Table of Contents

Introduction

Hello and thank you for joining us for this intro to Web development with JavaScript workshop.

My name is David VanDusen, Web bootcamp instructor at Lighthouse Labs, and your instructor today.

My background is in software development, focusing on Web technologies. I started programming at a young age and built websites during some of the most exciting times in the technologies' histories.

That said, I started out in fine arts and music and have played everything from piano in a jazz big band to singing in a thrash metal band. Programmers are all sorts of people.

Pair programming

You're encouraged to buddy up with one other person. That said, both of you should ideally do the setup work individually on your own laptops.

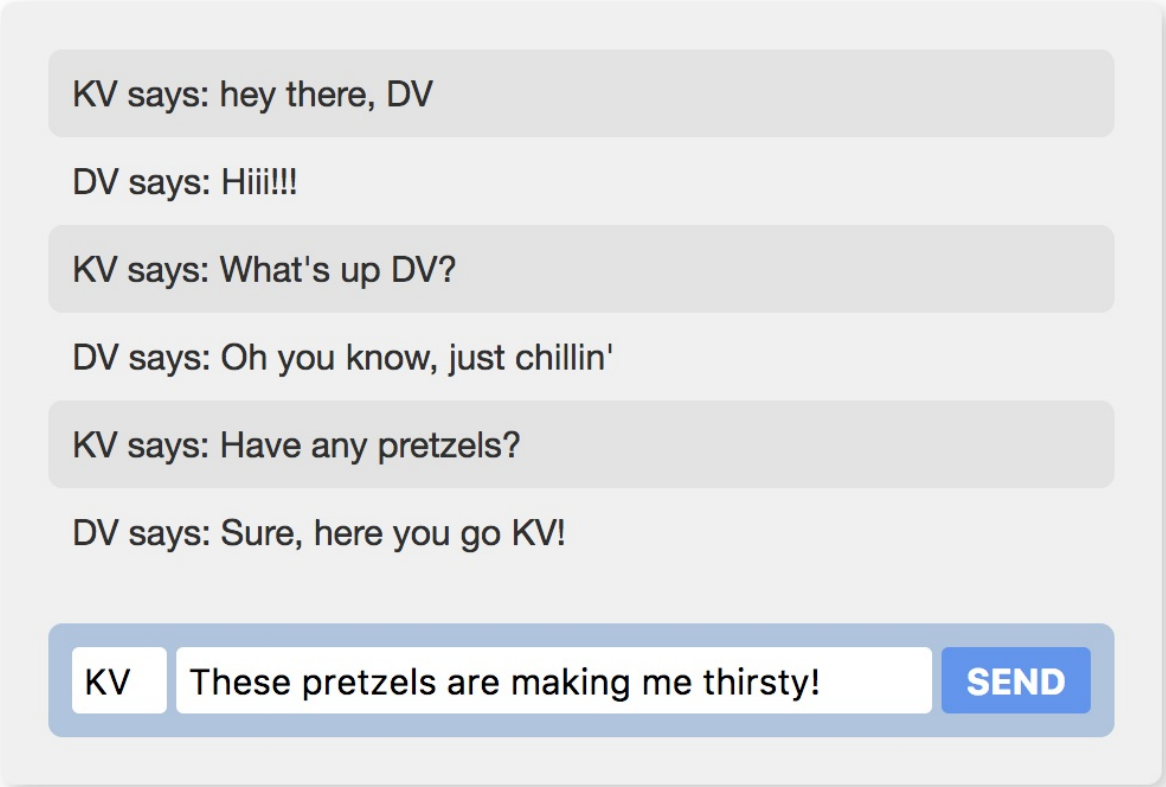
The goal

Today's goal is to get acquainted with JavaScript-based Web programming by creating a simple chat application. We'll use modern tools (Node.js, Web Sockets, etc.) and touch upon many different technologies.

The objective is not to fully understand every line of code, but to understand server-client architecture and to be inspired by how much can be created with so little code.

The product

The finished app that we'll be building in just a few hours today will look and feel much like this.

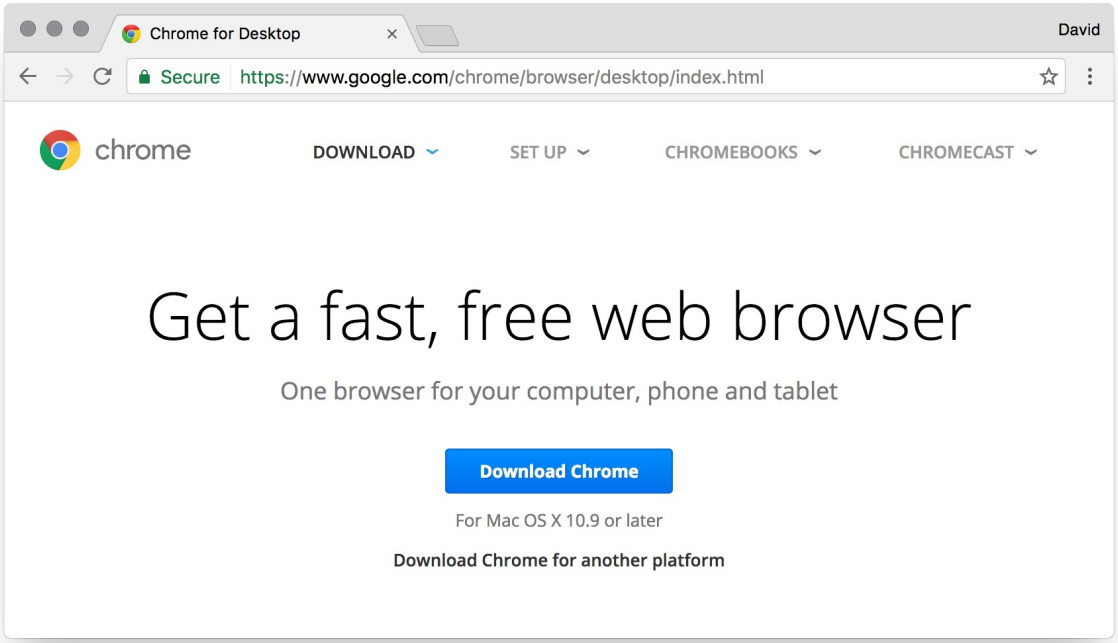

KV says: hey there, DV
DV says: Hiii!!!
KV says: What's up DV?
DV says: Oh you know, just chillin'
KV says: Have any pretzels?
DV says: Sure, here you go KV!
KV
These pretzels are making me thirsty!
SEND

Setup

This section lists how to get the software you need to build your app.

Google Chrome

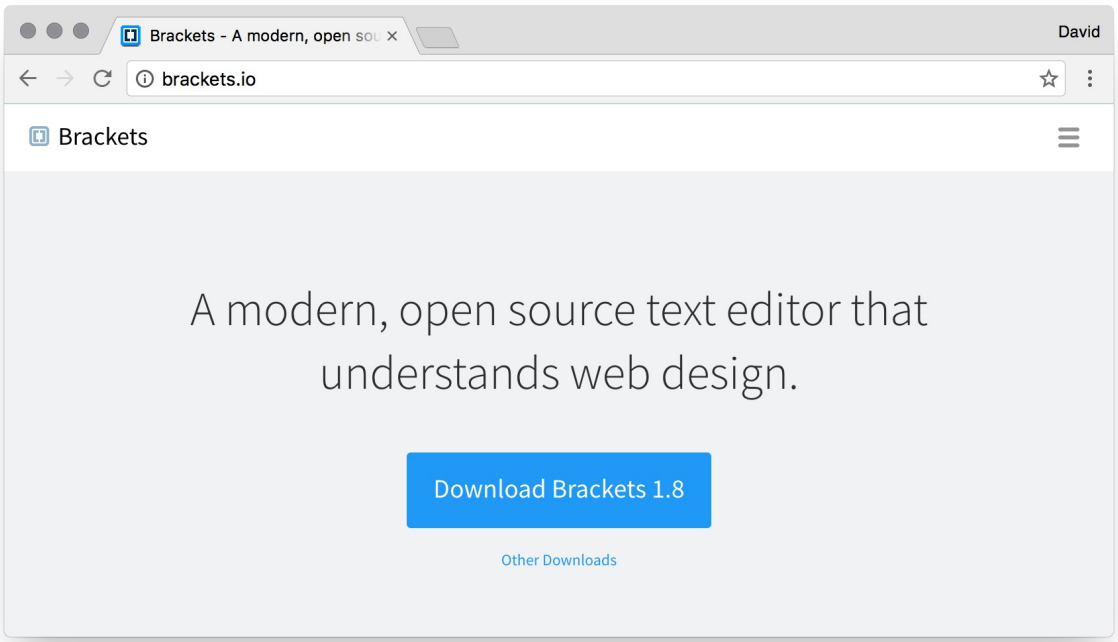
Different Web browsers provide different tools for doing Web development. The examples in this workshop will be using Google Chrome. You can download it from Google.



Brackets

Brackets is a free code editor from Adobe. Basically code editors are word processors that only work on plain text files, but they can provide other cool features if they know what programming languages you're using.

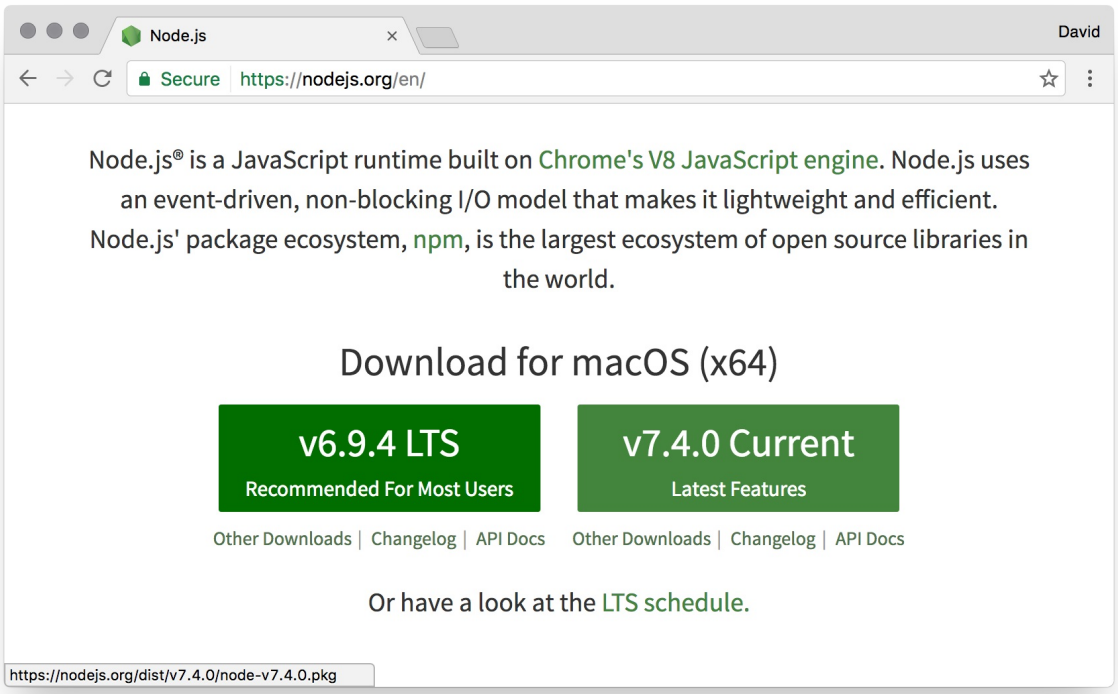
Download Brackets from brackets.io.



Node.js

Node.js is a program for running JavaScript code on the server-side. Because we'll be coding a Web server today, we might as well write it in JavaScript because that's the language we code the client-side browser code in.

Get it from nodejs.org.



Part I

HTML and CSS

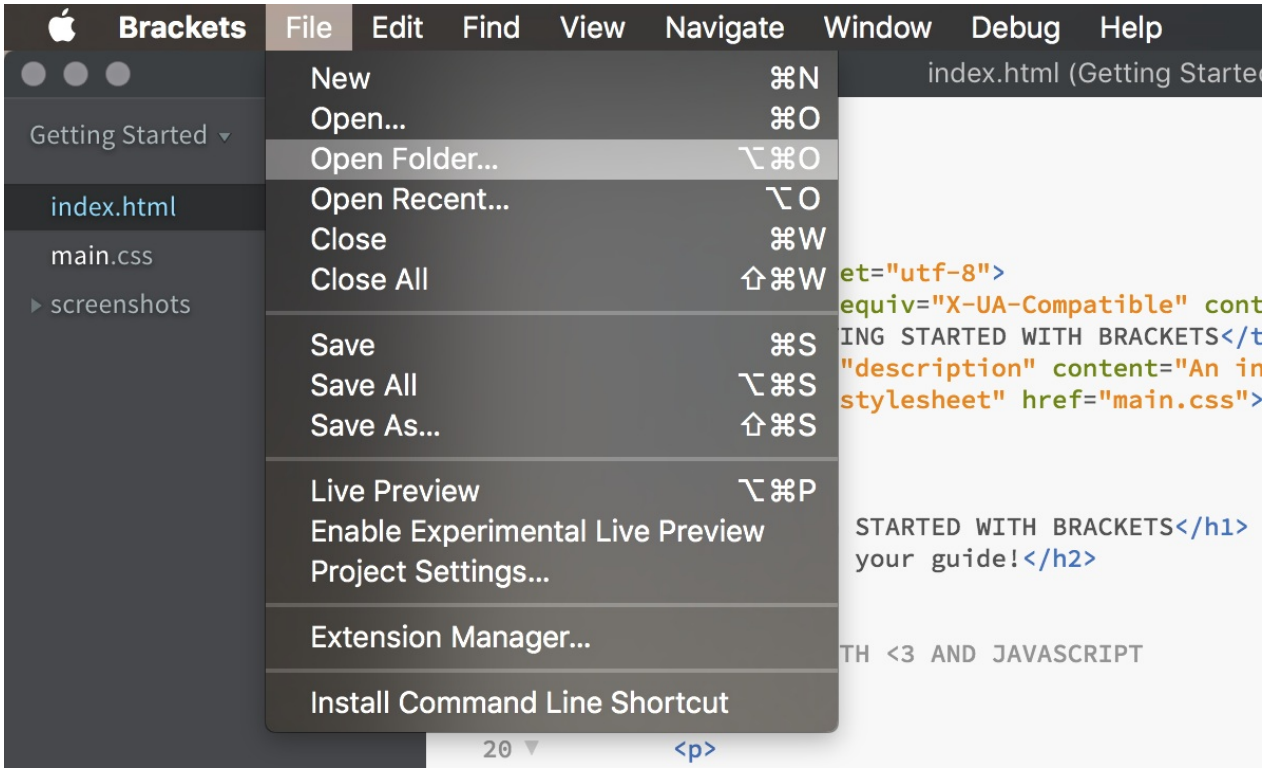
HTML

HTML is the most fundamental language used on the Web. With it, we add information into documents that explain how they are linked together — that's how the Web works!

Now let's use it to build our app.

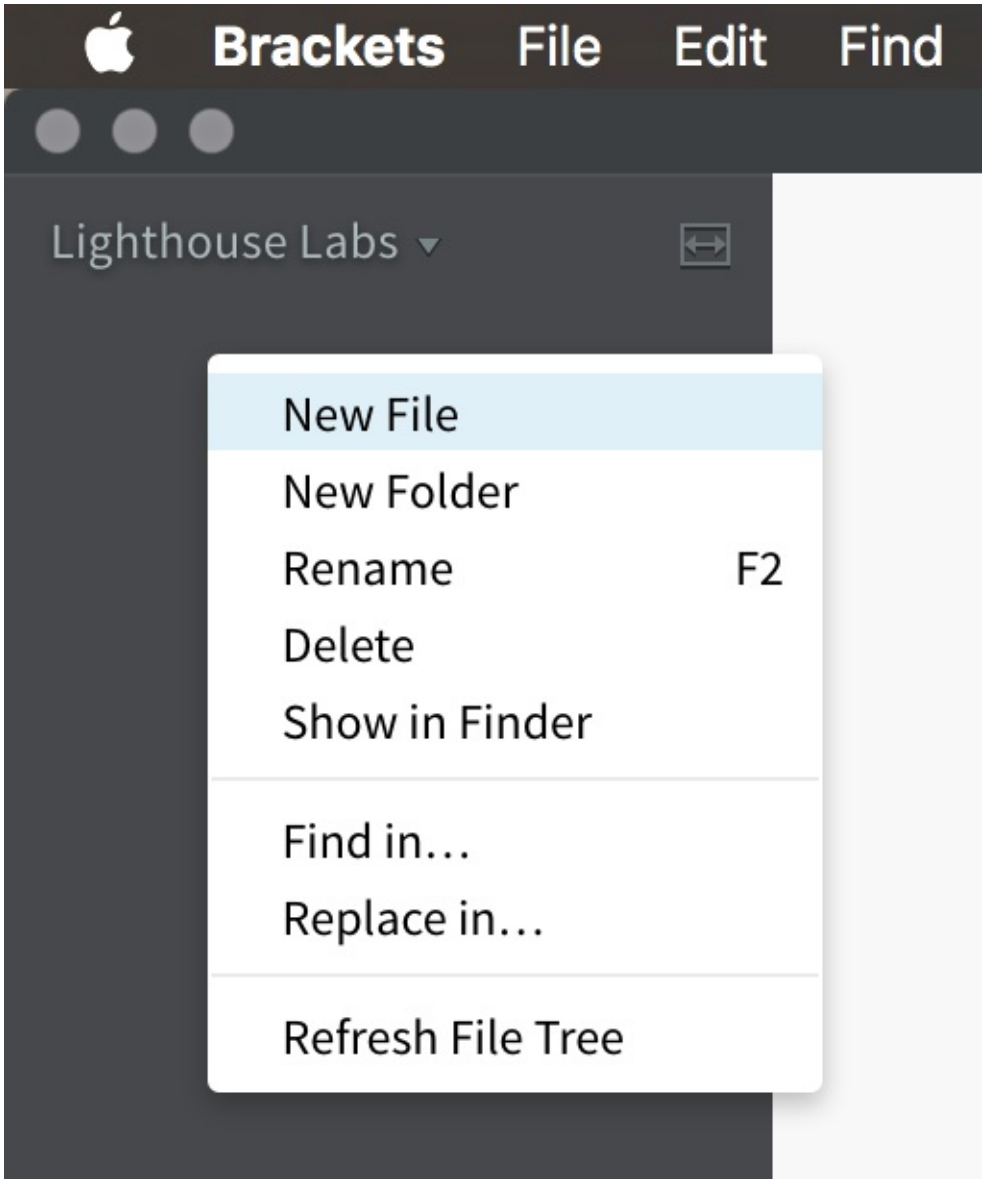
Step 1

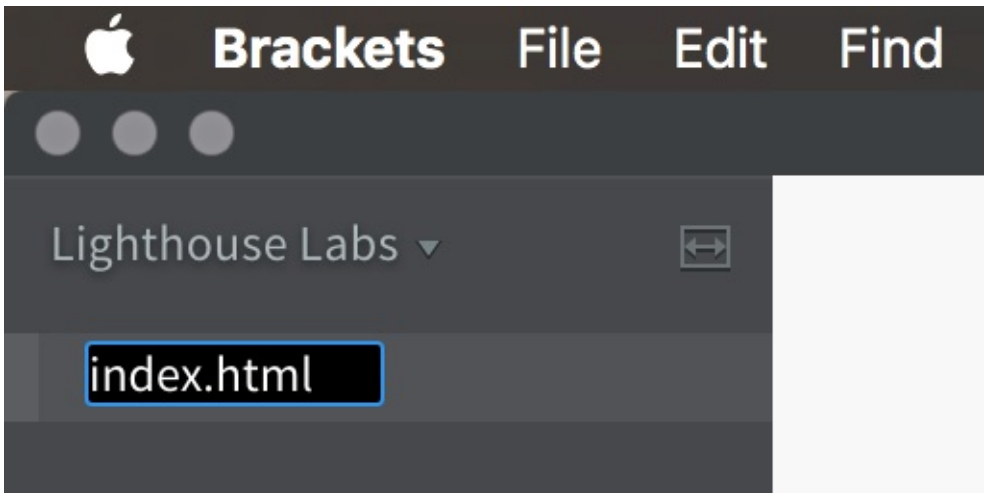
Create a folder on your computer to put the files for this project. A good name for the folder might be **Lighthouse Labs**. Then go to **File > Open Folder...** and select your new folder.



You should now see an empty workspace ready for us to create our project in.

Create an new file in your project called **index.html** by right clicking the grey area on the left hand side.



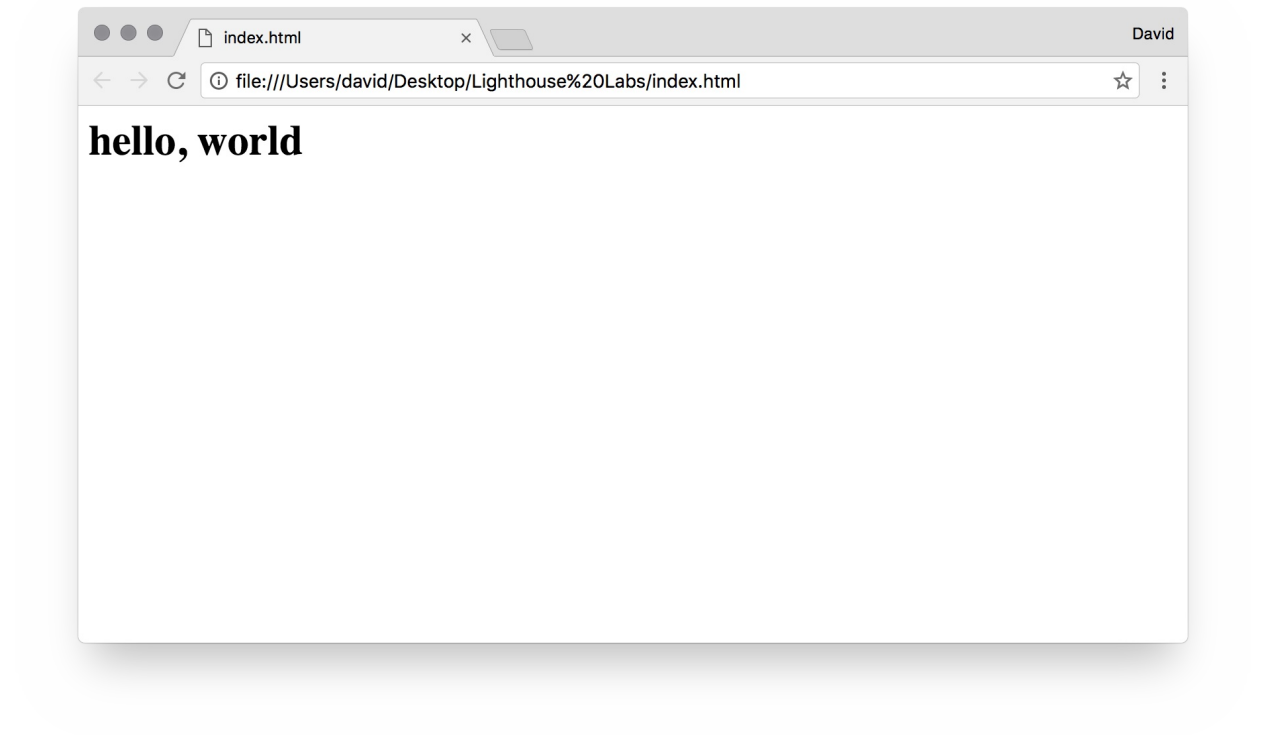


Open the file and add the following html code into it.

```
<h1>hello, world</h1>
```

It should look like this.

Save the file and then navigate to your project folder and choose to open the **index.html** file in your Web browser. You'll get something like this.



And that's it — you just built a Web page!

Step 2

Let's build out the "markup" (structure) of our web page so it looks more like our goal, which is something like this.

Here's a description of the what it contains.

1. A main area

2. Inside that there is the message history box
3. Inside the message history box is a list of messages
4. Below that is a form containing two input fields and a "Send" button. The first input field is for the initials of the person sending the message and the second is the message itself.

The initial HTML for this can be something like the following. Replace the content of **index.html** with this code. Please type it out instead of using copy and paste.

```
<main>
  <ol id="history">
  </ol>
  <form>
    <input id="initials">
    <input id="message">
    <button>Send</button>
  </form>
</main>
```

Note a few important properties of HTML here.

Tags define elements

Tags look like `<this>` and they have *meaning*. Examples of tags are `<p>` which means "paragraph" and `<h1>` means a "first level heading".

Most tags need an accompanying closing tag, for example `</p>` or `</h1>` after the text that goes inside the paragraph or heading.

Nesting

We indent our code when that content is *inside* a tag (between opening and closing tags) so that it's easy to understand what code is inside other code.

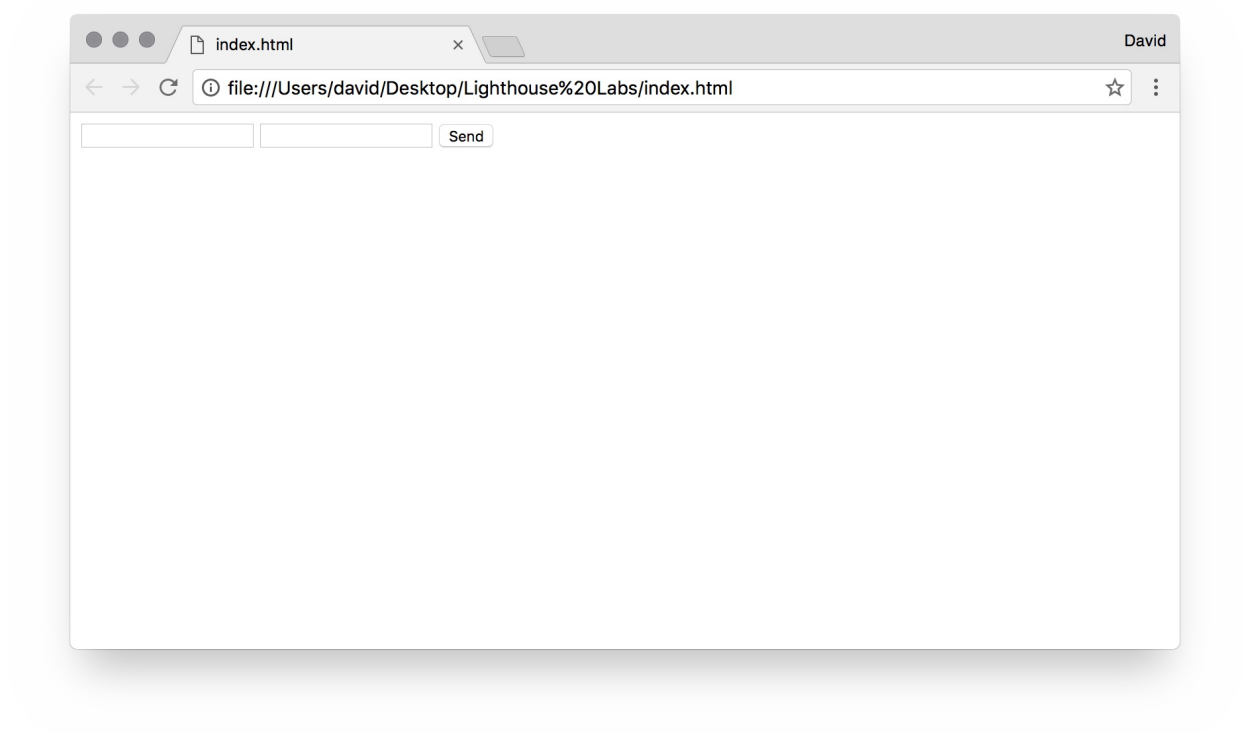
IDs

We give `id` 's to certain elements so that they can be quickly referenced later.

This `id="something"` key-value pair that is part of an HTML tag is called an HTML attribute. Elements can have zero or more attributes. Attributes are cool and come in handy later.

Step 3

Let's save and refresh the browser tab where the **index.html** file is open. It will hopefully look a lot like our final product.



Hm...not quite. Why does it appear this way and not like the example?

We have provided the browser with a good understanding of the *structure* of our Web page, but not how to present it. The presentation rules are defined using another language called CSS.

Questions?

If you have any questions about the above, flag me or another mentor so that we can answer any questions you may have.

Otherwise, it's time to move on to the next section.

CSS

If HTML is the bread and butter of a web page, then CSS is our butter knife.

Why is CSS the knife? Well, it controls *what goes where*. Using CSS we declare how we'd like the content (HTML) to be presented.

CSS stands for Cascading Stylesheets. The key word here is "style", referring to the way something *looks*.

Step 1

Let's prepare our HTML document to *link* a stylesheet (**.css**) file to it.

Add the following HTML tag to the top of your **index.html** file and save it.

```
<link rel="stylesheet" href="style.css">
```

When the HTML file is loaded again, it will ask the browser to fetch another file (called **style.css**) from the Web server and apply its rules to our HTML page.

Step 2

Oops, we don't even have that file in our project, let's add it the same way we created the **index.html** file.

Make sure to call it **style.css** exactly as it is in the tag.

Step 3

Inside the empty CSS file, add the code provided below. I would suggest that you type this out and do it incrementally (save and refresh the page to see the change after each major section instead of typing it all out at once.)

Note that if no changes are reflected in the webpage (upon refreshing the page) then please confirm that your `<link>` tag is correct. Still a problem? Ask for help from your peers or a mentor.

```
body {
  background-color: white;
  color: #303030;
  font-family: 'Helvetica Neue', Arial, sans-serif;
  margin: 20px;
}

main {
  background-color: #f0f0f0;
  margin: 20px auto;
  max-width: 450px;
  padding: 20px;
}
```

Step 4

I'm *not* going to explain to you what this code is doing and how it's working. Discuss among yourself and try and use your powers of reasoning, critical thinking and experimentation to solve the mysteries of CSS syntax. Look for patterns to help identify conventions and rules. HTML has its own completely different set of rules from CSS, but all languages have rules, and computer languages need strict rules. Use this to your benefit to reason toward conclusions.

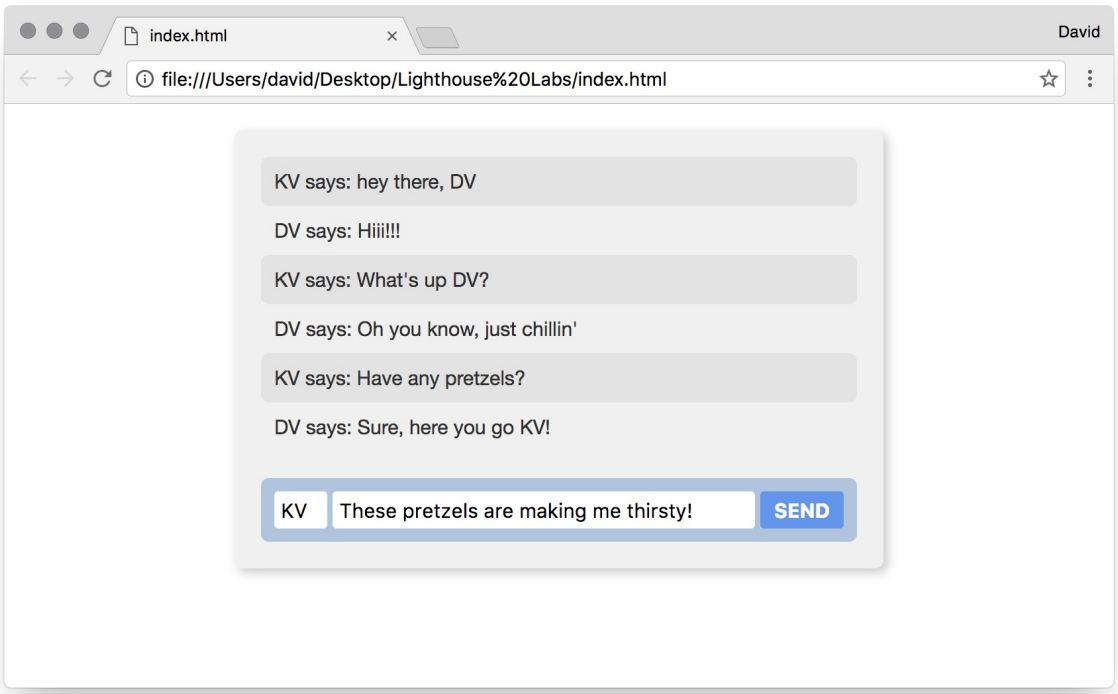
Also, feel free to Google terms like "CSS background color".

The end result should look a little bit closer to what we're trying to get to.

Step 5 - A challenge

Add a few "list items" (for example `<li>DV says: hey</li>`) inside of the `<ol>` element so that you have some fake messages that can be styled. Add at least four of these. They can be removed later when we get into JavaScript.

Now use CSS to try and make the page look closer to this.



If you have time to spare, give it your own unique look.

Properties

Here is a list of CSS properties that you may want to try.

- `width`
- `background-color`
- `padding`
- `margin`
- `border`
- `border-radius`
- `font-size`

Feel free to Google them along with the keyword "CSS" to learn by example. StackOverflow, w3schools and MDN are all great resources that you'll likely end up on naturally. They tend to be highest in results.

You'll likely need a few more CSS properties. Take your time. :)

Example CSS

There is no "one right solution" to CSS. There are many approaches to solving the same thing (admittedly in some cases there are some better than others) not to mention all the creative freedom you get with it.

That said, if you are looking for one possible CSS file so that you can move on, scroll on down!

```
body {
  background-color: white;
  color: #303030;
  font-family: 'Helvetica Neue', Arial, sans-serif;
  font-size: 15px;
  margin: 20px;
}

main {
  background-color: #f0f0f0;
  border-radius: 6px;
  box-shadow: 3px 3px 12px #c8c8c8;
  margin: 20px auto;
  max-width: 450px;
  padding: 20px;
}

ol {
  list-style: none;
  margin-bottom: 20px;
  margin-top: 0;
  padding: 0;
}

li {
  padding: 10px;
}

li:nth-child(odd) {
  background-color: #e3e3e3;
  border-radius: 6px;
}

form {
  background-color: lightsteelblue;
  border-radius: 6px;
  margin: 0;
  padding: 10px;
}
```

```
button {
  background-color: cornflowerblue;
  border: none;
  border-radius: 3px;
  color: white;
  font-size: 15px;
  font-weight: bold;
  padding: 5px 10px;
  text-transform: uppercase;
}

input {
  border: none;
  border-radius: 3px;
  font-size: 15px;
  padding: 5px;
}

#initials {
  width: 40px;
}

#message {
  width: 315px;
}
```

Part II

JavaScript: jQuery, Node, Express, and Web Sockets

JavaScript in the Browser

Let's add some behaviour to your webpage so it actually does something useful. That's where JavaScript comes in.

Step 1

Much like with the CSS, we need to link our new **.js** code with our web page, our **.html** file.

Add the following tags to the *bottom* of the HTML file.

```
<script src="https://code.jquery.com/jquery-3.1.1.js"></script>
<script src="app.js"></script>
```

Now whenever you refresh/load the web page, it will reference two separate JS files for the browser to download (at the end), one of which is external (3rd party) to our application. This 3rd party library called jQuery is there to make life easier. We don't **need** it, but with it we can write less code. This is why most websites use jQuery or other libraries like it.

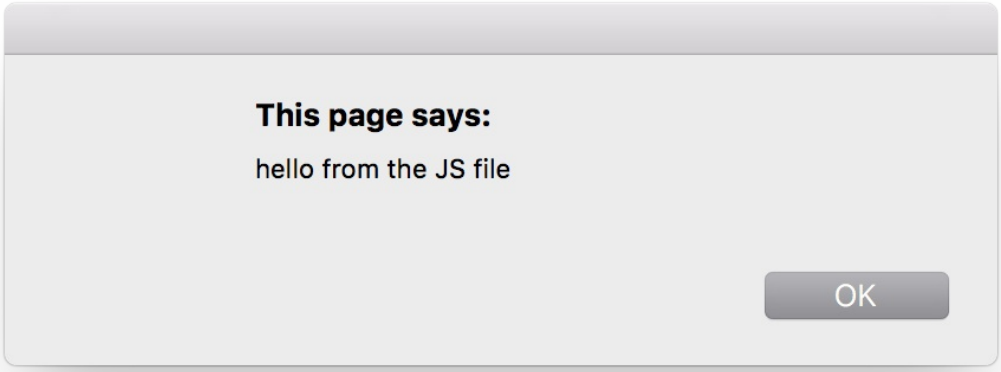
Step 2

The second file **app.js** needs to exist in our workspace, so let's create it just like you created **style.css** earlier.

It will be empty at first, but let's add the following code into it.

```
alert('hello from the JS file');
```

Save the file and refresh the HTML file in your browser. When the page loads you should see an alert pop up that says your message.



`alert` is a built in function that all browsers support, even though they may look slightly different on each browser. I'm guessing it's not your first time seeing one, so now you know how they happen.

Any time you call a function in JS you have to use parentheses `()` after it, and within the brackets you put in data that you want to give the function. Programming functions are much like math functions. They take in values and can do some crunching and give you back a computed value, or in this case, give you back some behaviour like popup that message you passed into it.

Every single predefined JavaScript function has documentation, so we can look up details about how they work. Case in point.

Very soon, we'll create our own function while using other functions. It's going to get a bit more real, so hold on to your suspenders!

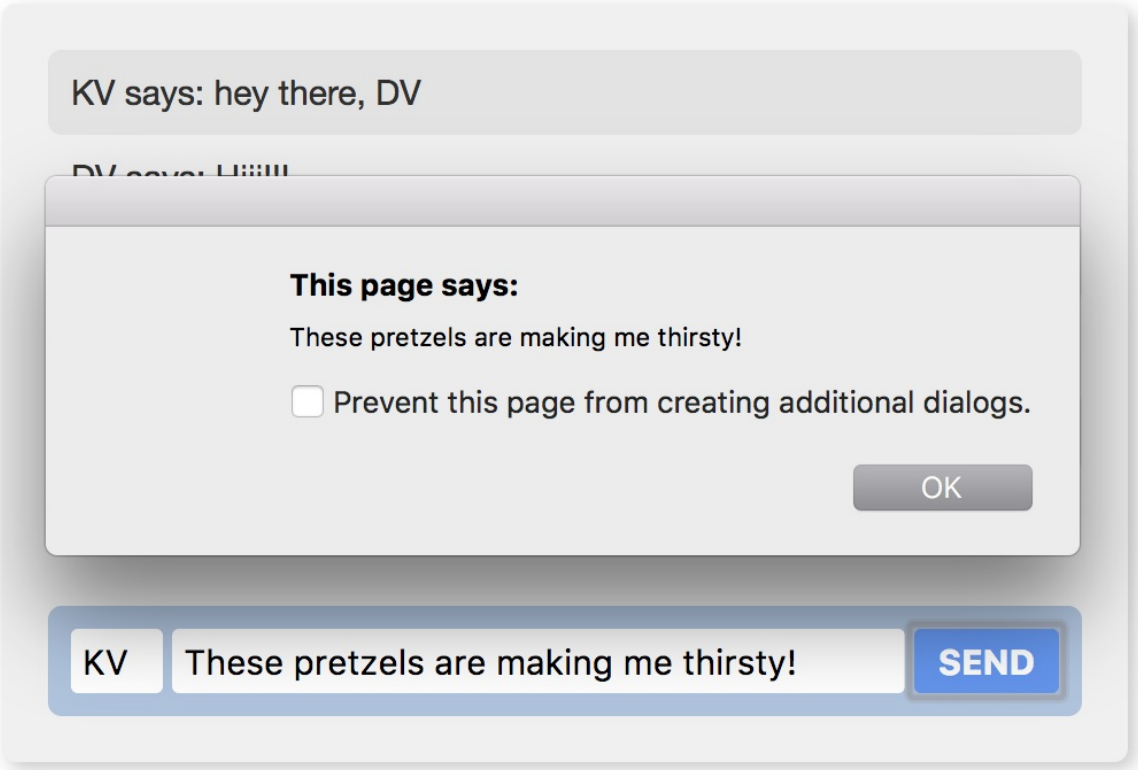
Step 3

Remove the `alert` code and let's get down to business. We want to make it so when our `<form>` is submitted (via enter key or by clicking the send button) we read the text content of the input field (with `id="message"` in the HTML file) and do something with it. For now let's just `alert` it.

```
$('form').submit(function () {
  var text = $('#message').val();
  alert(text);
  return false;
});
```

Remember, type that out yourself. Don't copy paste it. Assuming you started and closed all the brackets, quotes, and semicolons correctly, it should add some interesting behaviour to the page.

Save the file, and refresh the HTML page. Put some text into the second field and hit Enter or click Send. You should see it echoed back to you in an alert message. If not, review or otherwise seek help.



Explanation

The code above isn't doing too much, but it does need to be reviewed before we move on.

First, we use **jQuery** using the `$` function, telling it that we want to target all `form` elements on the page. There is only one form element on the page and it is the one at the bottom which contains both input fields and the button.

Note how we don't use the `<` angle `>` brackets when targeting elements here, just like with CSS. Angle brackets are solely used to *define* tags in the HTML page.

We then call/invoke another function called `submit` on that returned form element, saying that we want to be notified anytime that form is submitted.

In order to be notified any time this happens, we pass in our own custom function into the `submit` function. That's right, `submit` is a function that accepts a function. Read that again. Look at the code. Remember that we pass in data into functions within the parentheses `()`. They're not on the same line, but they are there.

This part may be a bit confusing but perhaps this annotated code can help you see it better.

```
1 $('form').submit(function () {
2   var text = $('#message').val();
3   alert(text);
4   return false;
5 });
6
```

Otherwise, call over a mentor for explanation.

Inside our function, the rest of the code is indented so that we know that it is within that function, just like we nest HTML tags with indentation. Make sure your code is indented correctly because improper indentation is a big source of confusion.

This inside code is capturing the text value of the HTML element with the ID `message` and then passing it to `alert` so that we can see it.

Lastly, the `return false` is there to tell the browser to cancel the original form submission logic (which is to attempt sending the form to the server, with loading spinner and everything.) This is common practice when adding custom behaviour to forms like we are doing here.

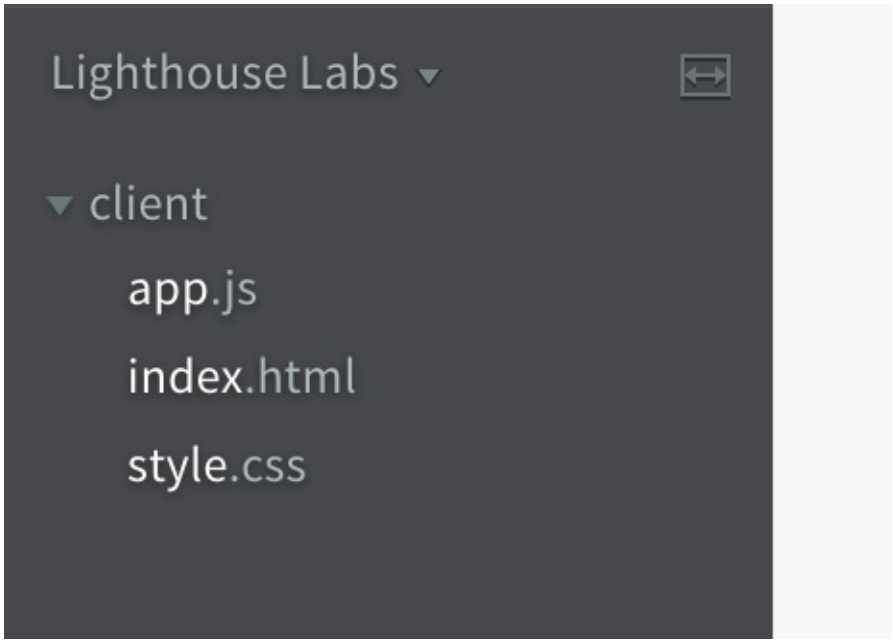
Node (JavaScript on the server)!

So far we've been building a website with some static content and a bit of client-side interactivity.

It's time to take it up a notch and graduate to "Web application" status. Server-side app logic is what separates a site from an app.

Step 1

Move all the "client" side files into their own folder called **client**.

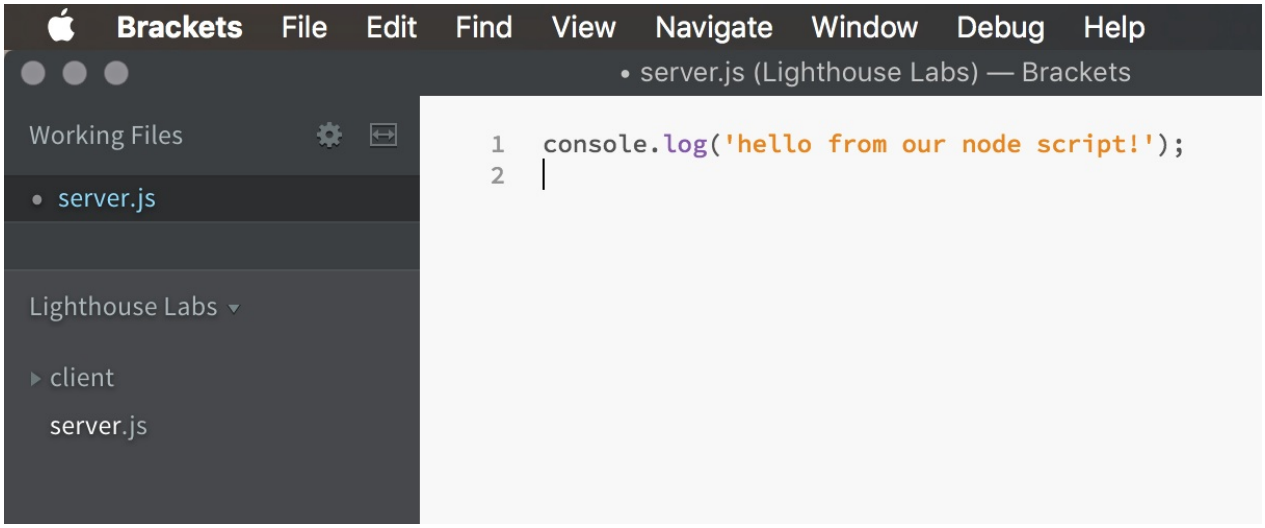


Step 2

Now let's add another JavaScript file to be run on the server side which will power the logic for our chat application. Call it **server.js** and put it at the top level (at the same level as the **client** folder.)

In it, put this one test line of code, to print a message to the screen so we know that it is working.

```
console.log('hello from our node script!');
```



Step 3

Let's run it. Open up the **Terminal** app on MacOS or **cmd.exe** on Windows to get to a command prompt.

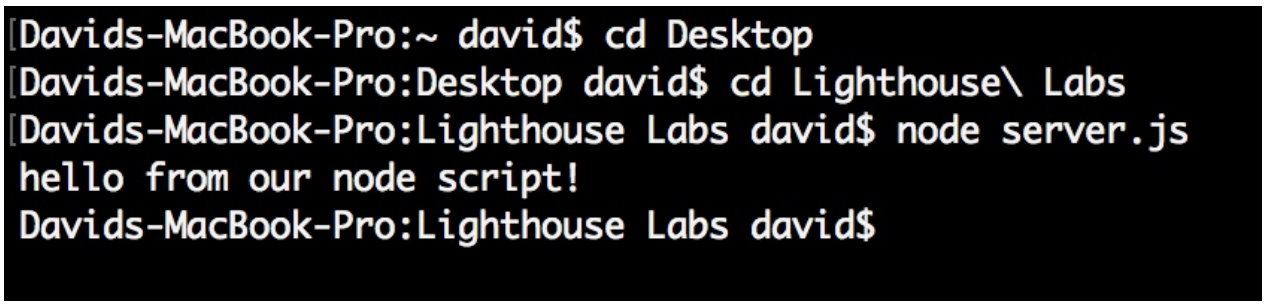
Next you'll have to change the working directory to the one where your project files are. In the example below I used the commands:

```
cd Desktop
cd Lighthouse\ Labs
```

Then to run the file I did the command:

```
node server.js
```

The output should look like this.



And there's our message, nice!

Express for our web server

So far we've created a Node script that prints something out to the terminal, which is great. Instead of doing that though, we need it to do something real.

In our case, we're building a web server, more specifically a *chat 'server'*. So we need to listen for incoming HTTP requests and then send down the HTML, CSS and JavaScript that we wrote earlier for our chat client.

While Node comes with basic HTTP handling support, a very (probably the most) popular 3rd party library (called "modules" in the Node community) named **Express** is used by most developers writing Web servers in Node. We shall use it too!

Step 1

Let's install and save the Express library into our project by running this command from the command line that we have open.

```
npm install express
```

This downloads some code using the Node Package Manager (NPM) that we can use to make our lives easier.

```
Davids-MacBook-Pro:Lighthouse Labs david$ npm install express
/Users/david/Desktop/Lighthouse Labs
└─┬ express@4.14.0
  ├─┬ accepts@1.3.3
  │ ├─┬ mime-types@2.1.14
  │ │ └── mime-db@1.26.0
  │ └── negotiator@0.6.1
  ├── array-flatten@1.1.1
  ├── content-disposition@0.5.1
  ├── content-type@1.0.2
  ├── cookie@0.3.1
  ├── cookie-signature@1.0.6
  ├─┬ debug@2.2.0
  │ └── ms@0.7.1
  ├── depd@1.1.0
  ├── encodeurl@1.0.1
  ├── escape-html@1.0.3
  ├── etag@1.7.0
  ├─┬ finalhandler@0.5.0
  │ ├── statuses@1.3.1
  │ └── unpipe@1.0.0
  ├── fresh@0.3.0
  ├── merge-descriptors@1.0.1
  ├── methods@1.1.2
  ├─┬ on-finished@2.3.0
  │ └── ee-first@1.1.1
  ├── parseurl@1.3.1
  ├── path-to-regexp@0.1.7
  ├─┬ proxy-addr@1.1.3
  │ ├── forwarded@0.1.0
  │ └── ipaddr.js@1.2.0
  ├── qs@6.2.0
  ├── range-parser@1.2.0
  ├─┬ send@0.14.1
  │ ├── destroy@1.0.4
  │ ├─┬ http-errors@1.5.1
  │ │ ├── inherits@2.0.3
  │ │ └── setprototypeof@1.0.2
  │ └── mime@1.3.4
  ├── serve-static@1.11.1
  ├─┬ type-is@1.6.14
  │ └── media-typer@0.3.0
  ├── utils-merge@1.0.0
  └── vary@1.1.0

npm WARN enoent ENOENT: no such file or directory, open '/Users/david/Desktop/Lighthouse Labs/package.json'
npm WARN Lighthouse Labs No description
npm WARN Lighthouse Labs No repository field.
npm WARN Lighthouse Labs No README data
npm WARN Lighthouse Labs No license field.
Davids-MacBook-Pro:Lighthouse Labs david$
```

Now we can start using it. Type out the following JavaScript code into the **server.js** file.

```
var express = require('express');
var app = express();

var http = require('http');
var server = http.Server(app);

app.use(express.static('client'));

server.listen(8080, function() {
  console.log('Chat server running');
});
```

Explanation

So a bunch of new code and syntax was just introduced there. Let's attempt to break it down.

The first few lines are just loading external modules. We then tell the Express app to server all static content from the **client** folder.

When all the setup work is done, we then tell the server to listen in on a certain port (in this case 8080.)

So for now it's a simple web server serving only static (HTML, CSS and JavaScript) files to any HTTP/Web traffic coming its way.

Let's try it out by running it using the same command as before.

```
node server.js
```

Now visit http://localhost:8080 in your Web browser. It will bring up the chat app page just like before, with no new functionality. The send button should be working as before, too!

Socket.IO for Real-time messaging

Now it's time to add the last missing, yet crucial piece to our app: *chat functionality*!

To do this, we will leverage yet another 3rd party Node module. It's called Socket.IO.

Check it out here: http://socket.io/.

Socket.IO will leverage a technology that browsers have called Web Sockets.

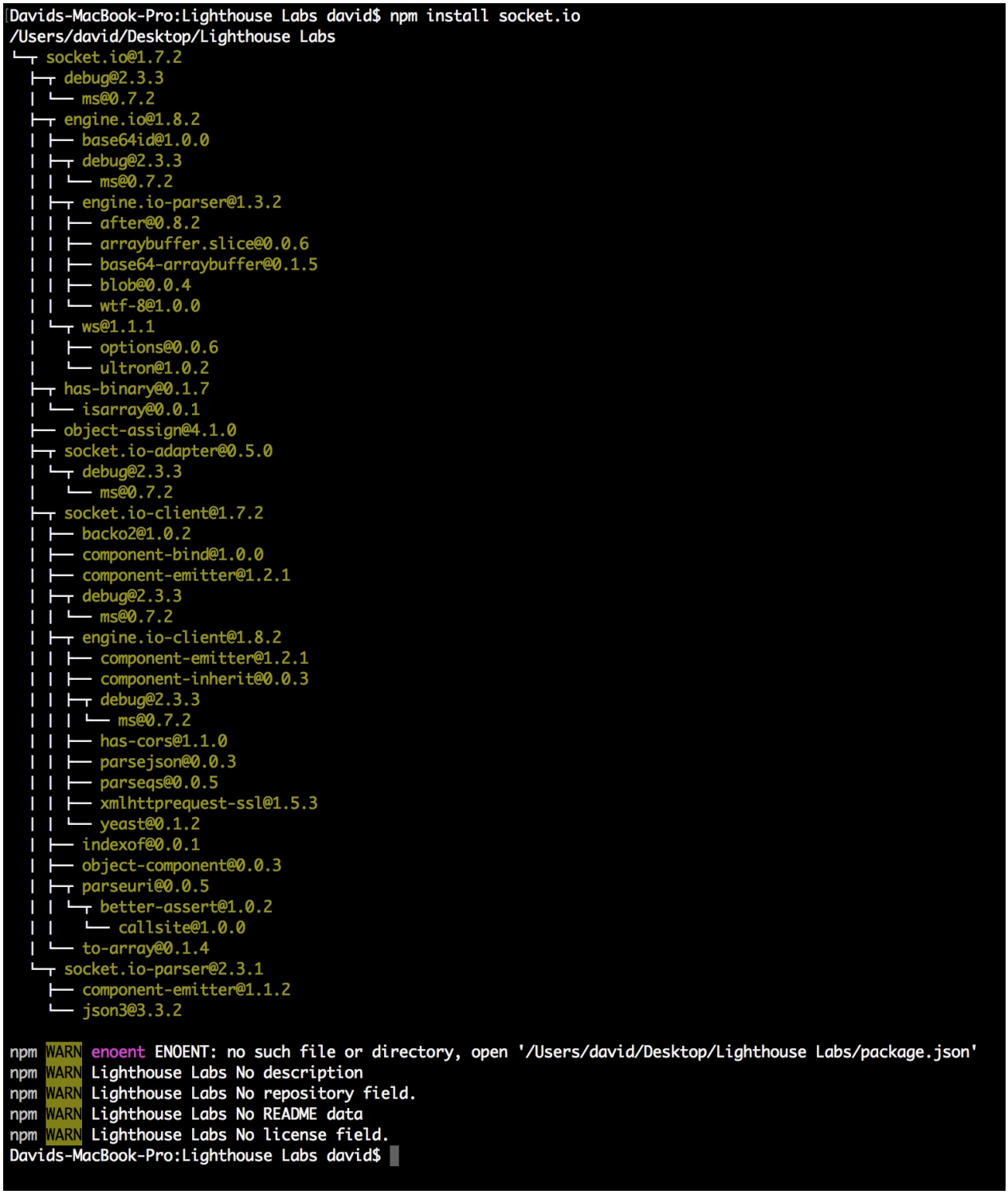
What's nice about Socket.io is that it will let us write similar JS code on both the client (browser) side and at the server (Node) side.

Step 1

Run the following command in the terminal window to install it on the server.

```
npm install socket.io
```

The output will look something like this.



Step 2

Now let's add some code in **server.js** that will use this library. This code block should be placed *before/above* the line of code that starts with `server.listen`.

```
var io = require('socket.io')(server);

io.on('connection', function (socket) {
  socket.on('message', function (msg) {
    io.emit('message', msg);
  });
});
```

Step 3

Next, we need to add similar logic to the client app.

Open the **index.html** file and modify the `<script>` tags so that we also reference Socket.IO. We can do this by adding one more script tag before our `app.js` script tag, so that it looks like this.

```
<script src="https://code.jquery.com/jquery-3.1.1.js"></script>
<script src="/socket.io/socket.io.js"></script>
<script src="app.js"></script>
```

Step 4

Open **app.js**, our client-side JS code file, and let's add some code to send and receive messages from the browser.

Add the following code to the very top of the file.

```
var socket = io();
```

This is saying that `socket` is now a reference to the Socket.IO library.

Step 5

In that same file, *replace* the line that starts with `alert` line from before with the following code.

```
  socket.emit('message', text);
  $('#message').val('');
```

The code above says to emit the textual message to the server instead of performing our temporary `alert` behaviour. The second line in the code simply clears the input so that another message can be typed by the same user.

We're not done yet, we need to listen for messages that are received from the server and append them into the message list. Add the following code at the *bottom* of the file.

```
socket.on('message', function (msg) {
  $('<li>').text(msg).appendTo('#history');
});
```

This part tells the browser that any time a message is received from the real time web socket connection with the server, create a new `<li>` (list item) HTML element and append it to the message history `<ol>`.

Final code for app.js

The **app.js** file should look like this.

```
var socket = io();

$('form').submit(function () {
  var text = $('#message').val();
  socket.emit('message', text);
  $('#message').val('');
  return false;
});

socket.on('message', function (msg) {
  $('<li>').text(msg).appendTo('#history');
});
```

Final code for server.js

The `server.js` file should look like this:

```
var express = require('express');
var app = express();

var http = require('http');
var server = http.Server(app);

app.use(express.static('client'));

var io = require('socket.io')(server);

io.on('connection', function (socket) {
  socket.on('message', function (msg) {
    io.emit('message', msg);
  });
});

server.listen(8080, function() {
  console.log('Chat server running');
});
```

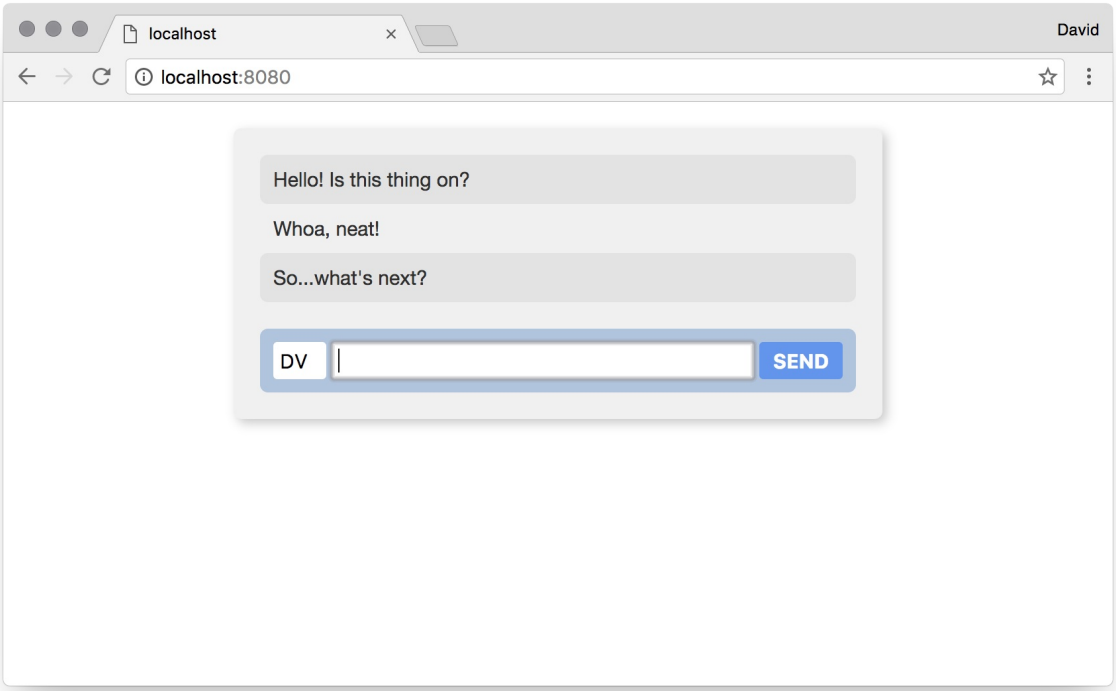
Whoa, it works!

Make sure all your files are saved, and the Node server is still running, and give it a shot.

That's right, it works! That's all it took. The basic chat functionality works. Have your peer go to the same URL that you're on and you guys should be able to communicate!

Note: newcomers to the chat room can't see any previous messages. We would have to implement that functionality for it to work that way.

Screenshot


localhost
David
localhost:8080
Hello! Is this thing on?
Whoa, neat!
So...what's next?
DV
SEND

Enhancements as Challenges (Stretch)

You may have noticed some bugs and broken features with this app.

1. Initial field - Part I

The initials input field (the first input box) is not being used at all. Modify the **app.js** so that it sends (emits) the text as something like `"DV says: hello"` where `"DV"` is from the first input field (with `id="initials"`) and `"hello"` is from the message field.

2. Initial field - Part II

The initials input field is actually meant to be 2 letter initials only. Make it so only a maximum of two characters can be entered into that field.

Hint: Google "html input field maximum length" for a convenient HTML attribute that can be added to your HTML file.

3. List of Messages

It would be nice to see the message history on the server anytime you join the chat room. Unfortunately, we don't keep/store a message history on the server at all.

One way to do this is to create an empty array that stores the messages.

Then, whenever a message is received on the server, it can push (append) that message string into the array.

Then whenever a user (new socket) connects to the server, (on `connection`) the server could iterate over all the messages in the array and `emit` them to that new client/socket.

4. Deploy

You can use another free technology called Heroku to make your app available on the Internet for anyone to use. There's a tutorial that will walk you through the process here: https://devcenter.heroku.com/articles/getting-started-with-nodejs#introduction.

Final Thoughts

This demonstration was a bold attempt at showing you how easy it is to get started with Web development, and that seemingly complex apps can be prototyped relatively quickly with modern software technology and approach.

Thank you for spending your Saturday with us; I hope you enjoyed it and that you have wonderful weekend!