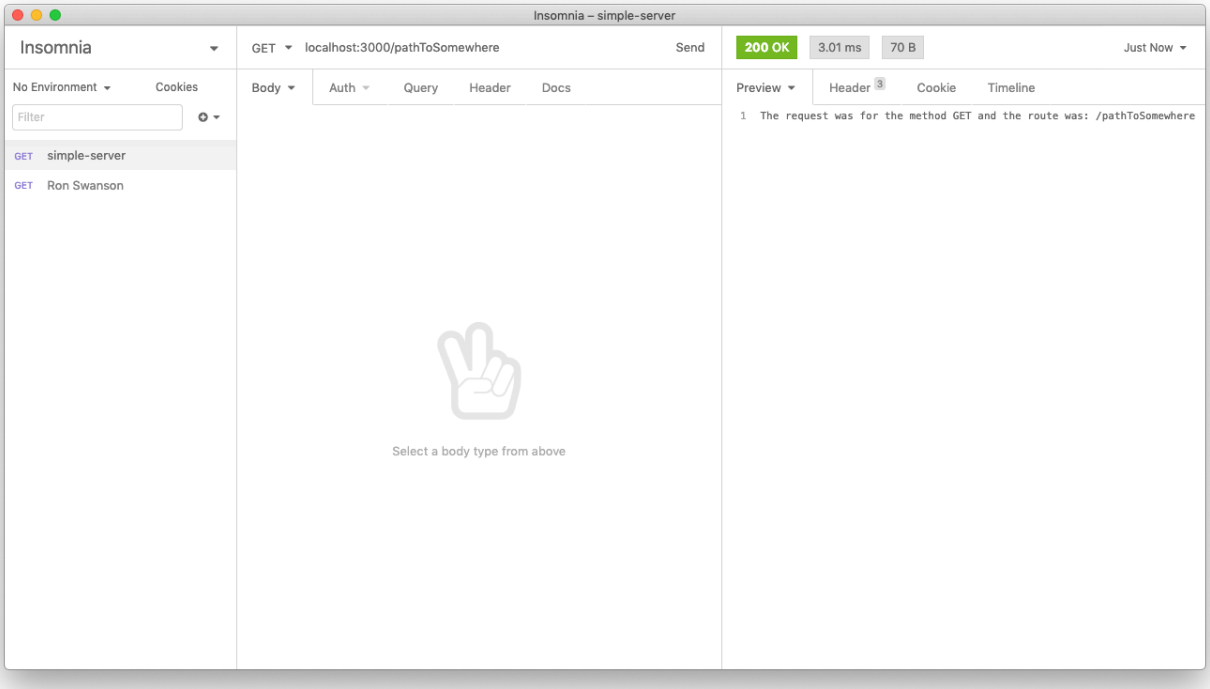
**NOTE**: Before we go any further, here's a quick note: Up to now, we've been using the default project space from `create-react-app`. When you made changes in there, the project immediately updated. Because we're starting from scratch here, you'll need to restart the server *every* time you make a change!

When we make a call with insomnia, we can see that we get the response string we had intended (the insomnia theme changed to white!):




You can see that we got the string: `The request was for the method GET and the route was: /` back. This was done by using our `res` object. Think of the `res.end` as being a way for us to write material to the response object. Ultimately the response object is a blank canvas for us to fill (we'll flesh this out a bit more later on).

We can add a specific path to our call too:



Note that the path gets registered as well in `req.url`, we get the response of: `The request was for the method GET and the route was: /pathToSomewhere`.

This response data is fine and great, but we also console.logged some information. Where did that go? Unlike front end software, since our program is running as a server, and not being sent to be run on a user's browser, the console logs to the environment where we're running our server (aka the terminal where we wrote `node index.js`).

When checking the terminal, though, notice that we're only seeing that our request console log prints out: `The whole request was: [object Object]`. This is not entirely helpful. Often times when you wish to view data that comes in object form, you'll use `JSON.stringify`. That will crash your server because req has circular references. You'll wind up with an error like:

```
TypeError: Converting circular structure to JSON
```

Not that you regularly will, want to print out the requests, but it's good to see what's coming in. To do so, we'll want to import the `util` package, which has an `inspect` feature, which we'll want to use to parse our `req`:

```
const http = require("http");
const util = require("util");

const server = http.createServer((req, res) => {
  const inspectedReq = util.inspect(req)
  console.log(inspectedReq);
  const method = req.method;
  const route = req.url;

  res.end(
    "The request was for the method " + method + " and the route was: " +
route
  );
});

const port = 3000;
server.listen(port);
console.log("Now listening on port " + port);

```

Now, when we run our code, we're able to see the whole of our request. There is a ton of information there (around 600+ lines):

```
IncomingMessage {
  _readableState:
   ReadableState {
```

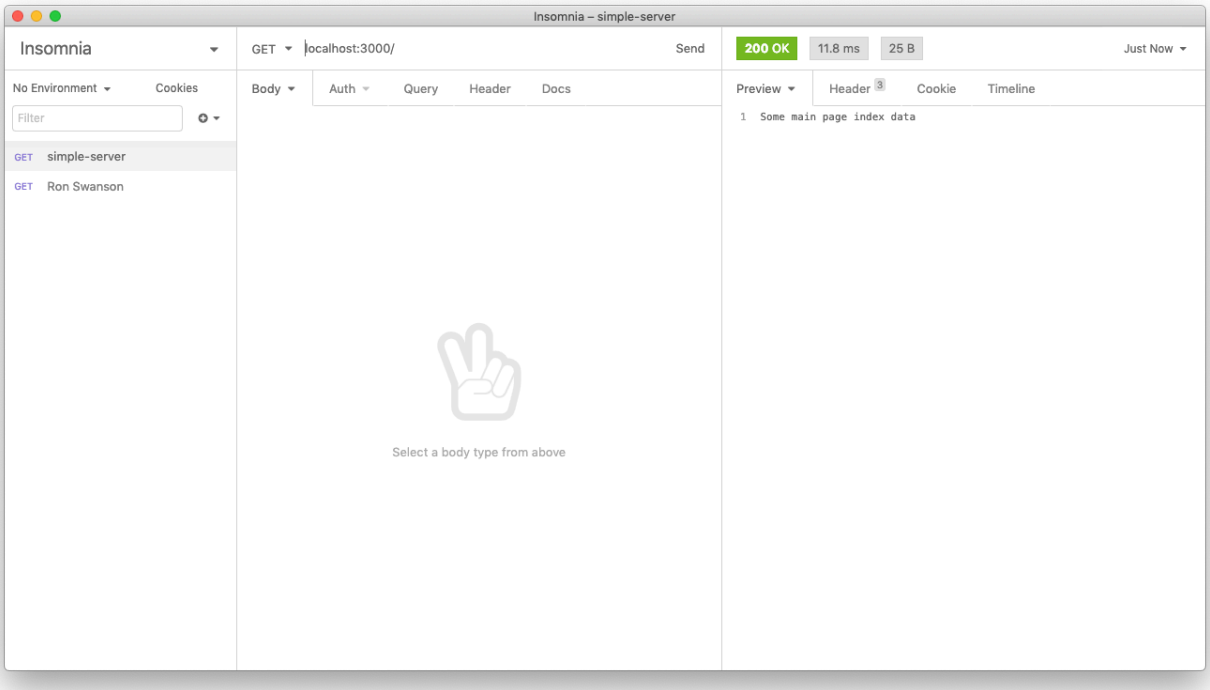
```
    objectMode: false,
    highWaterMark: 16384,
    buffer: BufferList { head: null, tail: null, length: 0 },
    length: 0,
    pipes: null,
    pipesCount: 0,
    flowing: null,
    ended: false,
    endEmitted: false,
    reading: false,
    sync: true,
    needReadable: false,
    emittedReadable: false,
    readableListening: false,
    resumeScheduled: false,
    paused: true,
    emitClose: true,
    autoDestroy: false,
    destroyed: false,
    defaultEncoding: 'utf8',
    awaitDrain: 0,
    readingMore: true,
    decoder: null,
    encoding: null },
  readable: true,

  ...

  trailers: {},
  rawTrailers: [],
  aborted: false,
  upgrade: false,
  url: '/',
  method: 'GET',
  statusCode: null,
  statusMessage: null,

  ...

  _consuming: false,
  _dumped: false }
```

There is a ton of data. We don't honestly care about most of it. However, we do are about `url` and `method`, which we can see are in the request (and which we already know are working because we can see it in the response).

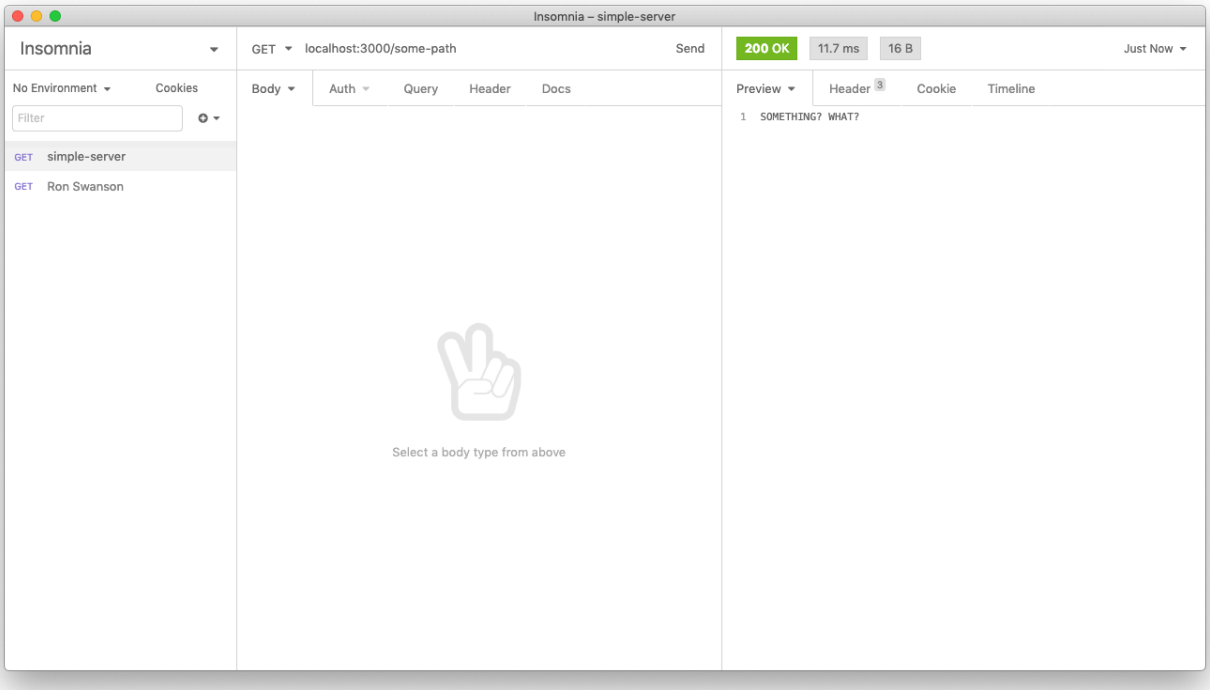
As of now, our server isn't all too powerful. It's just recognizing what's coming in, and printing the route and the method. That function which is logging our request and returning a string is the powerhouse of the server! It's ultimately a question how in depth we want to make it! Let's spruce our function up to be a bit more powerful (and let's format it so it's no longer a parameter to another function AND remove that gargantuan request log):

```
const http = require("http");

const serverFunction = (req, res) => {
  const method = req.method;
  const route = req.url;

  if (route === "/" && method === "GET") {
    res.end("Some main page index data");
  } else if (route === "/some-path" && method === "GET") {
    res.end("SOMETHING? WHAT?");
  }
  res.end("Where were you going?");
};

const server = http.createServer(serverFunction);

const port = 3000;
server.listen(port);
console.log("Now listening on port " + port);

```

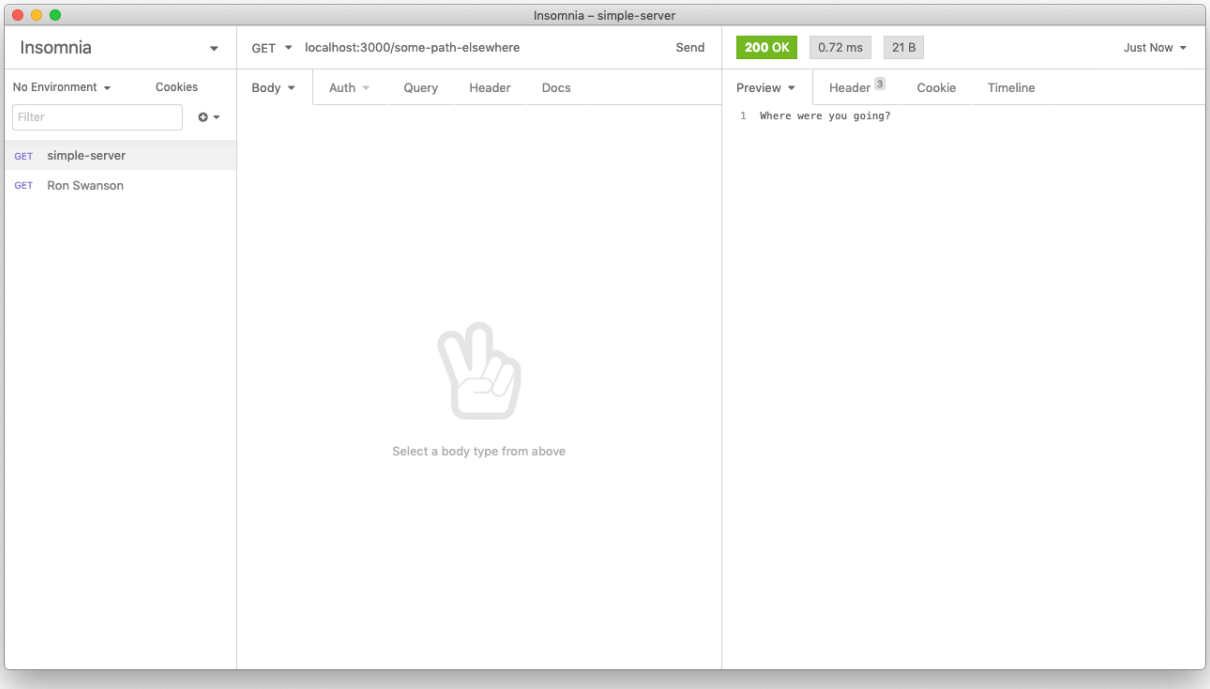
Now let's give this a go, first with our index:




Naturally, we got to our indexed path! You may be wondering why we didn't see "Where were you going?". We don't see the note send from `res.end()` at line 13 because `res.end()` is essentially what kicks off the response and ends the session. Let's try it with our other path `some-path`:



There we go! We got `SOMETHING? WHAT?` as a response! And finally, let's see it with a path that doesn't match:




There we go. We got our default response.

## Automating Server Code Updates

Presumably, by now you're very tired of stopping your server, and then restarting it just for a minor change. Luckily for us, there are tools to do that for us! One of the most popular tools out there is called `nodemon`. Nodemon is an automation utility that is often used for reloading a project, but can be used for any number of automation reasons you have when restarting a project. Let's install it locally with:

```
npm i --save-dev nodemon
```

Note what we did above. We used the flag `--save-dev`. This flag places dependencies as "developer dependencies". Get used to doing this for tools that you yourself use, but are not necessarily useful for production code (e.g. things like automation software for restarting a server when changing code, formatting software, testing software, etc).

There are multiple ways to run nodemon. The documentation suggests installing nodemon globally, but it's always a good idea to install packages locally so that when someone else goes to run your project, they don't find themselves missing something (even if it is just a script).

When you want to run a package locally, you can't simply type the name of the package like the documentation would suggest, otherwise you'll get an error (that is, unless you installed it globally):

```
(base) ➜  simple-server nodemon index.js
zsh: command not found: nodemon
```

What we want to do is use `npx`:

```
(base) ➜  simple-server npx nodemon index.js
[nodemon] 2.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Now listening on port 3000
```

There we go. Now any time a file linked within our project with a .js, .mjs, or .json extension changes, nodemon will restart the project.

Our start script is getting a little long, however, so it might just be easier to write our own start script. We can do this in the package.json. Let's change our package json to have a `start` script that we can run instead of typing out `npx nodemon index.js`:

```
{
  "name": "simple-server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "npx nodemon index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "Matt Lane <mjlane@monsanto.com>",
  "license": "ISC",
  "devDependencies": {
    "nodemon": "^2.0.2"
  }
}

```

Now when you run your server, you can just write `npm run start` , and `npx nodemon index.js` will get executed. Anything you want to put into your `scripts` you can run with `npm run <SCRIPT_NAME>`.

When we run our code with `npm run start` we'll then get an output that looks something like this:

```
(base) ➜  simple-server npm run start

> simple-server@1.0.0 start
/Users/mjlane/Projects/notes4011/modules/Express/module1Servers/simple-server
> npx nodemon index.js

[nodemon] 2.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Now listening on port 3000
```

Now that we've set up our program to autoupdate every time we change a file, let's make some major changes!

## Express

We've set up our base server with `http.createServer`. Express essentially sits on top of `http.createServer`. First, though, we'll need to install express:

```
npm i --save express
```

Now that we've saved express (note that we didn't save it as a developer dependency, but a regular dependency), we can import it and use it in our application:

```
const http = require("http");
const express = require('express')

const app = express()

app.use((req, res) => {
  const method = req.method;
  const route = req.url;

  if (route === "/" && method === "GET") {
    res.send("Some main page index data");
  } else if (route === "/some-path" && method === "GET") {
    res.send("SOMETHING? WHAT?");
  }
  res.end("Where were you going?");
})

```

```
const port = 3000;
app.listen(port)
console.log("Now listening on port " + port);
```

We can take our callback and place it inside of our `app.use`, and instead of creating an HTTP server, we just use `app.listen`. Additionally, notice that instead of using `res.end`, we're using `res.send`. Express can just as easily use `res.end`, but by using `res.send`, you gain some extra benefits from Express, such as the automatic conversion from javascript objects to JSON:

```
const http = require("http");
const express = require('express')

const app = express()

app.use((req, res) => {
  const method = req.method;
  const route = req.url;

  if (route === "/" && method === "GET") {
    const someObject = {
      hi: "there",
      whats: "up"
    }
    res.send(someObject);
  } else if (route === "/some-path" && method === "GET") {
    const otherObject = {
      not: "much"
    }
    res.send(otherObject);
  }
  res.end("Where were you going?");
})

const port = 3000;
app.listen(port)
console.log("Now listening on port " + port);
```

If we had tried that with plain `http.createServer` we would've had to use `JSON.stringify` on the objects before sending them off.

Ultimately, Express just sits on top of the http module, the following code also works completely fine (though is a little clunky with the mixture of `http` and `express`):

```
const http = require("http");
const express = require('express')

const app = express()

app.use((req, res) => {
  const method = req.method;
  const route = req.url;

  if (route === "/" && method === "GET") {
    res.end("Some main page index data");
  } else if (route === "/some-path" && method === "GET") {
    res.end("SOMETHING? WHAT?");
  }
  res.end("Where were you going?");
})

const server = http.createServer(app);

const port = 3000;
server.listen(port);
console.log("Now listening on port " + port);
```