CSIS 3380 – Nodejs Notes

Node.js is a platform for interpreting JavaScript code and running applications.

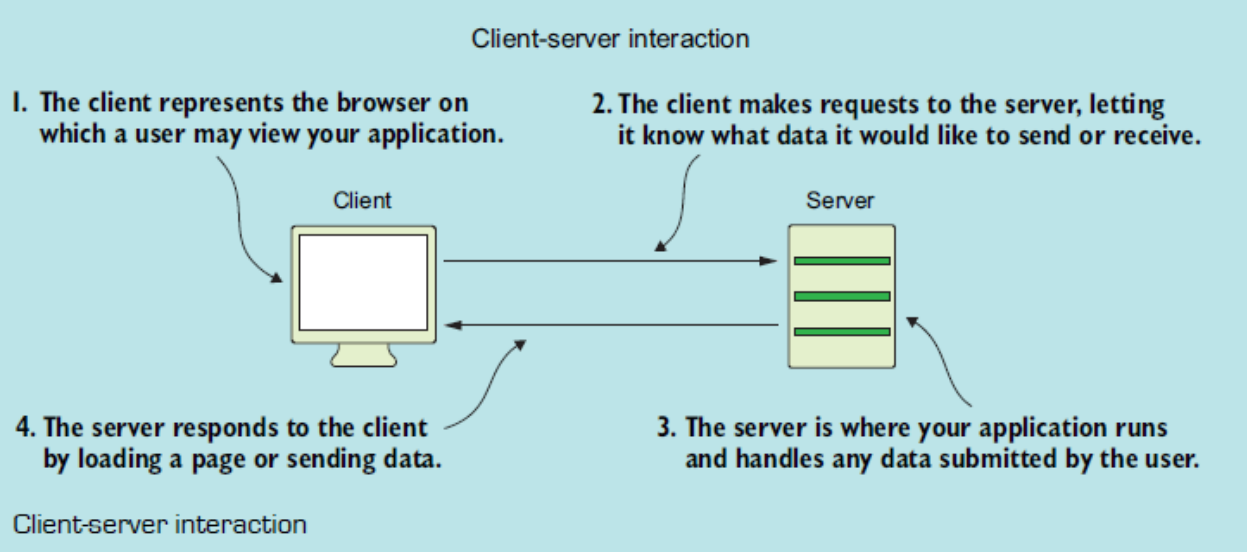
Node.js is built with Google Chrome's JavaScript engine, and it's considered to

be powerful and able to support JavaScript as a server-side language.

Client-side versus server-side

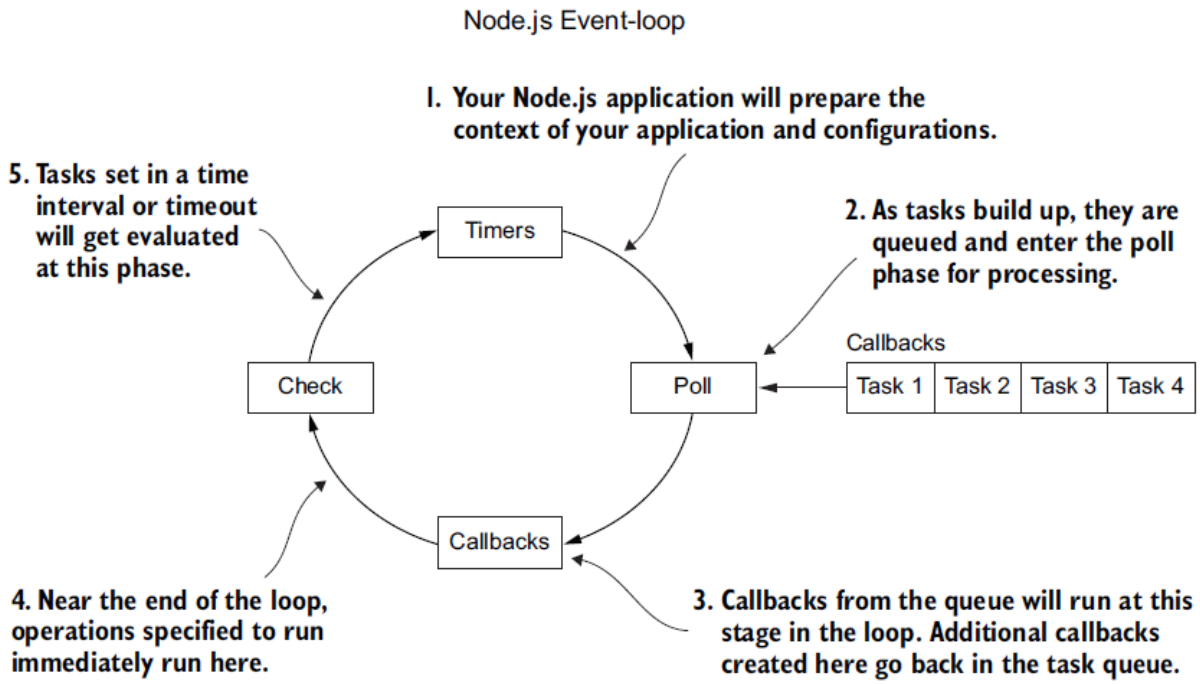
As a general overview, web development can largely be broken into two categories:

- ☐ *Client-side*—(front-end) refers to the code you write that results in something the user sees in his web browser. Client-side code typically includes JavaScript used to animate the user experience when a web page is loaded.
- ☐ *Server-side*—(back-end) refers to code used for application logic (how data is organized and saved to the database). Server-side code is responsible for authenticating users on a login page, running scheduled tasks, and even ensuring that the client-side code reaches the client.



You'll hear these two terms used a lot in application development, and because JavaScript has been used for both types of development, the line separating these two worlds is disappearing. *Full stack* development, using JavaScript, defines this new development in which JavaScript is used on the server and client, as well as on devices, hardware, and architectures it didn't exist on before.

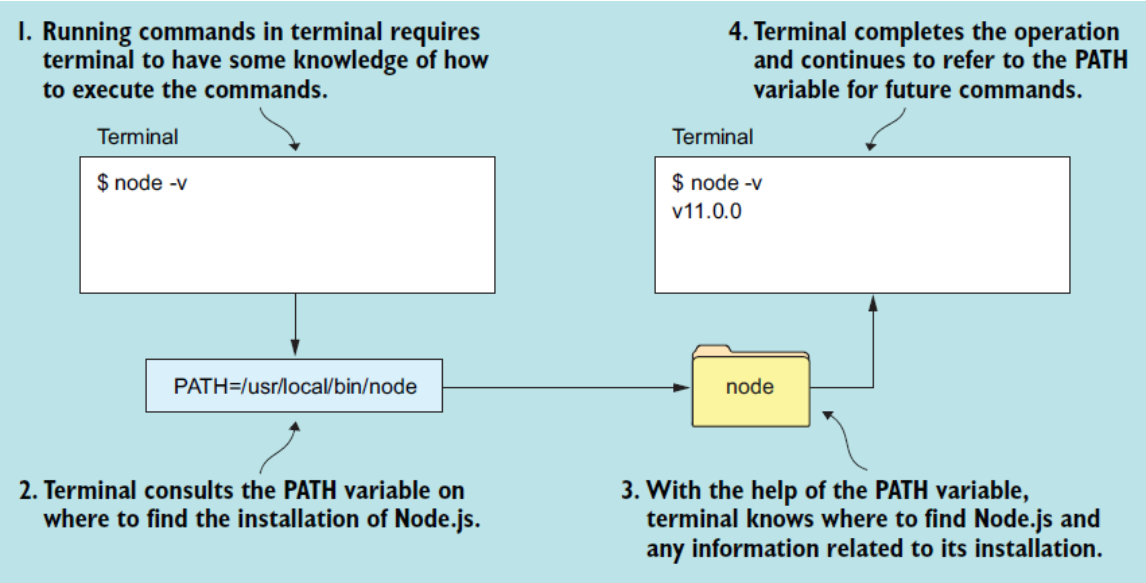
Node.js operates on an event loop using a single thread. A *thread* is the bundle of computing power and resources needed for the execution of a programmed task. In most other software, multiple tasks are matched and handled by a pool of threads that the computer can offer at the same time (concurrently). Node.js, however, handles only one task at a time and uses more threads only for tasks that can't be handled by the main thread. Most applications that don't require computationally intensive tasks



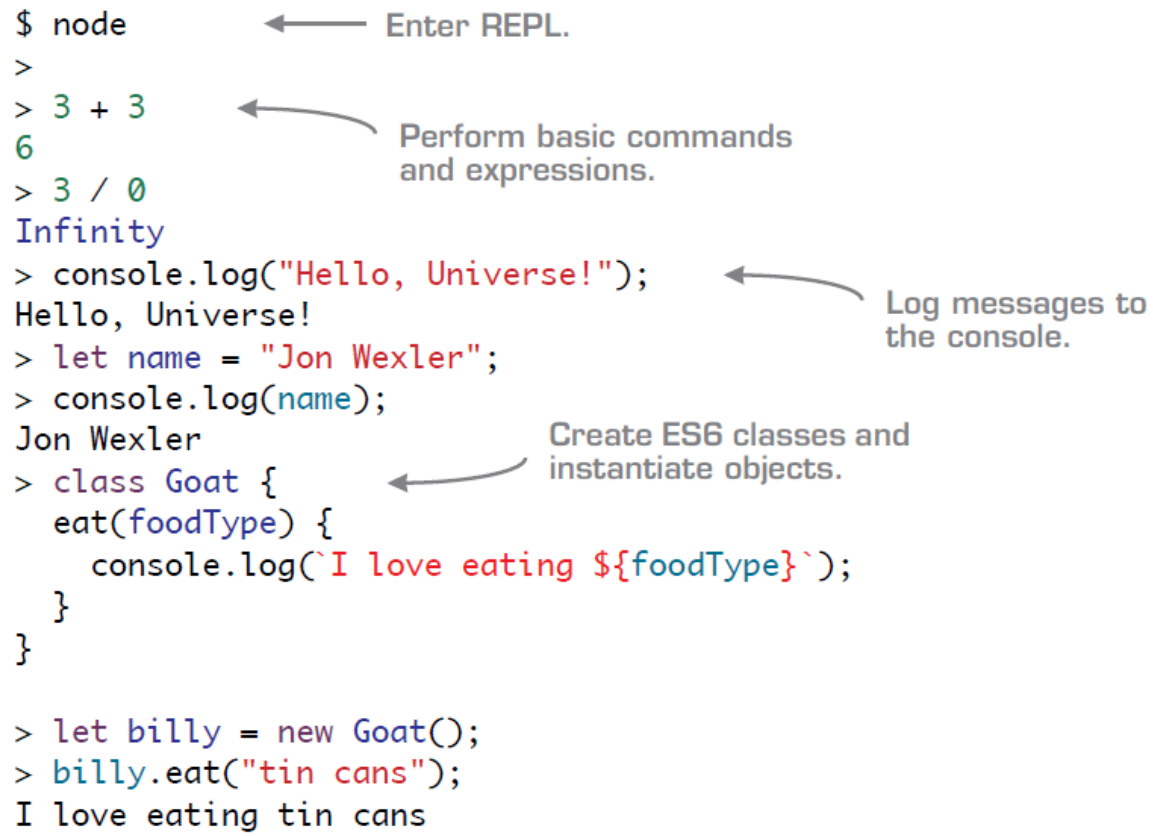
Installing Node.js

The simplest way to install Node.js is to go to the download link at https://nodejs.org/ en/download/ and follow the instructions and prompts to download the installer for the latest version of Node.js.

If you experience any problems starting Node.js in your terminal, follow the installation steps at https://www.tutorialspoint.com/nodejs/nodejs_environment_setup.htm.



REPL command examples



REPL commands to remember

REPL command	Description
`.break (or .clear)`	Exits a block within the REPL session, which is useful if you get stuck in a block of code
`.editor`	Opens an internal editor for you to write multiple lines of code. `ctrl-d` saves and quits the editor
`.exit`	Quits the REPL session

Running your JavaScript file with Node.js

The Node.js JavaScript engine can interpret your JavaScript code from the terminal when you navigate to the location of a JavaScript file and preface the filename with the `node` keyword.

Complete the following steps to test run your JavaScript file:

Create a file named hello.js and insert the following code:

```
console.log("Hello, Universe!");
```

1. Open a new terminal window.
2. Navigate to your desktop by entering `cd ~/Desktop`.
3. Run your JavaScript file by entering the `node` keyword followed by the file's name. You can also run the same command without the file's extension. Type `node hello` at the prompt, for example, for a file named hello.js

Running individual Javascript Commands

Create a file named **messages.js**
Enter the following

Messages.js

```
let messages = [
"A change of environment can be a good thing!",
"You will make it!",
"Just run with the code!"
];
```

Instead of executing the file, use REPL.

```
> .load messages.js
```

Then

```
messages.forEach(message => console.log(message));
```

Exercise:

Try building a file called `printer.js` with the code in the next listing inside.

```
let x = "Universe";
console.log(`Hello, ${x}`);
```

A Node.js application is made up of many JavaScript files. For your application to stay organized and efficient, these files need to have access to one another's contents when necessary. Each JavaScript file or folder containing a code library is called a *module*.

With your installation of Node.js, you also got npm, a package manager for Node.js. npm is responsible for managing the external packages (modules that others built and made available online) in your application.

Throughout application development, you use npm to install, remove, and modify these packages. Entering npm -l in your terminal brings up a list of npm commands with brief explanations.

npm commands to know

npm command	Description
npm init	Initializes a Node.js application and creates a package.json file
npm install <package>	Installs a Node.js package
npm publish	Saves and uploads a package you build to the npm package community
npm start	Runs your Node.js application (provided that the package.json file is set up to use this command)
npm stop	Quits the running application
npm docs <package>	Opens the likely documentation page (web page) for your specified package

When you use npm install <package>, appending --save to your command installs the package as a dependency for your application. you'll use npm install express -S to install the Express.js framework for your project.

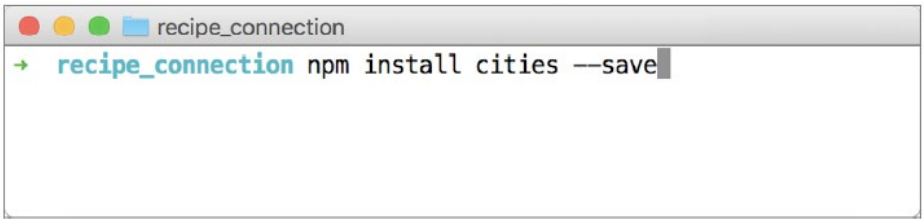
Every Node.js application or module contains a package.json file to define the properties of that project. This file lives at the root level of your project. Typically, this file is where you specify the version of your current release, the name of your application, and the main application file.

To get started, create a folder, navigate to your project directory in terminal, and use the npm init command to initialize your application. You'll be prompted to fill out the name of your project, the application's version, a short description, the name of the file from which you'll start the app (entry point), test files, Git repositories, your name (author), and a license code.

For now, be sure to enter your name, use main.js as the entry point, and press Enter to accept all the default options. When you confirm all these changes, you should see a new package.json file in your project directory.

```
{
  "name": "firstApp",
  "version": "1.0.0",
  "description": "",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "RG",
  "license": "ISC"
}
```

Now your application has a starting point for saving and managing application configurations and packages. Now install a third party package that is available, “cities”. This package help you find location properties of a place in the USA from its ZIP co

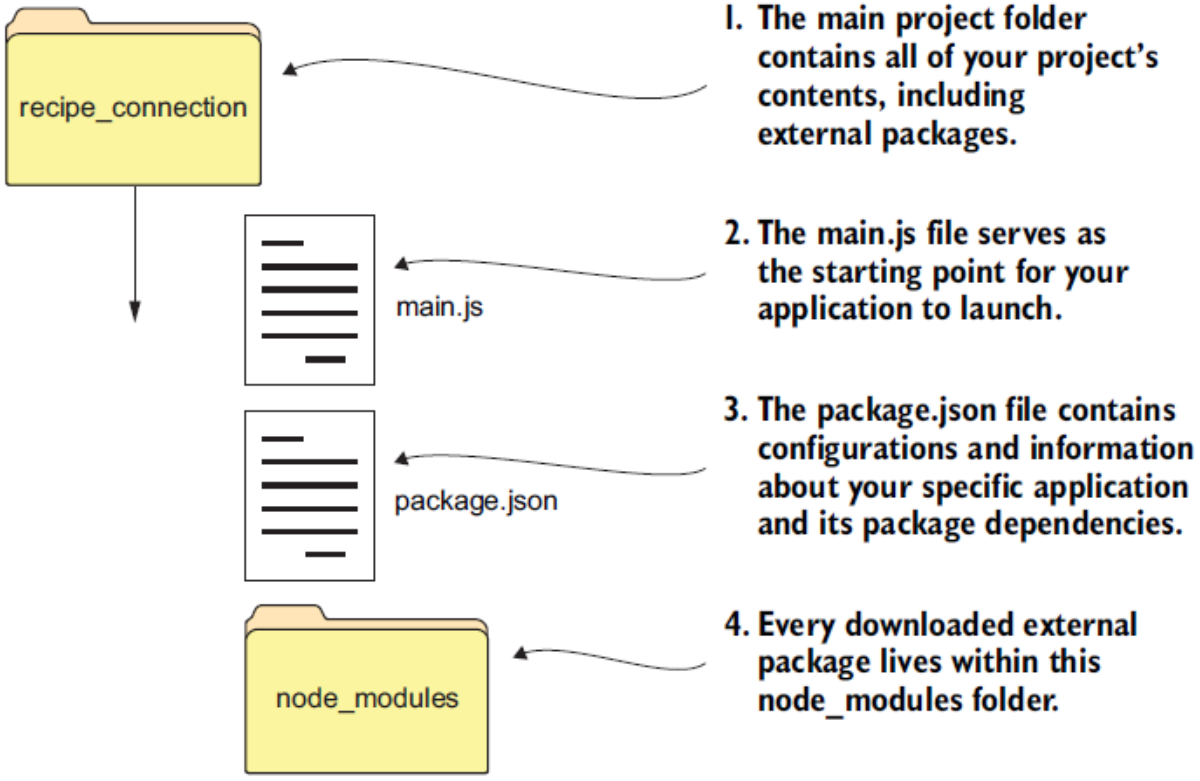


https://www.npmjs.com/package/cities

After you run this command, your package.json gains a new dependencies section with a reference to your cities package installation and its version, as shown in the following listing.

```
{
  "name": "firstApp",
  "version": "1.0.0",
  "description": "",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "RG",
  "license": "ISC",
  "dependencies": {
    "cities": "^2.0.0"
  }
}
```

Node.js application structure with node_modules



Implementing the cities package in main.js

```
const cities = require ("cities");
var myCity = cities.zip_lookup("10016");

console.log(myCity);
```

Sample result from running main.js in terminal

```
{
  zipcode: "10016",
  state_abbr: "NY",
  latitude: "40.746180",
  longitude: "-73.97759",
  city: "New York",
  state: "New York"
}
```

Display the results from the zip_lookup method.

BUILDING A SIMPLE WEB SERVER IN NODE.JS

Web servers and HTTP

A *web server* is software designed to respond to requests over the internet by loading or processing data.

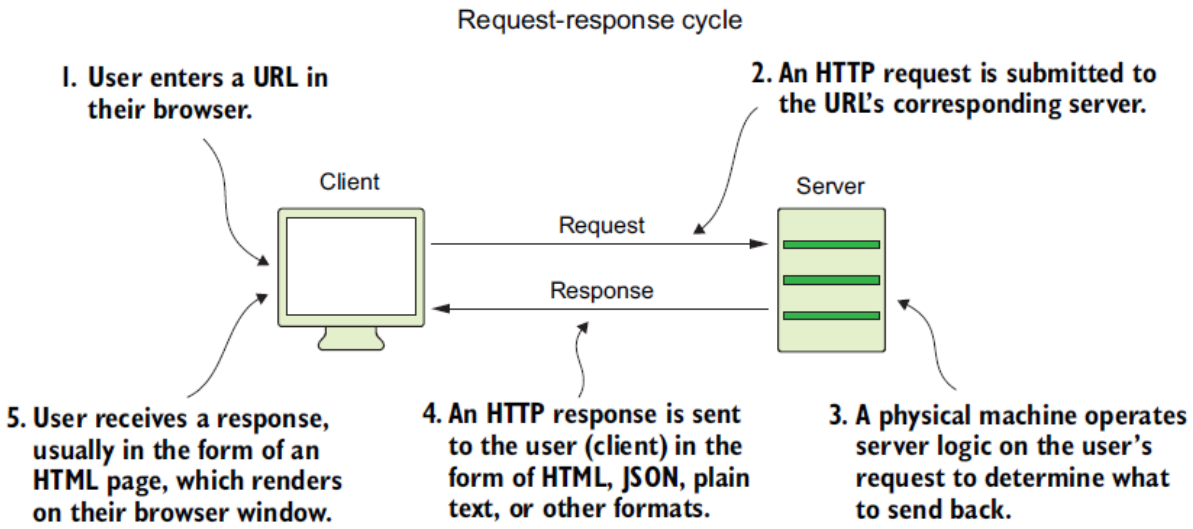
web servers follow *Hypertext Transfer Protocol* (HTTP), a standardized system globally observed for the viewing of web pages and sending of data over the internet.

Here are the two most widely used HTTP methods you'll encounter:

- GET—This method requests information from a server. Typically, a server responds with content that you can view back on your browser (such as by clicking a link to see the home page of a site).
- POST—This method sends information to the server. A server may respond with an HTML page or redirect you to another page in the application after processing your data (such as filling out and submitting a sign-up form).

Most web applications have made changes to adopt *HTTP Secure* (HTTPS), in which transmission of data is encrypted. When your application is live on the internet, you'll want to create a public key certificate signed by a trusted issuer of digital certificates. This key resides on your server and allows for encrypted communication with your client. Organizations such as https://letsencrypt.org offer free certificates that must be renewed every 90 days. For more information about HTTPS, read the article at https:

//developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https.



When you enter the URL you want to see in your browser, an HTTP request is sent to a physical computer elsewhere. This request contains some information indicating whether you want to load a web page or send information to that computer.

You may build a fancy application with many bells and whistles, but at the core lies a web server to handle its communication on the internet.

Initializing the application with npm

Before you get started with a Node.js web application, you need to initialize the project in your project folder in terminal. Open a terminal window, and create a new directory called simple_server with `mkdir`. You can initialize the project with `npm init`.

Coding the application

When you installed Node.js, the core library was installed too. Within that library is a module called `http`. You'll use this module to build your web server. In this section, you also use a package called `http-status-codes` to provide constants for use where HTTP status codes are needed in your application's responses.

In your text editor, create a new file called main.js, and save it in the project folder called simple_server containing the package.json file you created earlier. This file will serve as the core application file, where your application will serve web pages to your users.

Within this project's directory in terminal, run `npm i http-status-codes -S` to save the `http-status-codes` package as an application dependency.

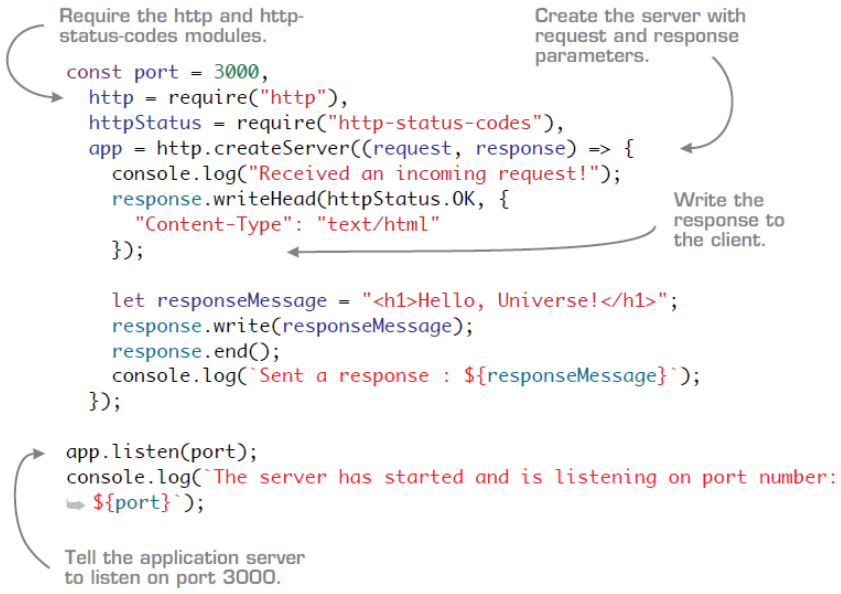
Simple web application code in main.js

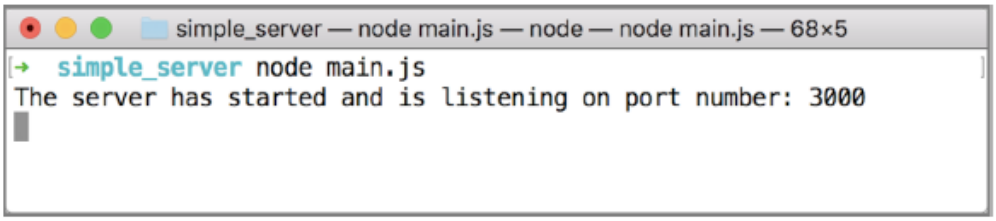
A simple Server:

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

app = http.createServer((req, res) => {
    // console.log(req.url);
    res.write('<html><body><h1>Hello, World!</h1></body></html>');
    res.end();
    // res.end('simple text');
})

app.listen(3000, (err) => {
    console.log('Listening on 3000');
});
```

Note: Checking for communication ok: You need to install `http-status-codes`

```
const port = 3000,
  http = require("http"),
  httpStatus = require("http-status-codes"),
  app = http.createServer((request, response) => {
    console.log("Received an incoming request!");
    response.writeHead(httpStatus.OK, {
      "Content-Type": "text/html",
    });
    let responseMessage = "<h1>Hello, Universe!</h1>";
    response.write(responseMessage);
    response.end();
    console.log(`Sent a response : ${responseMessage}`);
  });
app.listen(port);
console.log(`The server has started and is listening on port number: ${port}`);
```



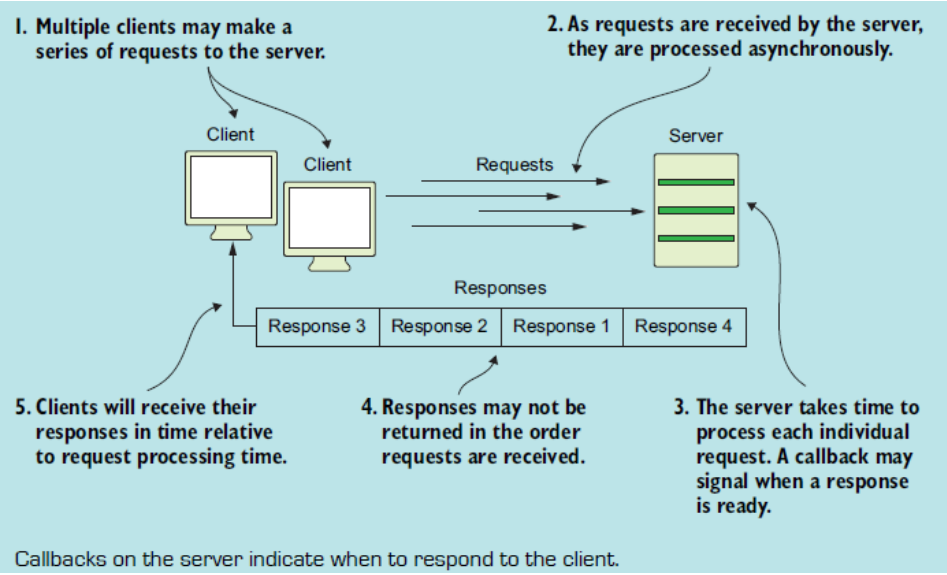


Callbacks in Node.js

Part of what makes Node.js so fast and efficient is its use of callbacks. Callbacks aren't new to JavaScript, but they're overwhelmingly used throughout Node.js and worth mentioning here.

A callback is an anonymous function (a function without a name) that's set up to be invoked as soon as another function completes. The benefit of using callbacks is that you don't have to wait for the original function to complete processing before running other code.

In the http web server example, incoming requests from the client are received on a rolling basis and thereupon pass the request and response as JavaScript objects to a callback function, as shown in the following figure:



Callbacks on the server indicate when to respond to the client.

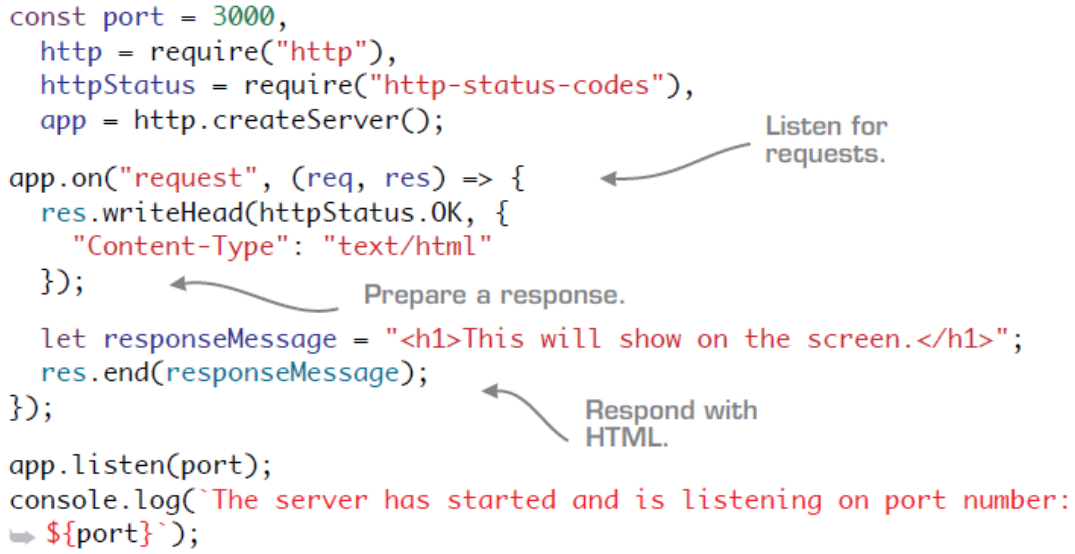
HANDLING INCOMING DATA

Create a new project called ***second_server*** within its own project directory, and inside, add a new ***main.js*** file.

In your code, you have a server object that has a callback function, **(req, res) ⇒ {}**, which is run every time a request is made to the server. With your server running, if you visit **localhost:3000** in your browser and refresh the page, that callback function is run twice—once on every refresh. In other words, upon receiving a request, the server passes a request and response object to a function where you can run your code.

A simple server with a request event listener in main.js

```
const port = 3000,
  http = require("http"),
  httpStatus = require("http-status-codes"),
  app = http.createServer();
  app.on("request", (req, res) => {
  res.writeHead(httpStatus.OK, {
    "Content-Type": "text/html",
  });
  let responseMessage = "<h1>This will show on the screen.</h1>";
  res.end(responseMessage);
});

app.listen(port);
console.log(`The server has started and is listening on port number: ${port}`);
```

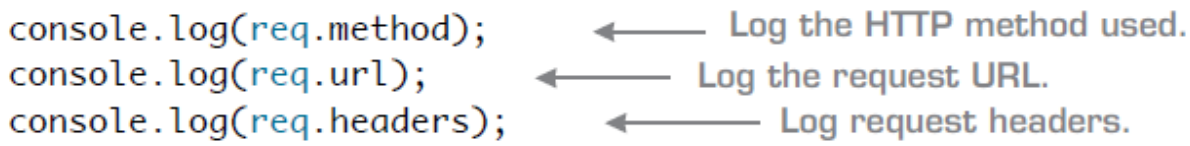


Run node main in terminal and visit http:// localhost:3000/ in your web browser to view the response containing one line of HTML on the screen.

you want to modify the content based on the type of request you get. If the user is visiting the contact page or submitting a form they filled out, for example, they'll want to see different content on the screen. The first step is determining which HTTP method and URL were in the headers of the request.

Routing is a way for your application to determine how to respond to a requesting client. Some routes are designed by matching the URL in the request object. That method is how you're going to build your routes in this lesson.

Each request object has a url property. You can view which URL the client requested with req.url. Test this property and two other properties by logging them to your console. Add the code in the next listing to the app.on("request") code block.



Because some objects in the request can have within them other nested objects, convert the objects to more-readable strings by using JSON.stringify within your own custom wrapper function, getJSONString.

```
const getJSONString = obj => {
    return JSON.stringify(obj, null, 2);
    };
```

```
console.log(`Headers: ${getJSONString(req.headers)}`);
```

```
const port = 3000,
  http = require("http"),
  httpStatus = require("http-status-codes"),
  app = http.createServer();
app.on("request", (req, res) => {
  res.writeHead(httpStatus.OK, {
    "Content-Type": "text/html",
  });
  let responseMessage = "<h1>This will show on the screen.</h1>";
  // console.log(req.method);
  // console.log(req.url);
  // console.log(req.headers);
  // console.log(`Headers: ${getJSONString(req.headers)}`);
  res.end(responseMessage);
});

const getJSONString = obj => {
    return JSON.stringify(obj, null, 2);
    };

app.listen(port);
console.log(`The server has started and is listening on port number: ${port}`);
```

When you restart your server, run main.js again, and access http://localhost:3000 in your web browser, you'll notice in your terminal window information indicating that a GET request was made to the / URL (the home page), followed by that request's header data.

Try entering a different URL, such as http://localhost:3000/testing or http://localhost: 3000/contact. Notice that you still get the same HTML text on the browser, but your console continues to log the URLs you type in the browser.

Adding routes to a web application

A *route* is a way of determining how an application should respond to a request made to a specific URL. An application should route a request to the home page differently from a request.

After you established a request to your web server; from there, you can evaluate the type of request and prompt an appropriate response. This example accepts any request made to the server (localhost) at port 3000 and responds with a line of HTML on the screen.

The next step is checking the client's request and basing the response body on that request's contents. This structure is otherwise known as *application routing*.

Simple routing in a web server in main.js

```
const routeResponseMap = {
    "/info": "<h1>Info Page</h1>",
    "/contact": "<h1>Contact Us</h1>",
    "/about": "<h1>Learn More About Us.</h1>",
    "/hello": "<h1>Say hello by emailing us here</h1>",
    "/error": "<h1>Sorry the page you are looking for is not here.</h1>"
    };
    const port = 3000,
    http = require("http"),
    httpStatus = require("http-status-codes"),
    app = http.createServer((req, res) => {
    res.writeHead(200, {
    "Content-Type": "text/html"
    });
    if (routeResponseMap[req.url]) {
    res.end(routeResponseMap[req.url]);
    } else {res.end("<h1>Welcome!</h1>");
}
});

app.listen(port);
console.log(`The server has started and is listening on port number: ${port}`);
```

```
const routeResponseMap = {
  "/info": "<h1>Info Page</h1>",
  "/contact": "<h1>Contact Us</h1>",
  "/about": "<h1>Learn More About Us.</h1>",
  "/hello": "<h1>Say hello by emailing us here</h1>",
  "/error": "<h1>Sorry the page you are looking for is not here.</h1>"
};

const port = 3000,
  http = require("http"),
  httpStatus = require("http-status-codes"),
  app = http.createServer((req, res) => {
    res.writeHead(200, {
      "Content-Type": "text/html"
    });

    if (routeResponseMap[req.url]) {
      res.end(routeResponseMap[req.url]);
    } else {

      res.end("<h1>Welcome!</h1>");
    }
  });

app.listen(port);
console.log(`The server has started and is listening on port number:
➥ ${port}`);
```

Define mapping of routes with responses.

Check whether a request route is defined in the map.

Respond with default HTML.

SETTING UP AN APP WITH EXPRESS.JS

Express.js, can reduce the time it takes you to get your application running.

Installing the Express.js package

Express.js provides methods and modules to assist with handling requests, serving static and dynamic content, connecting databases, and keeping track of user activity.

You need to download and install it by running the following command within your project directory in terminal: **npm install express.**

In other words, your application depends on Express.js to work, so you need to ensure that it's installed. Open package.json to see this Express.js package installation under the dependencies listing.

We're going to learn how to create a real server locally on our computer using Node.js and Express.

Complete this part by yourself. Create a new directory, and call it my-express-server, and then I want you to cd into that directory. Then inside that project folder create a new file called server.js.

Initialize npm. Using **npm init**.

Set the starting point as server.js.

Install Express … '**npm install express**'.

So now that we've installed Express, the next step is of course to require Express.

When we've required and incorporated Express into our file, the next step is to create a new constant called app, and this is simply a function that represents the Express module, and we bind that to app.

So we now have a constant called app, which is set equal express, and you'll notice that when you come across web sites built using Express, the word app is usually always used when you're referring to the Express module.

So now that we've created our constant app, the next step is to use this app, and we're going to use one of its methods called listen. And this tells it to listen on a specific port for any HTTP requests that get sent to our server.

As usual choose the port 3000, and if we hit save, now we have literally just built your first server.

```
const port = 3000,
  express = require("express");
app = express();
app.listen(port, () => {
  console.log(
    `The Express.js server has started and is listening on port number: ${port}`
  );
});
```

This is the bare bones of an Express server. Now let's run this server by saying node server.js. Now a port is basically just a channel that we've tuned our server to. Our server is just tuned into the channel 3000.

So now, if we hit save and we try to run our code again, node server, you can see now, instead of just hanging, it's telling us that server has started on port 3000.

Now if we head over to that ports location, which is localhost:3000, because we're hosting our server locally, and after the colon we specify the port that our server is set up on, which is 3000, and you can see that we get this error, where it says, “Cannot GET /“.

```
Cannot GET /
```

Well, it means that when our browser is trying to get in touch with our server on the port 3000, it's not able to get anything back. Now we have to figure out how can we write some code so that our server responds when a browser is making a request to our server.

We have to send the browser some information to display. We need to request and response that we can provide when a browser makes a get request. Just above the app.listen, we're going to say app.get. This is a method that's provided by Express that allows us to specify what should happen when a browser gets in touch with our server and makes a get request.

Now the first parameter it takes is the location of the get request. So when we type localhost:3000, the get request is being sent to the route of our web site, which is represented by a forward slash. So this is basically our home page.

Now when that get request happens, we can trigger a callback function, and this callback function can have two parameters: request and response. The method, app.get, defines what should happen when someone makes a get request to the home route.

So that's the first parameter. And then there's a callback function that tells the server what to do when that request happens. So let's printout this request object that we get when the callback gets triggered and see what it looks like.

```
const port = 3000,
  express = require("express");
app = express();
app.get("/", (req, res) => {
  res.send("Hello, Universe!");
});
app.listen(port, () => {
  console.log(
    `The Express.js server has started and is listening on port numbe
r: ${port}`
  );
});
```

Now the second object here is the response. This is the response that outcome server can make when the request gets triggered at this home location. We can tap into the response object, and we can use the send method to send a response.

We simply send back “Hello, Universe”. You can now see that we see the word ‘Hello, Universe’ in our browser. When we hit enter, the browser will go to that location and make a request to get some data back. And when that request gets made at that home location, then this callback gets triggered, and we send the browser a response, which is just the plain text of ‘Hello’.

Now that gets sent back to our browser and it renders it on screen. So you don't have to just send plain text. You can actually send HTML, as you can see in other routes.

ROUTING IN EXPRESS.JS

```
const port = 3000,
express = require("express"),
app = express();
app.get("/", (req, res) => {
    res.send(`Hello, Universe`);
    });
app.get("/items", (req, res) => {
res.send(`This is the page for vegetables`);
});
app.get("/menu", (req, res) => {
    let veg = req.params.vegetable;
    res.send(`This is the page for menu`);
    });
app.listen(port, () => {
console.log(`Server running on port: ${port}`);
});
```

Lab: Node -- Making a Calculator

Let’s set up a new website with a server using Node and Express to create a really simple web site that acts as a basic structure of simple calculator. Now, in the process of building this, we're going to get to create our very first web application, and it's not just a web site anymore.

So when our web site makes a request to our server, it's going to execute the code and only deliver the outcome back to the user, so the user doesn't get to see any of the code and logic of our calculator. It's all done on our server.

And once we grasp this concept, then we'll be able to make much faster, more complex web sites that can do computation before it even renders the web site to be delivered to the user, and we'll be able to

interact with databases, and query, and search, and manipulate our databases, and create much more interesting web applications.

Before we start creating our Calculator website, we'll need to set up a new project. Follow the steps below using your Hyper Terminal to complete this challenge:

- Make a new folder called Calculator on your Desktop
- Change Directory to this new folder
- Inside the Calculator folder, create a new file called calculator.js
- Set up a new NPM package
- NPM install the express module
- Require express in your calculator.js
- Setup express
- Create a root route get method with app.get()
- Send the words Hello World from the root route as the response
- Spin up our server on port 3000 with app.listen
- Run server

Cd to Calculator folder. Create a new const called express. This is going to be set to require express. So now that we've got our const express, then we're going to set up a new app that is going to be using the express module. And finally Create a home route, so that's going to be app.get and make sure your server is working as steps below:

Have a callback function with a request and a response, and we're simply going to respond by sending, “Hello world!” So now that we've defined our route, then we're going to spin up our server, and we do that with app.listen, and we're going to listen again on port 3000, and then we're going to have a callback that simply logs that the server is running on port 3000.

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

const app = express();
app.use(bodyParser.urlencoded({extended: true}));

app.get("/", function (req, res) {
  res.send(“Hello World”);
});

app.listen(3000, function () {
  console.log("server started on port 3000");
});
```

So now that we've set up our home route, we've got our app to listen on port 3000.

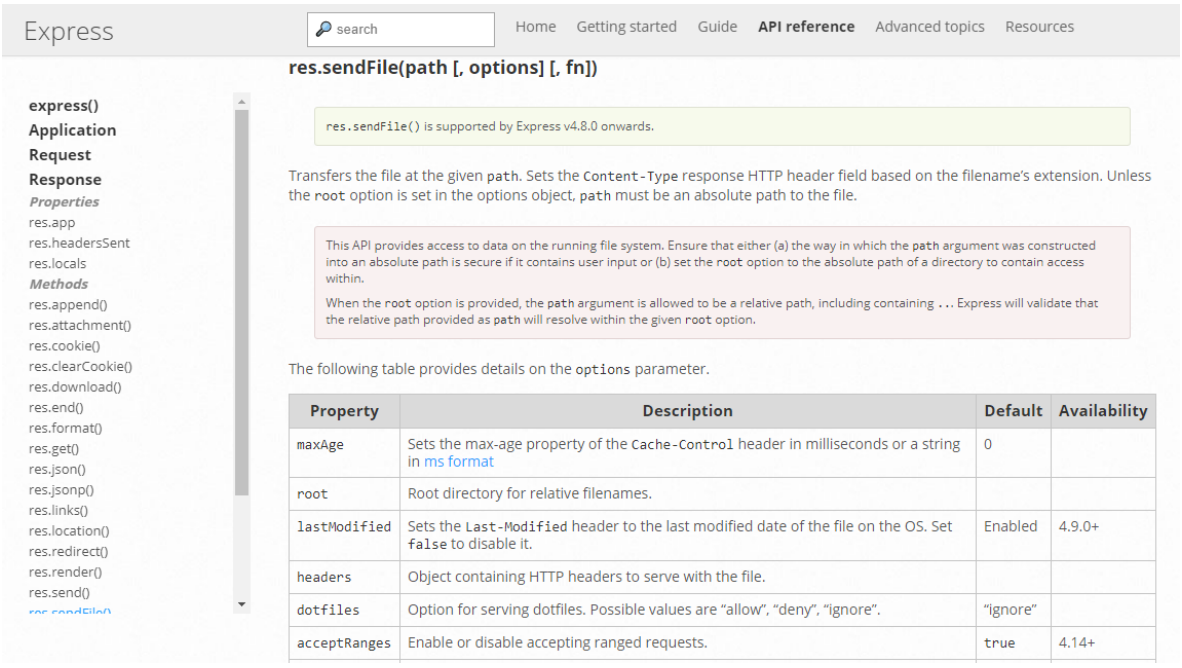
Now in the HTL module, we explored how to create and use HTML forms. Now in this lesson, armed with our knowledge of Javascript, Node, Express, we're going to put it to use in our web site, and we’re going to use the data that gets entered into the forms, and perform calculations on it in our server.

Inside the Calculator directory, Create a new file called index.html. Inside the body we're going to include a form. Now this form is not going to have a class, but it will have an action and a method. Keep the method as post and the action as index.html.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Calculator</title>
  </head>
  <body>
    <form action="/" method="POST">
      <input type="text" name="num1" placeholder="First Number" />
      <input type="text" name="num2" placeholder="Second Number" />
      <button type="submit" name="submit">Calculate</button>
    </form>
  </body>
</html>
```

When we use res.send, we're sending individual bits of HTML data. But if we want to send an entire web page, such as our index.html, we have to use something different. So if we head over to the Express API reference, and you can see it's organized by which part you're looking for. And we're looking for the response part, and you can see it has a whole bunch of different methods, for example res.send, which is what we've been using so far.

http://expressjs.com/en/4x/api.html#res.send



But there's also, if you scroll down, res.sendFile, and this transfers the file over to the browser when they make a get request. So, instead of saying res.send, we can say res.sendFile, and inside the parentheses we're going to give a single input, which is the location of the file that we want to send.

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

const app = express();
app.use(bodyParser.urlencoded({extended: true}));

app.get("/", function (req, res) {
  res.sendFile(__dirname + "/index.html")
});

app.listen(3000, function () {
  console.log("server started on port 3000");
});
```

Calculate button:

Currently server running on port 3000, running **calculator.js** file.

Open up Chrome Developer Tools, then head over to the Network tab, and make sure that down here you've got the All tab selected as well, we're going to test our form out.

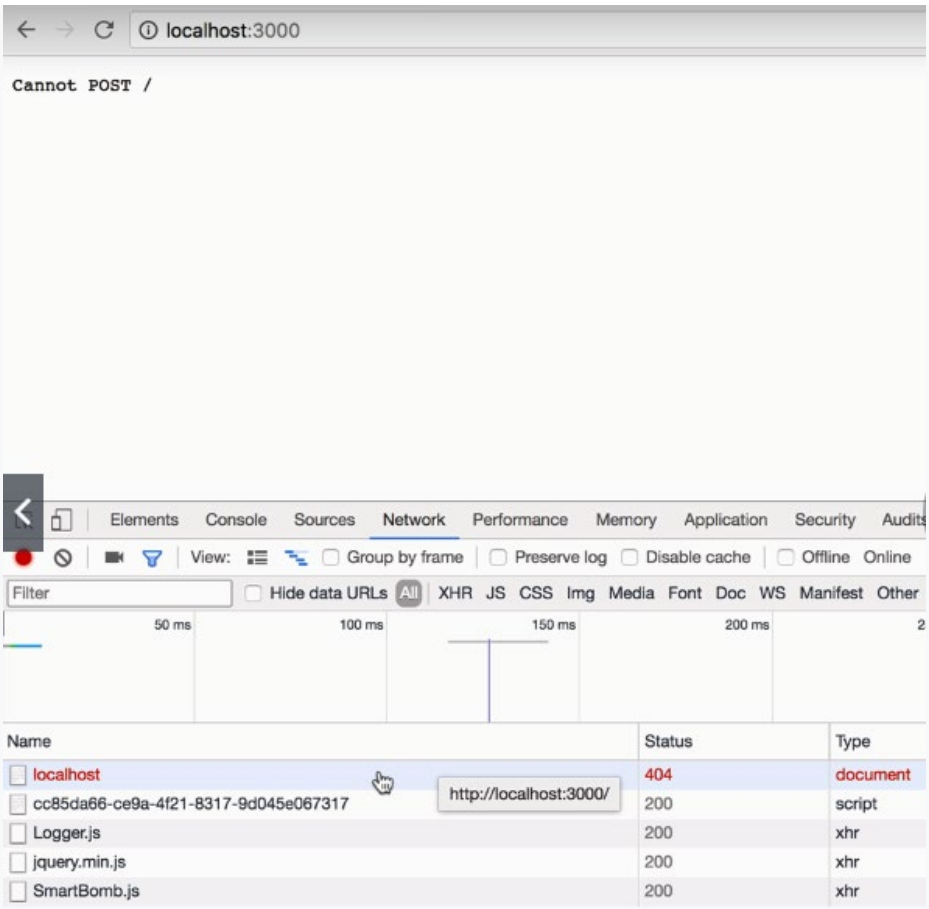
Put in a first number and a second number, and then press Calculate. Now a whole bunch of things happen, and all of these networking requests get logged down here. But we get this code 404, and everything's in red, which seems kind of like it's bad, and then we get this error up here saying, “Cannot POST to /index.html”.

Now where does that come from? It comes from action and a method inside the form tag. Now the method is the post method, so we're sending this data somewhere, and that somewhere is what's defined by the action attribute.

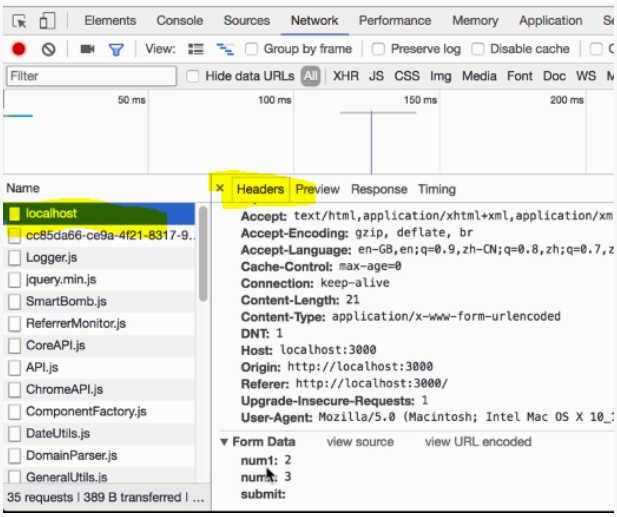
So we’re sending it to something called index.html, which is not what we want. We want to send it to our server, which is at the home route location, so it's just the forward slash. Now, if you don't have an action attribute, that's fine as well. By default, if it doesn't exist, then the form will simply send the data to the current page where it's on, so that will be the equivalent of this.

So our form has a post method, which means it's going to try and send the data that is entered into the inputs to a location that is our home route. So now, if we hit save, and we go back to our localhost:3000, our home page, and let's try again pressing that Calculate button.

Now this time, we're still getting a 404 and a “Cannot POST”, but if you click on it, and you go over to the Headers tab, and you scroll down, you can see that we've got a bunch of information, including some form data, and the data that we're getting access to is the parameter num1, which remember is bound to our first input using the name attribute, and then that has a value of 2, which is what we entered into the form previously, and the input with a name of num2 has a value of 3.



But we have a problem, because the status code is ‘404 Not Found’.



Let's add an app.post method to handle any post requests that come to our home route, and then we're going to have a callback with, again, req and res, request and response and send back, "Thanks for posting that!"

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

const app = express();

app.get("/", function (req, res) {
    res.sendFile(__dirname + "/index.html")
});

app.post("/", function (req, res) {
    res.send("thanks for posting");
});

app.listen(3000, function () {
  console.log("server started on port 3000");
});
```

Go back to our localhost:3000, I'm going to put in two numbers and press Calculate. You can see now, when we look at our localhost, we're getting a message back, and also we're getting the status code 200, which is 'OK'

Now we need to access pieces of form data? Because that's essentially what we need, right? We need to be able to get that data into here, into this callback function, so we can calculate the output, and then send the result back to the browser.

Now, in order to tap into those pieces of data, we have to install another NPM package, which is called Body Parser. Exit the server, and then in the console typw "npm install body-

parser". It's going to allow us to pass the information that we get sent from the post request.

Now we can require it in the calculator.js and parse data that comes from an HTML form. So whenever you're trying to grab the information that gets posted to your server from an HTML form.

Create a new const that's called bodyParser, and it's going to be requiring body-parser package that we just installed. And Body Parser works with Express, so we can say app.use, and we're going to specify the thing we wanted to use, which is bodyParser. Now Body Parser has a few modes, if you will. There is, for example, bodyParser.text, so parse all the requests into text, or bodyParser.json, which is that special format that we saw before, which kind of looks a bit like Javascript objects, or the one that we're going to be using is bodyParser.urlencoded. This is the one that we use when we're trying to parse data that comes from an HTML form.

And in addition to that, add an option called 'extended', and we're going to set it to be 'true'. And by setting that extended option to true, that basically just allows us to post nested objects. And it's not something that we're going to be using, but it's something that bodyParser is requiring you to explicitly declare. This is basically the code that you need to write every single time you want to use Body Parser.

Now why would you want to use Body Parser? It allows you to go into your routes, and you can tap into something called request.body, that is the parsed version of the HTTP request.

Let's log this and see what we get when we try to make a post request.

```
const express = require("express");
const bodyParser = require("body-parser");

const app = express();
app.use(bodyParser.urlencoded({extended: true}));

app.get("/", function (req, res) {
    res.sendFile(__dirname + "/index.html")
});

app.post("/", function (req, res) {

    console.log(req.body)
    res.send("Thanks for posting);

});

app.listen(3000, function () {
  console.log("server started on port 3000");

});
```

So restart server and reload web site, and let's put in two numbers, and hit Calculate. So we get sent back, “Thanks for posting that!” from the res.send, but we also execute the console.log, where we log the request.body, and that logs the form data.

By using Body Parser, we're able to parse the HTTP request that we get, and by using urlencoded we can get access to the form data, and we can then tap into each of these as if they were just properties of the object body.

We can, for example, log request.body.num1. And remember that naming comes from the name attribute of your input. We're only logging the value of the first input. If we go back to our web site and put in a number in here, say 5 and 6, then when we press Calculate, we get 5 logged in here, so that value gets stored inside this request.body.num1.

Now all we need to do is create a variable that's going to hold our num1, and that's going to be equal to request.body.num1. Then we're going to create another one called num2, and this is going to be equal to request.body.num2.

Then we can calculate the result, which is going to be num1 + num2, which is making a really simple calculator that adds two numbers. Then we're going to send back, instead of “Thanks for posting that!”, we'll say, “The result of the calculation is “, and then we're going to append that variable result onto the end.

So now save, update, go over to home page, and let's try and add 4 and 5 together, press Calculate, the result of the calculation is 45. num1 and num2 that we're getting back from bodyParser, it gets parsed as text, so if we want this to be a number, then we need to explicitly turn this into a number.

```
const express = require("express");
const bodyParser = require("body-parser");

const app = express();
app.use(bodyParser.urlencoded({extended: true}));

app.get("/", function (req, res) {
    res.sendFile(__dirname + "/index.html")
});

app.post("/", function (req, res) {

  var num1 = Number(req.body.num1);
  var num2 = Number(req.body.num2);
  var result = num1 + num2;

  res.send("the result of this calculator is " + result);
});

app.listen(3000, function () {console.log("server started on port 3000");
});
```

We do that by simply writing Number, with a capital N, and inside the parentheses we put in the piece of text that we want to turn into a number. When we're calculating results, instead of appending num1 to num2, we can add num1 to num2.

Let's try this again. The first number is 4, second number is 5. And now we get 9.

So if you're wondering how we got these words num1 and num2, then it's as simple as going into your index.html and changing the name here.

The important thing to take away from this is when you look at our web site and I right click and say View Page Source, you can see that all the client gets to see, all my browser gets to see, when I try to go to this web site is just a plain and simple HTML web site.

We now have a web application because our code is running on the backend as opposed to just simply having static files being rendered and loaded up, and having our Javascript run on the client side, or the front end.

Node.JS: Getting Response Body

The following code get response from a GET request

```
const https = require('https');

function getQuote() {
  // simple get request
  const request = https.get(
    'https://animechan.vercel.app/api/random',
    // response will be used in the following callback
    (response) => {
      console.log(response.statusCode);
    }
  );
}

getQuote();
```

The following code receives the JSON data, but as a binary stream

```
const https = require('https');

function getQuote() {
  // simple get request
  const request = https.get(
    'https://animechan.vercel.app/api/random',
    // response will be used in the following callback
    (response) => {
      console.log(response.statusCode);

      // when data receive "data" event is emitted
      // that has an "on" method, with parameters "event" and "callback"
      response.on("data", (data) => {
        console.log(data);
      });
    }
  );
}

getQuote();
```

So in order to get all streams we should concatenate all data, and also convert to String. Please pay attention that we need to start concatenation when data is received and finally when it is finished.

```
const https = require('https');

function getQuote() {
  // simple get request
  const request = https.get(
    'https://animechan.vercel.app/api/quotes',
    // response will be used in the following callback
    (response) => {
      // concatenate all streams into one object
      let body = '';
      // console.log(response.statusCode);
      // when data receive "data" event is emitted
      // that has an "on" method, with parameters "event" and "callback"
      response.on('data', (data) => {
        // toString converts the data stream to String
        body += data.toString();

        //along with data event there is "end" even
        // that emits when all data is reveive

        response.on('end', () => {
          console.log(body)
        } )
        // console.log(data);
      });
    }
  );
}

getQuote();
```

We have the response body as a String. We need to convert it into an Object we can access the data more programmatically. We get JSON, but we need to parse it so it shows nicely. Converting a string to a data structure is called parsing.

```
const https = require('https');

function getQuote() {
  // simple get request
  const request = https.get(
    'https://animechan.vercel.app/api/quotes',
    // response will be used in the following callback
    (response) => {
      // concatenate all streams into one object
      let body = '';
```

```
      // console.log(response.statusCode);
      // when data receive "data" event is emitted
      // that has an "on" method, with parameters "event" and "callback"
      response.on('data', (data) => {
        // toString converts the data stream to String
        body += data.toString();

        //along with data event there is "end" even
        // that emits when all data is reveive

        response.on('end', () => {
          //using JSON.parse to convert JSON to a data structure
          console.log(JSON.parse(body))
        } )
        // console.log(data);
      });
    }
  );
}

getQuote();
```

Debugging Node:

There is a brilliant system for debugging in Node. If you want to debug properly, or it's a complex problem and console logging and process exiting isn't really getting to it, there is a much better way of doing it, and it's built right into Node.

As you know, we start Node by giving it a file to run.

But if you instead run by:

Node –inspect-brk filename.js

Then open an empty chrome window and in the **URL** type: **about:inspect** and then look for "**Open dedicated Dev Tools for Node**" and click. => There's our code. We can now use a debugger to step through our code.