API Documentation

August 10th, 2022

Table of Contents

Contents

239 Why Do We Need APIs

Hey guys. In this module we're going to be talking all about APIs, Application Programming Interfaces. Now it's very likely, if you spend any time on the Internet at all, you would have heard of this term crop up every so often. So in the coming lessons of this module, we're going to explore what exactly APIs are, and what we can do with it.

Build

Weather Application & Email Signup Web Application

First things first. In this module you're going to be exploring a number of APIs, and you're going to be building a weather application, as well as a email sign up web application. This is going to allow you to collect real users' email addresses, and save it to your own database on MailChimp, so that you can contact them at a later point.

And this is really useful if you decide to create a landing page and email people when your product launches, or if you're trying to set up an email newsletter and stay in touch with your subscribers. So that's the goal.

APIs

What are APIs, and why are they useful?

Now, in order to get there though, we first have to understand what are APIs, and why are they useful. Now, if you went onto yahoo.com, and you took a look at their little weather section on the right, you'll notice that it takes the current location of your web browser, and gives you the upcoming weather forecast for your location. Now, if we wanted to create our own web site, and we wanted a little module like this, how would we do that? Because we're not going to go around collecting weather data, right? We're not really kitted out to be able to detect wind speed and the temperature and forecast what the weather will be like in the future.

Fetch Data from other Logins

So where do we get this data anyways? Well, we could use a weather data service, like OpenWeatherMap, where their job is to collect these pieces of data and organize it and do all of the complicated things to forecast the weather, and then all we have to do is to use their API to tap into that live weather data, and we'll be able to start pulling it into our web site, just like how Yahoo does it here.

Now similarly, when you go onto Tinder, and you've seen that section where it says 'Shared Friends' or 'Shared Interests', where did these pieces of data come from? Well, it comes from when the user logs in with Facebook. Tinder is able to ask Facebook for these pieces of data on the user, and then it can use that data to populate these sections, and it will do that through the use of the Facebook API in order to get that data from Facebook. Now out there there are loads of different types of APIs that you can use. For example, in the UK we have the Police API, which gives us granular data on where certain crimes have happened, on things like what happened with that crime, who was involved, and a whole lot of things. And through the use of an API like this, people are able to build interesting web sites, like this one.

This is called Murder Map. It has a little bit of a dark name, and it's a little bit of a strange web site, but it takes that data from the police API, and it maps out all of the murders that take place in London by location as well as by weapon, by date, and by the age and gender of the victim. And it's really interesting to see this data mapped out, especially if you're trying to figure out where you want to live. So maybe stay away from gun and knife crime, but maybe poisoning and other things won't affect you so much.

Upcoming Lessons

So in the coming lessons, I'm going to be introducing you to some really fun API. But there's all sorts of APIs out there, and they range from things such as APIs that give you data on the prices of various stocks, to things like an API that gives you data on various Pokémon, so all the way from very serious to very not serious.

You can access all sorts of data via APIs and use them to create your own web sites. Now we've talked so much about APIs. What exactly is the definition of an API?

API Definition

A broad definition that a lot of people go by is that:

> an API is a set of commands, functions, protocols and objects that programmers can use to create software or interact with an external system.

So what does all of that mean? Well, let's simplify it.

A Practical Example

APIs as the Waiter of a Restaurant

Let's say that you were going to a restaurant, and in a restaurant we know that there are the things that are on the menu, say various cakes and desserts that they sell, but you could also go into the kitchen and you would find there's loads of raw ingredients, and probably a couple of surprised chefs as to what it is you're doing in the kitchen.

Now if you had just decided that you were going to go into the kitchen pantry and just start eating things that they have they're, like just start spooning some mayonnaise, then the restaurant's going to be pretty shocked, right, and they're going to be pretty unhappy about that, because there are certain things which they sell and they will let you buy, but then there's other things which are kind of off limits.

Allow access to data

It's kind of similar with data. For every web site that has their own data, be it Facebook, which has data on their users, or something like the police API, which has data on crimes, there's certain pieces of data that they will allow you to access, but there's other pieces of data that's not really your business.

Documentation & Contracts

So how can these web sites tell you what are the things that you can actually access from them and how to do that? Well, if we were in a restaurant then they might provide that information in the form of a menu, right? So at this restaurant you can buy cakes, you can buy sandwiches, tea and coffee. And for a service like a weather API, say like OpenWeatherMap, then the kind of data that you can access include things like the temperature, the weather conditions, the weather images, and maybe the atmospheric pressure.

An API, some people will consider to be a contract. It's a contract between the data provider and the developer. And essentially what it says is these are all the things that developers can access, and these are the methods, the objects, the protocols that you would use to access them. And we, as the web site that hosts the data, will try to not change any of these methods or functions without notifying you.

An API is a set of commands, functions, protocols and objects

Coming back to our definition, then it starts to make a little bit more sense, right, where an API is a set of commands, functions, protocols and objects that programmers can use to create software or interact with an external system.

Now we've seen a lot of examples of an API that allows you to interact with an external system, but we haven't really seen any that allow you to create software, or have we?

Well, if you think about it, when we learnt about jQuery, what is jQuery? Well, it's an API, right? It's something that gives you access to a whole bunch of functions and objects that let you create software much more easily than if you were just writing vanilla Javascript.

APIs interact with external systems

Now if we think back to the Tinder example, where Tinder was getting data from Facebook to get the shared friends and shared interests of their users, well this is a case where they're using the Facebook API to interact with an external system, namely the Facebook database.

Now in this module we're going to focus on the types of APIs that allow us to interact with an external system, and most importantly an external server. So we're going to get hold of some piece of data from a web site, and we're

going to do that through their API, and we're going to read their documentation to see what are all the things that we can do. And we're going to use their API as a menu of things that we can do to interact with their data.

So that's a little bit on the theory of API, but in the next lesson we're going to look at how we can put it into practice. For all of that and more, I'll see you there.

240 API Endpoints, Paths and Parameters

Now in the last lesson we looked at how an API works, at least from the theory point of view.

Now in this lesson, I want to talk about how to actually put it into practice and use an API to get some data back from a couple of web sites.

APIs: Endpoints, Paths, Parameters, Authentication

Now, when we're talking about APIs, you will often need to think about things including endpoints, paths, parameters, and authentication. These are the four things I want to cover, and we're going to use a few APIs so I can show you what each of these things do.

Endpoints

The first thing I want to show you is an API endpoint, and every API that interacts with a external system, like a server, will have an endpoint. So who is this in the photo? Well, it's our friend Kanye Rest. So this is a free REST API for random Kanye West quotes.

api.kanye.rest

This web site hosts a database of Kanye West quotes, and we can access the data from this web site by using heir API. And the endpoint of the API is this URL right here, api.kanye.rest. And whenever you're using a different API, they're going to likely have a different endpoint, but they'll always tell you what it is in their documentation. Now because this is such a simple API, they've only got one thing that you can do with it, which is to get a random quote.

Let's go ahead and see what data we can get from this. So if I take this URL and I paste it into my browser bar, then my browser makes a get request to the kanye.rest server, and they send back a piece of data, which is a quote, and it says, "I hate when".

And the quote is, "I hate when I'm on a flight and I wake up with a water bottle next to me like oh great now I gotta be responsible for this water bottle". Classic Kanye.

Let's say that we created a web site where you could go onto the web site and every day you get a different Kanye quote. Well then you can see that we could probably do that just by using this simple API.

Paths And Parameters

sv443.net/jokeapi/v2

Now usually the web sites that have API tend to have more data than just a bunch of random quotes. In addition to endpoints, there's also API paths and parameters that you can use in order to narrow down on a specific piece of data you want from an external server. In order to illustrate parts and servers, we're going to use a joke API, which is an API that allow you to access a database of random jokes, and through the use of path and parameters, we're going to customize the kind of jokes we want to get back. In the course resources, you'll find a link to the joke API, or you can simply type out this URL (sv443.net/jokeapi/v2).

Now this joke API is a little bit more complex than the kanye.rest API. It's more complicated because it has more options for you as the programmer. So, for example, you can choose which category of jokes you want.

Do you want any sort of joke or do you want just programming jokes or do you want just dark jokes, or if you wanted to blacklist jokes that are not safe for work, or religious, political. You can change the joke type: a single joke or a two part joke, and you can even search for a particular string that is contained in the joke.

So where does all of this customization go if we wanted to implement it? Well firstly, we need to figure out the end point, which is the starting URL.

So if we scroll down in the documentation, you can see there's a whole bunch of endpoints, and this end point is the one that you use to get a joke.

So the end point looks like this. But notice how if we just put this URL into our browser, you can see we get an error. It says, "No matching joke found", "The specified category is invalid".

So what's going on here? Well, it's because, even though this is the end point, this is the starting URL, we have to add one other thing to complete it, which is the category or categories of jokes.

Try it out section

Now if we go up here and we use the 'Try it out' section, you can see that if we chose Any category, it goes to 'joke/Any', if we chose just the Programming jokes, so we check that, then it goes to '/Programming'.

So the difference here is that we need to specify a specific path after the end point.

So if we see this as the root of the tree, we have to add a branch. And let's say that our branch was the Programming branch.

So now it's going to give us a random programming joke, and it's a two part joke. "Why did the functional programming developer get thrown out of school?" "Because he refused to take classes." Well, that's a really bad joke.

Paths

/sv443.net/jokeapi/v2/Programming

Now we're introducing this concept of paths when we're trying to access an API. And notice how the end point is the URL up to that last forward slash, and then if we went down the Programming path, then we would only be getting programming jokes, but if we change the category to Dark, then we would only be getting jokes from the Dark category.

Now remember when we were learning about Node and creating our backend, we know that in order to create a new path, we have to plan for it ahead of time. So somewhere on this joke API server, they have to be able to catch when a request is made to this URL/Programming path, and then filter through all of their jokes and give us a random programming joke.

Now sometimes we might want to get a piece of data from an API that is something that they can't plan for. So, for example, if I wanted to search for a joke that contains a specific word, let's say I wanted a programming joke that contained the word 'debugging', well then they probably wouldn't have thought of this ahead of time.

Parameters

They probably don't have a path to address this specific query, so in order to allow the API to be flexible enough to deal with custom queries like this, usually APIs allow you to provide parameters.

And parameters go at the end of the URL, after a question mark, like this.

And then there's a key value pair that goes into the URL.

> The key is called 'contains', so that's basically our search string. And then after an equal sign is the query. Notice that's exactly what I typed in in here.

Now in this 'Try it out' section here, you'll notice that there are other parts which are also queries

For example, if I wanted to blacklist all the Not Safe For Work jokes, then you can see that that is a blacklist flag, and then it's equal to 'not safe for work'. And if I wanted to get rid of all the two part jokes from the jokes that I get back, then we have a type equals 'single'.

And notice how if we have more than one query, so here we've got one, two and three queries, the first query follows a question mark, and every subsequent query follows an ampersand, or an and symbol, like this.

Use symbols to specify parameters

So when you want to specify a parameter, remember that you need these symbols in the URL to be able to do that. The parameters come after a question mark, and then they are set as a key value pair with an equal sign in between.

And if you want to have more than one parameter, you separate each of the key value pairs with an ampersand symbol.

Let's see what we get if we go ahead and use this particular request. So I'm going to paste that into my browser, and I'm going to make this request from my browser to the joke API server.

So I've got the category which is 'Programming', because I've used that 'Programming' path there, and then I've blacklisted all the jokes that are not safe for work, I have specified that I only want a single part joke, and I want something that contained the word 'debugging'.

Joke: The six stages of debugging

So this is the joke I get back. "The six stages of debugging: 1. That can't happen. 2. That doesn't happen on my machine. 3. That shouldn't happen. 4. Why does that happen? 5. Oh, I see. 6. Man, how did that ever work?"

So now you can see, through the use of paths and parameters, we're able to narrow down on the data that we get back from the joke API to the precise thing that we want.

Now that you've seen basic APIs in action, and you've seen how we can use paths and parameters to get specific pieces of data from an API, I want you to have a play around with the joke API.

Take a look down at the documentation, see what sorts of things that we can get back from the API, and try formatting your URLs and to make some requests using your browser.

Authentication

So if you wanted to get a joke that contains the word 'arrays', or say if you wanted to see which categories are available from the joke API, have a play around with it, and make sure that you understand what's going on here when we're typing out this long URL. Pause the video now and try to give that a go. On the next lesson, we're going to talk more about authentication and decoding some of this data that we get back.

Pause the video now and give that challenge a go.

241 API Authentication and Postman

Now in the last lesson, we saw how we could work with the kanye.rest API and the joke API to start querying these external web sites for pieces of data that they may have.

We saw how we could use paths as well as parameters to narrow down on the exact type of data that we want. And we saw how we can make these requests using a formatted URL and putting it through our browser.

Now both of those web sites had very simple pieces of data, either a database of jokes or a database of Kanye West quotes.

Now when we come to data that's more monetizable, or allows developers to build more complex applications that might be used by hundreds or thousands of users, then these web sites have to start thinking very carefully about how to either monetize your use of their data or how to limit your use to a threshold.

Authentication

And the way that they would do that is through authentication.

So every time you make a request through the API, they have to be able to identify you as the developer, and they have to keep track of how often you're using their server to get data, and then charge you, or limit you, accordingly.

In order to illustrate this concept of authentication, we're going to use the openweathermap. org's API. And if you take a look in their pricing, they tell you that it's free to use their API as long as you don't make more than sixty requests per minute. So every single time we typed in this particular request to an API and we hit enter, that's a single request. And if you make more than 60 a minute, so more than one a second, then they're going to limit any further requests.

So for most developers to get started building your app or your web site, this is more than enough. But as you start getting more users, you have more traffic, and you're probably likely to have more revenue, then these data providers will also start charging you to use their data.

But the way that we would implement authentication in any of these categories is exactly the same. So let's see how we can use an authenticated API like this one.

So the first thing to do is to go ahead and sign up to the OpenWeatherMap, if you haven't already.

So you're going to create a new account. And don't worry it doesn't require any credit card details or anything other than a user name and email. Now once you've signed up then go ahead and sign into your account.

And once you're signed in you should see a page that looks like this. And up here there are a whole bunch of tabs and you want to tab over to the API key section.

Now here you can create a new key if you don't see a default one here. Once you've created a key, then you're just simply going to copy this entire string, and you're going to be using it when you make any requests to the API.

So let's see how we can make a request to this weather API. Let's take a look at the API docs for getting the current weather data.

Now if you read through this documentation, you'll see that there's a whole bunch of ways that you can get weather data, either querying by city name, or by a city ID, or by the geographic latitude and longitude values, and a whole lot more.

Now let's say that I want to use the simplest, which is to query by a city name.

Then you can see, if we click on the example API call, that they've got some sort of end point that ends here, and then begin the parameters, right, because we said the parameters start with a question mark.

So the first parameter has a key of q, which probably stands for query, I would guess, and then the name of the city, and then the next parameter is the app ID, which corresponds to the API key that we've got just now when we signed up.

So this, overall, is the structure of this sample API call. Now the reason why it's a sample is because here they provided you a demo API key.

Now in order to make an actual API call, we have to look at that actual endpoint, and it actually looks more like this. Instead of samples.openweathermap, it's actually api.openweathermap.

So if I change this from samples to api, then you'll see that this won't work with the app ID that they provided by default, and we get the error message "Invalid API Key".

Now remember that earlier on we already signed up and we actually got our very own app ID, and you should have done this as well.

Well in that case, if you simply paste that app ID here, after the equals sign, so this is the key value pair here, then we go ahead and make our request by hitting enter, you can see we get our data back. And this is the actual data for the current weather in London. And as you can see, as usual, there's a few clouds in the sky. This is the default weather for London pretty much, but it's not too bad.

You can try changing this q to a different city.

Parameters

Let's check what the weather is like in Paris. And we get some temperatures back, and we get some wind speeds. We get humidity and atmospheric pressure. But you might notice that the temperature is kind of weird, right? It seems like Paris is burning. 283 degrees. Even if that's Fahrenheit, that's still too high.

What's going on here? Their default unit for temperature for OpenWeatherMap is actually Kelvins. And in order to get either imperial, so Fahrenheit, or metric, Celsius, we have to add another parameter.

So depending on whether if your brain works in metric or imperial, go ahead and try to add these parameters to our API request, so that you get back the temperature in a format that you understand. Pause the video, and give that a go.

Key Value Pairs

All right. So we know that the first parameter gets added after a question mark, and it has to be added in a key value pair separated by an equals sign, but every subsequent parameter gets added after an ampersand.

So if we want to add another parameter here, we're also going to add an ampersand, and then the name or the key of the parameter is called units, and then the value could be imperial or metric.

So I'm going to add 'units=metric' because I want the results back in Celsius.

And you can see that Paris is currently 10.3 degrees. That sounds a lot more reasonable.

Now remember that the order at which you put these parameters doesn't matter. So, for example, I could have the query first, which is Paris, and then have my units, which is metric, and finally have my app ID like this, and I still get the same results. The order doesn't matter at all, as long as they're separated by these ampersands, or, if it's right at the beginning, the question mark.

Postman

Now notice how, as we start using more parameters, and especially when we have parameters with very long values like this, it gets very difficult to test our APIs using a browser, and editing these URLs.

It's very very fiddly, and you can often make very simple typos, because it's hard to see where each parameter ends and the next one begins.

So very often, when we're testing APIs, we'll use a tool called Postman.

It's completely free to download, and it's available for Mac, Windows and Linux.

So I recommend heading over to postman.com/downloads, and download the version that's right for your computer.

Now once you've done that, you'll end up with an application that looks something like this. So let's see how we can make that same request using Postman.

GET Requests & Queries

So I'm going to click on this plus sign to make a new request, and it's going to be a get request.

I'm going to try and get some data from our OpenWeatherMap.

And I'm basically going to try and do the same thing as before, so I'm going to use the API end point, which goes up to the first question mark, and I'm going to paste that in the URL, and then I get to add all my parameters.

So the first parameter is q, because that's the parameter to query by city name. Now if you wanted to query by latitude, longitude, or by a zip code, then you can use these different parameters. But in my case I'm just going to stick with the simplest. So I'm going to put the key as q and the value as London.

URL Structure & Key Query Checkboxes

Notice how at the top here, it's starting to structure my URL for me, while I get to work with this much clearer user interface, which has a table, and has like a checkbox for me to add or remove, and it saves this data if I want to use it later on, etc..

So let's add the next piece of data, which is our API key. And remember that the API key's parameter is called appid. So let's add the app ID and paste in the API key, and it's going to again add that to my parameter list.

And the final thing I'm going to do is I'm going to change the units to metric. Now I'm going to click 'Send' to send this request to OpenWeatherMap, and I get back my data.

Now notice how this data here is structured in a much better way, and the reason is because we are pretty printing the results, instead of, well, ugly printing the results.

JSON

What's happening here is, when we make our request to the OpenWeatherMap servers, the data that we get back is in something called a JSON format.

And what exactly is a JSON format?

Well, you get to find that out in the next lesson. So make sure that you've signed up for a account on OpenWeatherMap, and you've had practice using Postman as well as using your browser to authenticate yourself with the OpenWeatherMap API, and you're getting the data back in a JSON format.

Once you've done that, head over to the next lesson, and we'll find out more about the format of the data that we get back.

242 What is JSON

A JSON format stands for Javascript Object Notation.

And you might have noticed this as well.

Key-Value Pairs

The data that we get back looks remarkably like how we create Javascript objects, where we open up a set of curly braces, we have our key, then our colon, then our value, and each of these key value pairs are then separated by commas, and we can embed objects inside other objects.

Now there are a couple of differences though, say, for example, when we create an actual object in Javascript we always have a var or a let keyword in front, we have the name of the object without any quotation marks around it, because it's not a string, and inside our object, the keys are never strings, right? They're simply written as they are.

Now the reason why we use a JSON to pass data around on the Internet is because it's in a format that can be readable by a human, but it can also be easily collapsed down to take up as little space as possible.

So it's almost like if your Javascript object is a full sized wardrobe, but while you're transporting it, if you bought it from the store and you're taking it home, you probably don't want to move it as an entire wardrobe, right?

Just like the flat pack furniture at IKEA, you can collapse all of that data down into a single string. And as long as we maintain where all the curly braces and the colons and the commas are, then at a later stage, once we received that JSON as a string, we can build it back up into the original object.

Now JSON is not the only format that we can receive data from APIs.

Very frequently you'll find that various APIs, like OpenWeatherMap, will provide multiple formats for you to be able to get data from them.

So in addition to JSON, you'll often find a format called XML, which is extensible markup language, or it could also come back as simple HTML, hypertext markup language.

But JSON is currently the most favoured format, because it's much lighter weight than the other two, and also because it's very easy to turn back into a Javascript object.

Now if you want to prettify the output that we get in our browser, similar to how Postman does it, turn the raw output into a pretty version, you can go ahead and download a Chrome browser add on called JSON Viewer Awesome.

It's, again, free to download, and you can simply add it to Chrome as an extension. Once it's been added, you'll see it up here.

And now if I go back to that previous API request and I refresh it, you'll see it now gets formatted by JSON Viewer Awesome into a tree structure, or a chart structure, or the original JSON input.

Another really useful thing about JSON Viewer Awesome is, when we click on the individual pieces of data that we want, say this id, or this icon, and we hover over this green button, we can actually either copy the value, or copy the path that it takes to get to this particular value.

Notice how we can collapse some of these trees.

So this icon is nested inside an array, inside this weather piece of data, and then that is inside the original 13 item JSON object.

So if I go ahead and just paste what I copied just now, you can see that to get to that icon, we first tap into the JSON object, and then get hold of the weather object, and then get the first item, the 0 item inside the array, and finally we tap into the value of the key icon.

So once we've turned our JSON into a JSON object, this is how we would get hold of this piece of data to use in our web site.

Putting APIs into Practice

OpenWeatherMap API with JSON data format

This is what we're going to be doing in the next lesson.

We're going to be putting into practice everything that we've learned about APIs and the JSON data format, in order to use the OpenWeatherMap API to create our own web site that shows the weather for certain cities, and also allow the user to search for the weather in certain cities. This is what we'll be doing in the next lesson, so hopefully I'll see you there.

243 Making GET Requests with the Node HTTPS Module

Now that we've seen how we can use APIs to get data from various web sites, including things like quotes or jokes or live weather data, it's time to put it into practice and get these pieces of data from an API and then use it inside our own web apps.

Client Browser & Your Server & Someone Else´s Server

This is what the process will roughly look like.

> The client browser, so this is our user, is going to be typing in our web address into their browser, say Chrome, and that is going to make a request to our server, and that's going to be a GET request. So it's going to try and get the HTML, CSS and Javascript from our server.

Request API: Parameters, Paths, Key-Value Pairs

Now at this point what happens is our server should return all of those pieces of data, the HTML, CSS and Javascript, and that's in the response. But in order to be able to give them that response that includes some data from somebody else's server, we're going to have to make a request to that other server.

And it's again going to be a GET request, so that they will give us a response in the form of the data that we need.

And we're going to do this via their API, so via the menu that they provided for us to make our requests, where they've specified what are the things that we need to pass over, like what parameters, what paths, what key value pairs we have to use in order to get the response and the data that we want.

Response API: Data

So then, once we get the data, we can go back and incorporate that data into the files that we send back to our client, the user for our web site.

That's just the high level overview of what's going to happen in the code that we're going to write.

But effectively we're looking to create something very simple like this.

We should be able to display to the user the temperature of a particular location that they're interested in, and also display a weather symbol for the weather conditions there, so something that looks a bit like this. To begin, let's go ahead and create our new project, and we're going to do that on the command line using the Hyper terminal.

Command Line Module

So go ahead and open up Hyper, and cd over to your desktop. Now once you're there, you're going to create a directory using mkdir, and we're going to call it WeatherProject. Once I hit enter, you'll see my new folder created inside my desktop.

And if any of this is confusing to you at all, make sure you go back and complete the command line module, because we've gone through all of this in detail already over there. Let's continue.

New Project

WeatherProject

Now let's go ahead and cd into the WeatherProject, and once we're inside this folder, so we're now here, I'm going to create some new files using the touch command.

So I'm going to create a file called the index.html, and then I'm going to create another file called the app.js. And of course you can do this all on one line just by writing 'touch index.html', and then ' app.js'. I'm not going to hit enter because it's just going to create it all over again, but this is now what we've got.

npm init & default settings

Now the next step is we're going to initialize NPM with npm init, and I'm simply just going to hit enter for all the default settings.

package.json & install more packages (express)

So now you can see we've got our package.json created. And using NPM I'm going to install a couple of packages. And the only module we need to install right now is the Express NPM module.

So I'm just going to hit enter, and hopefully it's going to fetch everything I need from the Internet, and create my package log and download my node modules.

And now I'm ready to open up my weather project inside Atom.

Now that I've got Atom open, I'm just going to drag my weather project, the entire folder, into Atom. And now I've opened up my directory here, and I can start editing my index.html and my app.js.

So this should all be pretty familiar to you by now.

Create a new Node app with app.js

Now I'm gonna focus on my app.js file, and I want to create a new Node app.

To do that I'm going to require the Express module, so create a const called express to require my Express module, and then I'm going to create a const called app, which is going to initialize a new Express app. And then at the very bottom I'm going to do app.listen, and I'm going to be listening on port 3000, and I'm going to add my callback function, which is just going to console.log that the server is running on port 3000. Cool.

Now all I have to do is to add an app.get, so what should happen when the user tries to go to my home page, the root route.

Well, let's add our callback function, our req and res, and then inside our app.get all I'm going to do for now is just to use res.send to send over some text.

Let's just say, "Server is up and running."

> So now let's hit save and go back to our Hyper terminal, use Nodemon to run our app.js, and you can see that this is now logging from here, "Server is running on port 3000.", and we can now go to localhost:3000 and see that our server is up and running.

So it's all connected.

We're now ready to get started.

Build With Muscle Memory

Now I know that some people like to save all of this boilerplate somewhere, and then just copy and paste it whenever they create a new project, but I think when you first get started working with it, it's quite nice building up a little bit of muscle memory and just typing it out every time.

And this way when something does change, you're able to react to it, rather than having to go and debug it because you've copied it from somewhere. Instead of sending, "Server is up and running.",

GET Weather Data

I'd really prefer to send the current weather data. In order to do that, I have to somehow make a get request to the OpenWeatherMap's server, and be able to fetch the data back as a JSON, and parse it so I get the relevant piece of information.

How do we do that? If we didn't know how to do this, then we'd probably ask our good friend Google, right?

Make get requests to external server with node.

The first result we get back on Stack Overflow points to a package called request, and, in old versions of this course, I used to point students toward this module, because it's really nice and it has a really simple implementation.

But, as of January 2020, the request module is now deprecated, so it's basically being retired. So if you've used request before in one of your previous projects, then it will still work, but for all new students, I recommend to use a different module.

HTTP - the Standard Library

And if we take a look at this blog post by Twilio, which came up actually in fact as the first result for our search, they're telling us that there are five ways of doing this, and the first way is the native way, using the Standard Node Library, something called the HTTP module. And in fact they're actually using the HTTPS module, which is the secure version. But in addition you could also use the Request module, Axios, SuperAgent, or Got.

Now all of these last four are external NPM packages, but I want to show you how you can do it natively.

So we're going to be using the native HTTPS Node module.

So if we search for it inside our Node documentation, you can see that somewhere down here, there is the option to form a get request. And all we have to do is to get hold of the HTTPS module, to call the get method on it, and then, once we get our results back, we can either log the status code, or simply just get hold of the data.

This is what we're going to use in our code.

So right at the top here, I'm going to create a new constant called https, and then I'm going to require this HTTPS module. And we don't actually have to install this using Node, because it's one of the native Node modules, which is already bundled within our Node project, so we can simply just go ahead and use it. Inside my app.get, before I send the result back to my client,

I'm going to use my HTTPS module and call the get method. Now the get method is going to need a URL.

If we go into Postman, where we've got our already prestructured URL, which includes all the parameters that we've added here.

Make sure that you've checked that it actually works, that you actually get back some data, and everything looks like it's working.

Then we can copy this entire formatted URL, and then paste it into our get method right here.

Now the thing to remember about this URL, though, is it needs to have the strict formatting of the https:// and then the rest of the URL, because even though we could take this URL, paste it into our browser, and our browser will automatically add all the necessary https etc., it doesn't work like that when we use our Node module. We have to have this first part there.

Now you can either just paste it into your browser and then select it and copy it, and now when you add this string you'll see that it's automatically included that first part, or you could simply just have typed this as well.

It doesn't really matter. Now because this URL is so long, and we can't even see the beginning and end of our get method, what I'm going to do is take this string out of the method, and use a constant to hold it instead.

So we'll just call that url, and set it equal to the entire string.

Now I can simply pass the URL in as the first parameter, and then I can create my callback function, and I can see everything all in one line, which is brilliant.

Now this callback function is going to give us back a response.

So normally, when you see it written, say in the documentation, you'll see people shorten it to res, but because we've already got req and res here, I don't want to have another res, because it's kind of confusing, so I'm going to call this the full name, the response, and when we get back our response, all that I'm going to do is just log this response to see whether if this entire process of making a HTTP get request over the Internet to this URL, where we're supposed to fetch some data, whether it actually works, and whether we actually get something back.

So now let's hit save, and let's refresh our localhost:3000.

Now if we go to our Hyper terminal, you can see that we've gotten back a whole bunch of data, including the method of request that we made, the path that our requests went to, and most importantly the status code that we got back from the external OpenWeatherMap server.

And it says 200, which basically means a OK. So now that we've got everything working, in the next lesson we're going to dig into our response a little bit more, and we're going to learn more about the status codes and how to get the actual data out of all of this mess.

So for all of that and more, I'll see you on the next lesson.

244 How to Parse JSON

Now in the last lesson, we saw how we could use the native Node HTTPS module to perform a GET request across the Internet using the HTTPS protocol. Now we passed in a URL, which is this one. And once we got a response back, we simply just logged the response. Now, if we wanted to, we can actually be more specific.

We can actually log the status code that was associated with the response. So if I go back to my browser and hit enter to refresh, I'll be able to trigger my code again, and we'll be able to scroll to the bottom and see the response code that was printed.

So we got 200.

Now, if you're curious what 200 means, then you can head over to the MDN web docs and take a look at all the response codes that you could possibly get back from your HTTP requests.

> https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

HTTP Response Status Codes

Now, they kind of fall into broad categories, the one hundreds are sort of informational, the two hundreds are usually a really good sign, the three hundreds refer to redirect web sites, which normally you don't encounter, the fours are interesting because they usually mean that the client, or the user, who's interacting with your web site made some sort of error, and finally the fives are an error on the server. But you can actually dig into it.

So, for example, 200 specifically means just a OK, the request is successful. And because we were making a get request, then it means the resource has been fetched and it's transmitted to the message body. That is the best outcome that we could get when we're making an HTTP request to an external server.

Now, however, we could be getting one of these other ones. Now the one that you've often seen on the Internet, if you've spent any time on the Internet, is the 404, and this corresponds to the server that you made the request to cannot find the requested resource, so basically it's a resource not found.

Now, we can trigger this quite easily by simply changing one of these paths. So instead of calling it weather, let's say we made a typo and we wrote weatheer instead. Now if I hit save, go back, refresh my web site, go to my Hyper terminal, you can see I'm getting 404.

The OpenWeatherMap server is basically telling me that this resource that you're looking for at this path, weatheer, does not exist. So let's restore our path.

Now, another type of error you'll often get is when you have to authenticate yourselves with an API provider. Let's say that you had made an error in the app ID. So let's just botch our app ID by adding an extra character in front of it. Now I'm going to hit save, refresh my web site, now go back to my Hyper terminal, and you'll see I'm getting a 401.

And, if we look it up, a 401 refers to an unauthorized HTTP request. So this is because we did not authenticate ourselves properly with the right app ID. So let's delete that extra character, go back and refresh, and we now get our 200. This is what we're looking for most of the time.

But very often, when we're testing APIs, when we're working with external servers, we're very likely to get different messages back. So when you do, have a look at the documentation on Mozilla, and match it with the corresponding code to see what's happened.

There's some pretty funny ones in here if you actually read it, including the error code 401, I'm a teapot. The server refuses to attempt to brew coffee with the teapot. And this is just a little remnant leftover from some past April Fools joke, although you might one day encounter a server that actually does give you that code.

> httpstatusdogs.com

Now, if you're interested in a bit of light hearted entertainment, there's also the HTTP status dogs, which basically uses dogs from the Internet to represent each of these codes, like 200, or 404, Not Found, 401, Unauthorized, etc., and it's pretty hilarious to look through.

Checking the status code

Tap into the response we get back to search for data

Now, in addition to checking the status code, we can also tap into the response that we get back, and call a method called on, and search through it for some data. Now this will correspond to the actual message body that we got back, that OpenWeatherMap has actually sent us. So let's try to implement this method. Because our response is spelt out fully, we'll have to tap in to response. and then we're calling the on method, and then inside the on method, we can tap into a specific moment, say when we receive some data in the response.

So let's first call the response on when we receive some data. And I'm going to create a callback function that's going to contain the data that we get, and then I'm just going to log the data. Now let's hit save and go over to our web site, refresh, and check out the response in our terminal. You can see that we're getting the response code 200, which means everything is working, but the data that we're printing out looks a little bit funny. It seems to be all jumbled up and we can't really make much sense of it. So what exactly is this?

Well, this is actually hexadecimal code.

And if we copy it and put it into a hexadecimal converter, we can convert it to text, and you can see that the text we're getting back is pretty much the beginning of the JSON that we're getting back from OpenWeatherMap that we saw earlier here.

> https://cryptii.com/pipes/hex-to-text

And this is just simply a different way of representing that data.

Now what would be far more useful for us, though, is to actually get a Javascript object, and we can do that by converting the data into a Javascript object. And to do that we would need to write JSON. parse, and this will turn a JSON in some sort of string format, say the hexadecimal, or binary, or text, and then turn it into an actual Javascript object.

So inside this method I'm going to pass our data, and now let's store this inside a constant called weatherData, and let's go ahead and console.log that. And now you can see that we're able to print an entire Javascript object, so there's no strings in the keys, and everything is organized as you would any other Javascript object.

We've basically taken our IKEA flat pack, and we've unwrapped it, and followed the instructions to recreate the 3D wardrobe, or, in our case, the 3D Javascript object using JSON.parse.

Now we can also go the other direction if we wanted to.

Let's say that we had an object called, I don't know, let's just call it object, and let's give it a name, let's put my name in it, and then maybe, and then taking this object we can use the other method from JSON, which is stringify. And when we stringify an object, it basically does the opposite. It will go ahead and turn a Javascript object into a string.

So now if I print it, you can see that it's completely flat packed like this, right, taking up the minimum amount of space necessary. All our keys have become strings, but keeping all of the syntax that's required to be able to bring it back to life later on.

And this is effectively going the other direction, taking the 3D wardrobe and packing it down to a single string using that method, JSON.stringify.

Now let me delete all of that, and I want to be able to use my weatherData to pull out specific pieces of information from it, like the temperature, or the description, or an icon that I can use to display in my web site.

So how do I do this? How do I get this temperature specifically printed out?

Well, because we've got basically one huge nested object, we can tap into the object, look for the main key, and then look for the temp key.

So the code for that would look like this.

Let's create a const called temp, and then let's tap into our weatherData. The first level that we go down is this indentation, and we get to a key called main, so weatherData.main. And then once we're inside the main object, then our value lives inside a key called temp, so then it's .temp.

So now if I log my temp, we should be getting 5.84. So it's 5.84 degrees here in London at the moment.

Now, while that wasn't very difficult, that also wasn't a very complex JSON.

If we end up with a really long JSON, sometimes it might get confusing digging through the levels and seeing how to get to each of these values.

Now I want to show you a neat trick.

So earlier on I showed you that you can install the JSON Viewer Awesome as a free Chrome browser extension. Once you've done that, if you head over to the API that we're calling over here, so putting this particular URL into the browser, then we get back the same data here. And if we want to access a particular piece of data, all you have to do is click on it, hover over this icon, and then click 'Copy path'.

Now this path, if I paste it, is basically the way to navigate to the piece of information that you're interested based on the object that you have.

So the object we've got back is weatherData. So we write weatherData. and then paste in the path, then we get hold of the temperature. Now if I get hold of the feels_like temperature, then I can also do exactly the same thing. It becomes main. feels_like. This is a really neat tool for us to be able to work with JSON, and I highly recommend you install it if you haven't got it already.

Now going back to our JSON, the next piece of data I want is the description of the weather, so you can either use the method of looking through it and see how you can get this description, or you can use JSON Awesome and get the path and do what we did just now. But I want you to try and print out the weather description, so pause the video and give that a go.

All right. Let's create a new constant, and I'm going to call it weatherDescription, and it's obviously going to come from our weatherData JSON object. And then I'm just going to use the easy way, which is write a dot, and then take my description, copy the path, and then paste it over here. So effectively what we're doing here is we're taking this entire object, we're digging into the weather object using that key, and then the weather object is actually an array. It's an array with only one item.

So we get hold of that item using the 0, and then we get hold of the description by tapping into that key. And now, if we console.log this weatherDescription, then we get "overcast clouds" being printed.

Overview of Lesson

So in this lesson, we looked at how to get hold of the data from the response, and then to parse the JSON data that we get back into an actual Javascript object, and then we saw how we can dig through our Javascript object to get these specific pieces of data that we're interested in.

Next Lesson

Now in the next lesson, we're going to be taking these pieces of data, and using Express and Node to put the data into an actual HTTP web site that we can display to our user.

So head over to the next lesson to continue building out our site.

245 Use Express to Render a Website with live API Data

Fetching data

From external servers, to JSON, and using the logged data

So in previous lessons, we saw how we could fetch some data from an external server, and then get the data in the form of a JSON, and then parse it into an actual Javascript object, which we can dig through to get the pieces of data that we want.

So now that we're able to log the data that we're interested in, the next step is putting it onto our web site.

Revision from our Node and Express modules

Response

This will be a little bit of revision from our Node and Express modules.

In order to pass the data back inside our app.get, we have to tap in to our response. This is the response that our server is going to send to the client's browser. Now, in this case, the response could just be the data that we've got, right? So we could say res, which refers to this response. Now you see why I didn't want to name this res as well, right, because it would be mightily confusing which one I'm using.

So res refers to our app.get response, and the response we're going to give the browser is going to be a send method. So we're going to send the data that we've got.

"The temperature in London is ", and then we'll tag on the temp at the end here, and then maybe we'll even add a degrees Celsius.

Setting headers

Now remember that, if we run our code as it is right now, it's going to crash, because we can only have one res.send in any given one of these app methods. So we can't send here and then send here, because once we call send, then that's basically the end. There's no more sense going forwards. And if you do try to do that, you'll end up with an error that says something like this, "Cannot set headers after they're sent to the client. Error HTTP headers sent." Something like this. Instead we're going to delete this previous send, hit save, and now, when we refresh our web site, you can see we're actually getting the data, and we're displaying it in the web site.

The temperature in London is 6.27 degrees Celsius.

Send data as h1

Here's a question. How can we send this as a h1 instead of just a piece of text like this? Pause the video and see if you remember how to change it. All right.

So we can actually write HTML in our res.send or our res.write. So all we have to do is just to wrap the final string inside a h1 tag, so the h1 beginning tag and the h1 end tag. Now if we go back and we hit refresh, then we actually get it displayed and formatted in the way that we want.

Challenge

Weather Description

Here's your challenge. See if you can remember how we can also send the weather description, so something like, "The weather is currently ", plus the weather description. But remember that we can only have one res.send, because that's the end, that's the final thing that happens. So see if you remember how you can do this. Pause the video and try to complete this challenge. All right.

Res.Write

So we know that we can only have one res.send, but we can have multiple res.write, right?

So we're going to wrap this inside a res.write, and we're going to then wrap this inside another res.write, and finally we're going to call res.send once we've written all the things that we want to send. And our weather description is probably going to go in as maybe a paragraph, or it could be a h3, depending on what it is that you want.

So now when we refresh you can see "The weather is currently overcast clouds. The temperature in London is 6.25 degrees", and we've now been able to send multiple lines of HTML using our res. write in combination with our res.send.

Display Images with Condition Codes and Icons

The final thing that I want to display in here is an image.

And if we go back to the API for OpenWeatherMap, you can see that they actually have a bunch of condition codes in their API. The weather condition codes are these ones here, the ID of the weather, and they correspond to different weather conditions, whether if it's a thunderstorm, or drizzle, or rain, or snow, etc.

But we also get back an icon name, and that icon name corresponds to the weather condition as an image, and OpenWeatherMap actually provides all of the images for all of the icons that they send you. All you have to do is just fetch it like so.

So say if the icon was 10d, then you would insert this 10d into this URL to get the image, and leave everything else as it is. And the image would look a bit like this, which we can use display into our web site.

So let's see if you can figure out how to get hold of the icon from our weather data. Pause the video and try to complete that challenge. All right.

Solution

So it's pretty simple, because we already know how to get the description. Then the icon is as easy as taking all of this, and then just getting hold of the icon at the very end. Now the alternative is, of course, you could have also just copied the path here and pasted the path in as well. But once we've gotten the name of the icon, we have to create the image URL, and this comes from our API here.

Refactoring elements

And so I'm just going to copy this entire URL, and paste it in here as a string. And then, instead of using this part, which is the icon ID, I'm going to add it in by using string concatenation. And of course I'm going to insert the icon that came from here. Now that we've got the image URL, we're going to write another res.write, and this one is going to contain a image element. So pause the video and see if you can send back to the browser an image element that will display this image URL. All right.

So we're going to again be using res.write, and in this case though, we've got an image element that we're going to create. Now images are self closing tags, so we don't need to close off the tag, but what we do need is a source, and the source is going to go right here. So let's pass in our image URL and hit save. And now when we refresh our web site, you can see the corresponding image is now picked up.

Now we've got our live weather condition, temperature, and even the condition image being displayed in our web site, and that's all thanks to us being able to get live data using an API, making a HTTP get request to get the data as a JSON format, parsing it and fetching the specific items that we want, and then sending it back to the browser using the HTML that we want to write.

Next Lesson

POST Request

Now in the next lesson, I want to show you how we can make a post request from the web site, so that the user could log onto our web site, type in a particular city that they're interested in, and then we can give them the actual weather for that city.

So for all of that and more, I'll see you on the next lesson.

246 Using Body Parser to Parse POST Requests to the Server

Previous Lessons

Fetching Data From API and Placing Into Web Site

In the last lessons, we've done the difficult part, which is fetching live data from the OpenWeatherMap API, and then getting that data formatted and placed onto our web site, like so. But it's a little bit boring if it just displays the weather for London, right? That's not much of a web site.

Get The User To Type In Data

What if we could get the user to type in the city that they're interested in, and then we simply replace this query with whatever city they want, so like, maybe, Paris? So we could say, well, Paris currently has scattered clouds and it's about the same temperature. But of course we'll have to change this, and we'll have to add some sort of input to let the user be able to do this.

Break down URL parts

Instead of having an entire long string here, I'm going to break down each of the parts of our URL. So I'm going to create a const called query, and I'm going to set that to equal London, at least for now, and then I'm going to delete this part here and replace it with the query. Now additionally, I'm also going to take out my app ID, and I'm going to save that as a separate const as well, and I'll call that apiKey. And we'll save it under these two names. And then I can replace that with my apiKey. And finally I'm just going to take out the units, and add it on at the end as well.

Change Data to Update Query

So now all we have to do to change the data that we get back is to update this query. So how can we do that? How do we update it?

Create an HTML Template

Inputs

Well, if we go into our index.html, which currently is completely blank, we can start off by creating a HTML template, and our title will be called our Weather App, and inside the body all I'm going to do is add a text input. So this is going to be a input of type text, and it's going to have a name which is called the cityName, and it's not going to have any value at the moment.

Labels

Now in addition to an input, I also want to have a label, and it's the label for this particular input. So let's give it an id, and let's call the id the cityInput, and let's put cityInput here as well. Now our label is going to say "City Name:". Hit save.

And now we want to be able to render this index.html when the user calls app.get at the root route. So instead of all of this, I'm going to cut it out and I'm going to paste it down here, and I'm just going to comment out this code for now.

Show Label and Input

Now our web site is again quite simple, and inside our app.get I'm going to try and send this file, index.html, over to the browser. I'm going to call res.sendFile, and then I'm going to get hold of the directory name, and I'm going to add "/index.html". And hopefully, when I refresh my web site now, it's going to display that web site. It's going to show my label and my input.

Send Data: "Go" Button

So now that I'm free to type whatever it is I want in here, I need some way of sending that data, right? So I need a button in here as well.

So let's add a button, and the type of button is going to be a submit button. Now this button doesn't really need a name, but what it does need is some text. So we'll just call that 'Go'.

Create a POST Request When The User Sends Data

And if I wrap all of this inside a form, then I'll be able to create a post request when the user hits this button. And that post request is ready here in the method, but the action is actually going to go to our root route. Our form is going to make a post request to this route on our server, and we would have to catch it using app.post, then specifying the root route, which is the one that we said we would hit up over in our form, and then create our function with our req and res. Inside here we're just going to log, "Post received."

So now let's refresh our web site. We can see our label, our input, and our submit button. Let me just write the word 'London' in here, click go, and then go into our Hyper terminal, and we can see "Post request received." So it's hit up our post route, and this is why we're getting that logged. But in order for us to actually get the text that the user typed into the input, we have to do a little bit more.

Install Body Parser

We have to actually install another package that we've been using previously.

So I'm using Control C to exit out of Nodemon, so that I see my prompt again, and I'm going to use NPM to install Body Parser, body-parser. And body-parser is the package that's going to allow me to look through the body of the post request and fetch the data based on the name of my input, which is called cityName.

Create A Constant Called bodyParser

So now let's go ahead and create a constant called bodyParser, which is going to require the body-parser module, and then let's get our app to use bodyParser, and we're going to set the urlencoded to use the extended as true setting. From previous modules, we've used this a few times, and this is just the necessary code for us to be able to start parsing through the body of the post request.

request.body

But once we've done all of that, instead of console logging, "Post request received.", we can go one step better than that. We can say request.body. the name of the input, which is called cityName, and we should now be able to console.log the text that went into that input.

Check Input Results In Terminal

So let's delete the previous console. log, hit save, start up our server again with 'nodemon app.js', and now go to refresh our web site, type in 'London', hit 'Go', and check our Hyper terminal.

Receiving Data From POST Request

And you can see we're now receiving our data from our post request. Now that we've parsed through the body of the request, you can see now how we can start using that as our query, right? Let's go ahead and take all of this, and paste it into our post request, and then uncomment it using Command /, and instead of console logging the request.body .cityName, we're going to be using that, of course, as the query.

Nothing Hard-Coded

So instead of having our hardcoded 'London', we're going to write request.body.cityName, and we can delete our console.log now. So now if we hit save, go back to our web site, refresh, let's type in a different one, let's say what's the weather like in Paris. And now it sends us the weather, but of course we need to modify this text as well.

Change Words For Queries

So down here where it says 'London', let's get rid of that and let's instead add the query in here. So let's go back and let's check a different place.

Bali Weather

Let's see what is the weather like in Bali. Here you go. The temperature in Bali is 30 degrees, pretty nice, but there's currently thunderstorms. And that's what our image shows as well.

So now we're able to get dynamic data based on what the user typed into the input, catch that data in our app.post, and then use that query to structure our URL and get the data for that particular location.

Overview

Now we've seen all aspects of our API in use, including authentication, including paths, including queries, and we've been able to parse the JSON data we get back and send it over to the browser using our Express and Node modules.

Next Lesson

Email Collection Web Site

Now in the next lesson we're going to be taking this one step further, and we're going to be building our final project, which is our newsletter email collection web site, which is basically going to be a landing page that you can use to collect user emails.

So for all of that and more, I'll see you on the next lesson.

247 The Mailchimp API - What You'll Make

New Project

Hi guys. Welcome back. This is Angela from The App Brewery, and it's a new day, which means that it's time for a new project. So in the coming lessons, we're going to be building something really exciting, at least I'm really excited to build it with you.

So far none of our web apps that we have built have been deployed, and none of them have really been all that exciting to deploy, to be honest. But everything is going to change for the next project.

Newsletter App

So the web app that we're going to be building is a newsletter app. So I'm probably signed up to far too many newsletters, but some of them are real gems, and I really enjoy reading them.

So if you are somebody who wants to set up your own newsletter, where you want to email people who are interested in hearing from you, well, then you need a sign up page, right?

So this is what we're going to be building.

The Power of the Backend

It's a single page web site that looks pretty nice, and on the front end it looks incredibly simple, but on the backend it's got some powerful functionality that will allow you to sign people up to your mailing list. So let's give it a go.

Let's try, say, let's say that Jack Bauer wanted to sign up to my mailing list. Now once I click sign up, if it was successful then I get taken to the success page, and if it wasn't then I will get a failure page.

So now, finally, we were successful, and the data that we entered into that collection box doesn't just disappear. No.

It gets sent to our list on MailChimp, and all of that data gets added into our list, and our new subscriber gets added.

So that means now you can go ahead and send your newsletters, or your email campaigns, to your subscribers, and this list will build up as more and more people subscribe to your newsletter.

Deployed Website (not local host)

Heroku + MailChimp

So the special part of this web site is that, if you look at up here, that is not a localhost anymore. This is an actual HTTPS web site, and you can include this URL in your socials, on Facebook, Instagram.

And anybody can access this, because it's now on the World Wide Web. So this is going to be a really cool and really useful web site by the time that you've finished making it, and it's all down to this integration with MailChimp.

And we're going to explore how to work with their APIs, and how to post data and communicate with their server in order to send them pieces of data that we want them to hold on to.

So I hope you're excited. Once you're ready, let's head over to the next lesson.

248 Setting Up the Sign Up Page

Node and Express Page Challenge

Set up the project

All right guys. So you should be pretty familiar to setting up new projects using Node and Express by now. So I want to set this to you as a challenge, and right now on the screen you should see a checklist of a number of things that you need to do in order to set up this project. And I want you to pause the video and try to do all of those things so that we get to a point where we're ready to go and we can start coding up our sign up form. So good luck and I'll see you soon. All right. So I hope that went all right.

We're going to be repeating these steps every single time we create a new project, and it's pretty simple once you get used to it. So first things first. We're going to change directory into a location where we want to create our new project, and we're going to call it Newsletter-Signup.

And now that we've created this new directory, let's cd over to it, and inside here we're going to create our new files. And the file that we're going to write our server side code in is going to be called app.js.

Now in a lot of Node.js, and especially Node.js projects that use Express, you'll see that their main server file is called app.js.

Now it's actually completely up to you what you want to call it, but this is something that you will see out in the wild as well. If you don't want to create each of these files individually by writing touch this file, touch that file, you can actually chain them together just by using touch once, and then you can add all of the names, all of the files that you want to create in one go.

So, for example, we need to create signup.html, as well as success.html, and also failure.html. By having all three files with a single space between them, when I hit enter, you can see that all three get created simultaneously.

So the command line is super powerful and it is here to be your friend.

So the next thing to do is to create our NPM to initialize NPM. And I'm just going to keep to these standard options just by hitting enter. And finally I'm going to hit enter to create our package.json.

So now that we have that, we can go ahead and install our NPM modules. And again, just as I did with the touch command, I'm going to install a number of modules simultaneously, so add all of their names, and then I'll include a space in between, so that we can install all of them at the same time.

So the ones I need are body-parser, express and request. So now let's hit enter, and we'll let that download in the background. All right.

So now that it's done, let's open up our project in Atom, and let's head into our app.js and require those modules that we just installed.

So the first one was express, next was body-parser, and finally we've got our request package. And let me just add in that comment, so that our JSHint is not going crazy on us. All right.

Now that we've done all of that, the last thing to do is to setup our server to listen on port 3000.

So first things first, we need to create our app constant, and this is equal to a new instance of Express. And now we can use that app to listen on port 3000. And once that port is set up and ready to go, we're going to log that our server is running on port 3000. All right. Cool. That's it.

That's all you need to do to set up a brand new project. And as you go along creating new projects, this process will very soon become second nature.

HTML page

Sign up form

Now that we've done all of this, the next thing that we need to do is we need to create a HTML page that has a sign up form on it. So where can we get one of those in a hurry? Well, it's time to start piecing together some of the different things that we learnt about in this course.

If you remember, in the Bootstrap module, we spoke about how Bootstrap can cut down on development time and make our web site look really nice. So let's head over to getbootstrap.com and go over to their Examples section. So here they've got a whole bunch of examples that allow you to quickly get a project started, and that's exactly what we're going to do. So if you scroll down, you'll see that there are some complex forms, which is not what we want, but there's also some simple forms, and this is exactly what we're going to use.

So right click on this web site, and we're going to view its page source, and then we're going to select all of this, and then paste it inside our signup.html. So now, if you right click on this and copy full path, then you can see that our page looks nothing like what they had over here. Now why is that?

Well, the first thing you notice is that this doesn't look very bootstrappy at all. The fonts are not bootstrapped, the buttons don't look bootstrappy. So we haven't actually got Bootstrap enabled in our web site. And why is that?

Well, if you take a look at this section where it says 'Bootstrap core CSS', well, they've actually got it included as a local file, and that file we don't currently have. So let's go onto the Internet and grab the Bootstrap CDN so that we can also incorporate Bootstrap into our web site.

I'm just going to copy this entire link, and I'm going to replace this line with that. So now let's go ahead and refresh our web site, and you can see that we've got a lot of the Bootstrap elements that have been enabled. The fonts are now Sans Serif, and it's looking a little bit nicer. The only problem is that they still don't look anything like each other, right? And the reason is because, if we have a look at the source, in addition to the Bootstrap core CSS, they've also got a custom style sheet for this template. And you can find this template by following this link. Aha. So they've got some custom CSS here. Well, let's go ahead and rip off that as well, so that we can put it into our web site.

Now I use these terms like rip off and steal, but actually it's fine. Bootstrap allows you to do this and it's set all of this up so that we can speed up our development time and make it easier for us to create beautiful web sites. So inside our newsletter sign up, I'm going to create a new file called styles.css, and I'm going to paste all of that CSS into here. And, as usual, we're going to change our link ref to refer to that CSS, which was called styles.css.

So now, when we refresh our web site, then you can see that we're getting pretty close to the desired look and feel. Now all we have to do is just to modify the words and the inputs and the buttons to fit our project. So inside our signup.html, we're going to change our title from 'Signin Template for Bootstrap', which looks like we obviously took it from somewhere, to our Newsletter Signup. Next is this image, which is the Bootstrap brand logo, which looks like this, and instead we're going to include our own custom image.

So I'm going to create a images folder, and inside this images folder I'm going to drop in a image that I have, which is called lablogo.png. Now feel free to add in whichever image you would like for your newsletter web site because, after all, this is about your web site. So now I can replace this source from this http address to "images/lablogo.png". This is a image called lablogo.png inside a folder called images, and that corresponds to what we've got over here. So let's hit save and let's check to make sure that it appears. All right. Looking pretty good.

Now let's change some of these bits of text. Instead of 'Please sign in', let's say 'Signup to My Newsletter!'.

And instead of having an email address and a password, what I actually want are three input fields: the first name, their last name, and their e-mail address, because we're not actually signing anybody in, we're signing them up. So I'm going to add another input, and I'm going to delete these labels, because currently they're only for screen readers. And I'm also going to delete our checkbox that says 'Remember Me', which we don't need.

So let's go ahead and add a new input. So we have three of them now, and the first one, the type is going to be just a simple text input, and the place holder is going to be "First Name". The second one is going to be, again, type of text, and the place holder is going to be "Last Name". Now we don't need an id for our inputs, so I'm going to delete that. And finally, the last one is going to be type email, and the place holder is going to say "Email". All three of these are required, and the first one is automatically focused when we load up our web site, so that means it's got that blue highlight around it. And, of course instead of saying 'Sign in', we should say 'Sign Up', or 'Sign Me Up'. And the copyright text down here I'm just going to change to say 'The App Brewery', or whichever company, or your name, if you wish.

So now that we're happy with our text and our images, the next thing to fix are these input boxes. You can see that in their input boxes, they all look very together, right? And that's because the corner radius is only applied to the top left and right for the first input, and then the bottom left and right gets a rounded corner on the second input. And we can see that when we go into the styles.css, you can see that they're targeting the input that has

type email and the input that has type password to apply these things to. Now we no longer have an input with a type of password, so I'm going to change this one to .top, this one to .bottom, and I'm also going to add another class called .middle. So now we can go over and apply those classes to our inputs. So this one is going to have a class of top and this one is going to be middle, and this one is going to be bottom.

So now, if we take a look at our rendered newsletter page, then you can see that we've got the rounded corners for the top and the bottom looking pretty nice. Now it's just the middle one. It doesn't actually need any rounding of the corners, so we can go ahead and add the border radius to zero. So that means that it no longer has curved edges. And the final thing that you can fix if you want is this last bit looks a little bit heavier in terms of its shadow, and that's because we need a margin-bottom of -1 pixels just to pull it up a little bit.

And that is now looking perfect.

So, as you can see, at the moment I'm accessing this web site as a static page. Now that's not how we do things anymore when we have a server, but, in order for that to work, you need to set up the get method for our home route, and you also need to make sure there's two S's in express.

Challenge

Set up local host

So, as a challenge, I want you to set up the get route to our sign up page so that we can test it on the browser at localhost:3000. Pause the video and remind yourself of what we did in previous lessons in order to get our app up and running using the server. OK. So this shouldn't be too difficult as we've done it a few times already.

The first thing to do is we're going to say app.get and then we're going to specify the route. And we're going to use the home route in this case, so that when people go to our home page, they first see our newsletter sign up. Then we're going to add our callback function with req and res, and inside here we can specify that our response is to send the file that is at the location of our directory name of the current file plus the string "/signup.html".

So now when we request the home route from our server, then it should deliver the file at that directory name, which is something like 'Desktop/Newsletter-Signup/ signup.html'. Let's see if that worked by running our server with Nodemon, and we're going to use app.js to spin up our server. Server is running on port 3000.

So let's head over to localhost:3000, and you can see that something is not quite right. At the moment our changes to the text have been replicated, but our image is gone, and our CSS seems to be gone.

So why is that?

Well, if you take a look inside this web site, we get the Bootstrap style sheet from a remote location, but our custom style sheet is local. So this is basically a static page in our local file system that we're trying to pull up. And it's exactly the same story with our images. In order for our server to serve up static files such as CSS and images, then we need to use a special function of Express, and that's something called static.

So we need to say app.use(express.static), and inside the parentheses we're going to add the name of a folder that we're going to keep as our static folder, and I'm going to call it public.

So now, inside my Newsletter-Signup, I'm going to create a new folder called public, and inside this folder I'm going to add a new folder called css, and I'm going to place my styles.css inside there.

Now I'm also going to drag my images so that it's also inside our public folder. And now I have all of my static files in one place under a particular folder name. And by using app.use(express.static), providing the path of our static files, then we should be able to refer to these static files by a relative URL, and that is relative to the public folder.

So imagining that we're currently inside the public folder, in order to get to our style sheet, we have to go to the css/styles.css. Now for images, it's exactly the same, inside the images folder, to lablogo.png. So let's hit save and let's refresh, and you can see that all of those static files, the CSS and the images, are now able to be rendered all because of this one line of code that specifies a static folder where we have all of those files.

Challenge

Program the POST Route

Now that we've managed to serve up our web site using our server, the next challenge for you is to program the post route. And we're going to use Body Parser to grab the data from the sign up form, and we're going to console.log it from our server. So pause the video and see if you can figure out how to do that. All right.

So, in order to create the post route, we're going to say app.post, and we're going to target, again, the home route, and then it's time for our callback, and inside the callback we're going to try and log some of those things that the user entered into the input. In order to do that, remember we need to use body-parser. And we've already required it, but we need to tell our app to use it. So you have to say app.use(bodyParser), and, more specifically, the urlencoded method of bodyParser. And we're going to set extended to true as usual.

So when you start a new project, it is pretty much inevitable that you will use Express and Body Parser. So some people find that it's easier to just keep a copy of all of this somewhere, and just paste it in for every single new project.

Now I prefer to write it all out, because on one hand that will give you more practice at coding, and secondly sometimes things change, and if you don't write it out, you're not always aware of what you've put in there, so that can lead to unexpected crashes. All right.

So now that we've set up Body Parser to be used by our app, the next thing to do is to pull up the values of those things inside our form.

So let's create a variable called firstName, and we're going to set that equal to the request.body. name of the input. So for the first input, we're going to give it a name of fName, and for our second input we're going to call it lName, and for our last one, we're just going to call it email. Now that we've got all of those names set up, we can pull up the values by referring to it inside Body Parser. So request.body.fName should be equal to our first name, and our last name should be equal to request.body.lName. And finally the email should be request.body.email.

So let's try logging some of these values: firstName, lastName, and email. Let's hit save and refresh, and let's enter some values in here. So the form got submitted, but we didn't trigger any log statements.

What's going on here? Well, if you take a look inside our signup.html, you can see the form currently doesn't have any actions specified.

So in order to fire a post request to happen, we first have to specify the method as post, and we also have to specify the action, that is the location that we're going to send our post request to, which is going to be the home route.

And this ensures that when we click the Submit button, the information in here gets posted to this location, so that our server can then pick it up in this route that we have preplanned.

So now, if we try again, you can see that we get all of that logged in our console. So our Body Parser code is working, and our sign up form is able to submit the data in the inputs to our home route using a post request.

So now we've done all the prep work of setting up our sign up page and getting the data that's inside the inputs.

Now in the next lesson, we're going to set our web page up with the MailChimp API to start sending this data over to their servers. For all of that and more, I'll see you on the next lesson.

249 Posting Data to Mailchimp's Servers via their API

All right guys. Now in the last lesson we've set up our requests and responses to the get requests and the post requests on our home route. And by doing this we're able to load up our sign up page with bootstrap enabled and we're able to send this data over to our servers using body parser.

So now it's time to start incorporating our API.

So if we google for MailChimp API then we get taken to developer dot MailChimp dot com. And this is the documentation for MailChimp latest API version 3.0. So we're going to go ahead and click get started with the MailChimp API.

Authentication

Now as they say here in order to use the API we need to get a API key to authenticate ourselves with their servers because after all you don't want some random person to start adding subscribers to your mailing list right. And you definitely don't want somebody to just delete your entire mailing list by making post requests that are not authorized.

So let's go ahead and setup our API key.

If you don't have a mailchimp account then this is a good time to set one up to sign up for a mailchimp account. Just head over to MailChimp dot com and then click on sign up for free. And here you can create a new account with MailChimp and it's completely free and it doesn't require any credit card details. The process is pretty self-explanatory but you want to choose the free plan and tell them that you don't have an existing list of email subscribers. And finally you can click on Not right now for finding the marketing path.

Now once you're done go ahead and log in and then we're going to get your API key so to do this click on your account name go to accounts extras and click on API keys.

Now once you're here you can scroll down and click on create an API key and that will generate a new API key for you. So you want to go ahead and select your API key from here and then paste it at the very bottom of your app. And we're going to combat it out for now or come back to it a little bit later on.

Instruction Manual

Add Subscribers To Our List

So now that we've gone our API key then already to interact with MailChimp servers through their API. If you head back to MailChimp dot com slash developer and then scroll down to click on their API reference then we can use this as our instruction manual to figure out how to add subscribers to our list.

The first thing to do is to head over to the list section and there's a number of things that you can do here. You can create you can read. You can edit and you can delete.

Now we're interested in the creation parts because we want to subscribe some list members. So I'm going to click on here and I end up at this post drought which allows me to batch subscribe or unsubscribe list members. Now the process is exactly the same. If you want to batch subscribe. Lots of people or simply subscribe a single person all you need is a list I.D. and you can provide the members that you want to subscribe in the body of your request. So how do we do all of that.

Well first things first let's get our list I.D..

If you go back to MailChimp dot com and presuming that you already logged in you can simply go to audience and manage audience settings and scroll down to get your unique I.D.. This is your audience I.D. or also known as list I.D. and that is going to help MailChimp identify the list that you want to put your subscribers into. So let's go ahead and copy that and paste it at the bottom of our app G.S. and we're going to comment it out as well.

Now that we've got our API key and our list I.D. then we're ready to move on to the next step which is to start sending our request to MailChimp.

Let's go back to MailChimp dot com slash developer and click on get started to see how we can start interacting with the MailChimp API. If you scroll down on this page you can see that they've got a section on code examples

which is usually a nice way of understanding how to work with a particular API.

So here they've got a curl request which is actually something that's done in the terminal but in our case we're going to implement this using our code.

So take a look at this data section. This is the sort of data that they would expect us to send to their server. So if I just copy this and paste it into our code then you can see that this is essentially a flat pack Json right. So it's all full of strings and it's delineated by these curly braces and colons.

So what we would have to do is to create our data that we want to post as a Jason. So first let's create the javascript object so let's create a new variable called data and I'm going to set that as a new javascript object.

Now inside our data object we have to provide all our key value pairs with keys that MailChimp is going to recognize.

So if we go back to the API reference and go to our list slash audiences you can see that our data is going to be sent via the body parameters using a key code members.

Now this members happens to be an array of objects each representing a member that we want to subscribe. So we can add up to 500 in our array to subscribe at the same time. But if we click on Show properties you can see that each of these members can have their own properties like the email address for the subscriber the status and the merge fields.

These are the merge fields that will contain their first names or last names.

Let's create are members key value pair first. So our data is going to be called members and this members remember has to be an array of objects. So we're going to open up another set of objects and we're only going to have a single object in our array because we're only going to subscribe one person at a time.

Now the email key is called email address. And just to prevent myself making a typo I'm just going to copy and paste it in here and the value of this key is going to be the email that we got from the body of the post request. So let's put the e-mail here and then the next one is going to be these status.

So let's again copy and paste it and you can see it can have four possible values subscribed unsubscribe cleaned or pending and we're going to go for subscribed. So status is equal to the strings subscribed. And then the next one is going to be the merge fields and the merge fields. Notice how this is now an object. So we have to put in our merge fields and then open up a set of curly braces and you can set the names of these merge fields by going to your audience and then going to settings and going to audience fields and merge tags.

So if you take a look down this list most contacts come by default with a first name last name birthday and the word that's inside this field is the merge field and by default MailChimp sets it to f name an L name in all caps. You can either change that or as I would recommend keeping it the same. So let's use f name as the first key which is going to hold the first name from the form and then l name which is going to hold the last name from the form.

Now we have our data object completed but this is javascript and what we need is actually to turn this into a flat pack Json. So I'm going to create a new variable called Jason data and I'm going to set this to Json dot string if I and I'm going to pass in my data into here so that I turn this data into a string that is in the format of a Json now because I'm actually never going to change any of these variables.

I can actually change them all to constants so now we have our Json data.

This is what we're going to send to MailChimp and we're ready for the next step which is to make our request. Now remember previously we used the ATP s module and all that we did was say 80 G.P.S. dot gets and then inside here we put in a your cell and then a callback function. Right. But that as you can guess only makes get requests when we want data from an external resource. But in our case we actually want to post data to the external resource.

So if we go back to the documentation of the H2 CPS module you can see that in addition to a CPS dot get there's also one called Dot request. And in this case in addition to adding a U.R.L. to make our request to we can also specify some options. Now it tells us that these options Accept all options from the ATP dot request module. So let's click on it and see what options we can have. So in our options there's something called Method and this allows us to specify the type of request we want to make.

For example get or in our case it'll be post heading back to our code let's go ahead and create a constant for the HP G.P.S. module and require it. Now down here just below where we've got our Jason data but still within the app dot post we're going to create our request.

So we're gonna say 80 G.P.S. dot request and we're going to pass in firstly are you RL And then we're going to pass in our options. And finally we're going to have a callback which is going to give us a response from the MailChimp server. So let's fill in each of these fields.

Now your role is going to come from the main MailChimp endpoint. We can find out which you are allowed to use by going to the MailChimp API documentation.

Section

Code Examples

So in the section code example it shows us that this is the U.R.L. that they're using. And in fact all the way up to here is the MailChimp API end point and then afterwards we have optional paths that we can tag on and if we wanted to subscribe some members then we would be posting to forward slash lists forward slash the list I.D. that we want to target. So I'm going to copy this entire your URL and I'm going to paste it into my you are all here.

Now in addition I have to add a list I.D. because in MailChimp you could have multiple lists and you have to tell it which list you want to subscribe members into. Now remember previously I'd already saved the list I.D. of my main audience and I've really got it down here. I'm just going to copy that and tag that to the end of the U.R.L..

So there's just one last thing before we're done with the U.R.L..

Notice how here it says U.S. and then it's got an X.

You have to replace that X with the number that you have in your API key after the word U.S. MailChimp has several servers that they're running simultaneously because they're a big operation. And when you sign up you get randomly assigned to one of them. And it could be anywhere from us one to us 20. Take a look at your API key and see what number you've got after the word us and then copy that number and replace the X with that number. So now we finally completed. Are you RL The next step is to create some options.

Javascript Object

Options

So I'm going to call it options and it's going to be a javascript object now the most important option I have to specify is the method. So let's specify a method and let's set it to post.

Now the next thing I have to do if I want this POST request to go through successfully is to provide some form of authentication. We've already got hold of our API Key earlier on from MailChimp but now we need to think about how we're going to use it. If we go back to get started and take a look at authentication methods it tells us that we can use the basic HP authentication and enter any string as a user name and our API key as the password. For example it might look something like this. Any string coal on your API key.

Authentication

Now in the options for our ATP request function there is something called auth which allows us to do this form of basic authentication. So that's what we're going to add next. In addition to the method key in our options we're going to add auth colon and then we're going to specify a string.

Now MailChimp said that we could use any string as the user name and then separate the user name from the password which is going to be the API key with the colon. So it should look something like this.

It doesn't matter what you write here as long as you have something and then you have a colon and then you paste in your API key making sure that your region in the API key matches with the region in your U.R.L..

So now we're finally ready to make our HB request and when we get back a response we're going to check what data they sent us. So we're going to see a response start on just like how we did before. We're going to specify that we're looking out for any data that we get sent back from the MailChimp server and then we're just going to log our data but we're going to use the Json parse to parse it first.

So now some of you might be wondering OK. So we made our request but no where in this request or in our U.R.L. or in our options have we specified what it is that we want to send to MailChimp namely our Jason data which

comes from here and it includes the email address and the first and last name of our subscriber. So that's a very good point.

In order to send that data if we take a look back at the documentation for the request method we have to save our request in a constant. And then later on we can use that constant request to send things over to the MailChimp server by calling request.

All right let's go ahead and create a new constant code request because we don't want to conflict with our req here and then we're going to use this request and we're going to call request dot write. And inside all right we're going to pass the Jason data to the MailChimp server and then to specify that we're done with the request we call request dot end. And now let's test it out. So let's hit save on our update J.S. and let's just check hyper to make sure that we don't have any errors.

And now let's go to our local host three thousand and try to sign up. So I'm going to put in my own name and then I'm just gonna use a random email let's call it. I don't know. 4 five six eight email dot com and click sign me up. Now at this point if we look inside our terminal you'll see what we get sent back as the data from MailChimp. And it seems to suggest that a new member has been added and there were no errors in this process.

So now if we go to our MailChimp account and go to our audience you should be at to see that you should have a new contact. And if you click on that number one or two that you might see there you should see the e-mail that we just inputted the first name and the last name and it's all now been added to our audience.

And if you want to add another member you can go to the U RL bar hit enter again to restart it and add a new person. Now when I click sign me up and go back to our audience and hit refresh you should see we now have two entries and you can keep adding this but our website still has a number of small faults. Right.

Like when we click sign me up it kind of just hangs it doesn't do anything doesn't give any feedback to the user whether it was successful or whether if it failed.

So in the next lesson. That's exactly what we're gonna be doing. We're going to be sending the user to a success page if everything went well. And we're going to be sending them to a problem page if it went badly. So have a play around with your existing web page. And once you already head over to the next lesson.

250 Adding Success and Failure Pages

In the last lesson we mentioned that it's not very good user experience to let your Web site simply hang in timeout once the user has submitted their details.

They're not going to have much confidence in this newsletter if we don't give them some feedback. Right. So let's go ahead and do that. This is gonna be a challenge for you.

How can we figure out what response code we got back after making this HP request. And if that code was two hundred then it probably means that things went OK and we're going to send the user a piece of tags to be displayed in the website.

Now if the code wasn't two hundred and we got a different code then we're going to send them a different message. Something about the process failed and they should try again later. So pause the video and see if you can complete this challenge. All right.

So we know that we can get hold of the response object that we get back from making the request and we can tap into the status code to see whether if it was 200 or whether if it was something else what if we check whether if the status code was equal to 200 and if so then we go ahead and use red dot send to send a piece of tax to be displayed in the Web site.

Now you could of course send a paragraph or in each one element but I'm just going to send a very simple text. I'm going to say success fully subscribed and if the code was not 200 then I'm going to residents and there was an error with signing up. Please try again.

So now let's go ahead and test it out.

When I click sign me up I get successfully subscribed. And this is written in the website. But let's say that the API Key was wrong or if the list I.D. was wrong. Well then in this case when I tried to sign up then I get there was an error with signing up. Please try again this works.

But it's a bit plain right.

Custom Message

Bootstrap

Nobody wants to call on a website that looks nice initially and then after a bit of interaction it ends up looking really really plain. What if we wanted to use bootstrap to jazz it up a bit. Well we can because previously we created the beginnings of the success thought HMO and the failure dot HMO.

In addition to all sign up dot each team out here we can include some bootstrap code that will quickly give us a nice looking page that looks a lot better than what we had before. And we can also include button and allow the user to interact with it.

Jumbotron Boilerplate

So I'm gonna head over to the bootstrap Web site and I'm gonna head over to components and I'm gonna add a jumbotron to our Web site something that looks a bit like this.

So let's go ahead and copy all of this code and we're going to add our HMO boilerplate. This one is going to have a title of failure. Sounds a bit sad but let's paste our code for our jumbotron and the each one is going to say oh ohoh.

And we're also going to have a jumbotron in our success not HMO. And this one is going to say "Awesome", now in order to use bootstrap of course we need bootstrap style sheet. So let's go ahead and copy that over to our success and our failure. Web sites

Challenge

Now it's back to you for a challenge instead of using resort sand to send these messages can you figure out how we can send these. Success and failure. Web pages over instead. Pause the video and try to complete the challenge.

All right so that's gonna be as simple as changing resident send to read that send file and then we add our directory and then we tag on our file name which is going to be forward slash success dot HMO the name of this file and then

this line is going to be read that send file failure dot HMO. So now let's check it out try some random stuff and let's hit.

Sign me up and awesome.

You've been successfully signed up to the newsletter look forward lots of awesome content so that's pretty cool and let's test out are failure site there's one other problem that I'd like to address.

Add Try Again Button And Step After Failure Message

It would be nice if they failed on signing up to be able to try again. And if they had a big button here that said try again. That takes them back to the original page over here. That would be pretty nice right. We can do this by using our page over here. tn can simply add a form that includes a submit button and this button will say try again and it will have the action of posting to a new route which is the failure route. Now our form is not going to have a class but our button is going to get some bootstrap classes namely Btn Btn large Btn warning to turn it yellow

so now let's try again to go to our failure page and now you can see we have a try again button but at the moment it doesn't work because we don't yet have a route defined for the failure route. That's making a post request. So let's do that here below our post request for the home route.

I'm going to add another one for our failure round and this one is going to have a completion handler that redirects the user to the home run so I can simply say rest don't redirect and the path I want to redirect it to is simply the home route.

So now when we end up on that failure route then when we click this button it will take us right back to the sign up page. So that means when I click this button that says try again we're going to trigger a post request to the failure route and that is gonna be caught by our server over here and it's going to redirect to the home route which triggers this app dot get and it sends the sign up page as the file to be rendered on screen.

So that's pretty much all the functionality that we need for our very simple newsletter sign up page. Now there's a whole world of customization that you can add to this. And of course you can make it even more complex with better pages better rendering and customizing these pages.

But what's more important than that is how do we get our server to not just be hosted locally because at the moment this website only works when we're running our servers locally on our computer.

How can I put this onto the worldwide web so that we can send our friends and family and clans and customers a U.R.L. so that they can go ahead and use this Web site and sign up to our MailChimp list as well.

Well that is what we're going to tackle in the next lesson where we make our website go live so that it's a real web app available on the worldwide web for all of that. And we'll ask you on the next lesson.

252 Tip from Angela - Location, Location, Location!

Now one of the common things that I hear students say is hey why is it that when I go through the tutorial everything seems so easy and so obvious. But then what I try to do something myself like create a project myself or try to complete a code challenge and it seems like almost as if I actually didn't learn anything at all.

Practice

Well the reason is because that the learning comes from the practice.

So don't view watching the videos as learning. That's just me introducing you to a concept.

It's only when you apply it yourself then you actually learn.

So always tried to watch the video see what's happening what the gist is. If you take down a few notes names of functions or things that you might need to use and then pause the video and try to complete it yourself that way you're actually putting the skills into practice and you're actually learning instead of just absorbing passively.

Psychology of Learning

And one of the biggest learnings I've read from and one of the most interesting things from psychology studies is something called the Dunning Kruger effect which is where when you first start learning something you can feel like wow I know so much I know everything there is to know about this topic.

But then as you gain more knowledge you realize Oh no actually there's a lot more stuff out there that I don't know. And then you fall into a pit which is called the valley of despair and this is also sometimes referred to as imposter syndrome where even then you feel like you've spent a lot of time learning this skill. You feel a bit like an impostor because you think people are going to find out that I actually don't know anything.

And both of these are exaggerations.

You are somewhere in between both of those places.

Keep Practicing

And if you just keep persevering keep practicing keep putting what you've learned into action then you're gonna get there in the end.

So keep going.