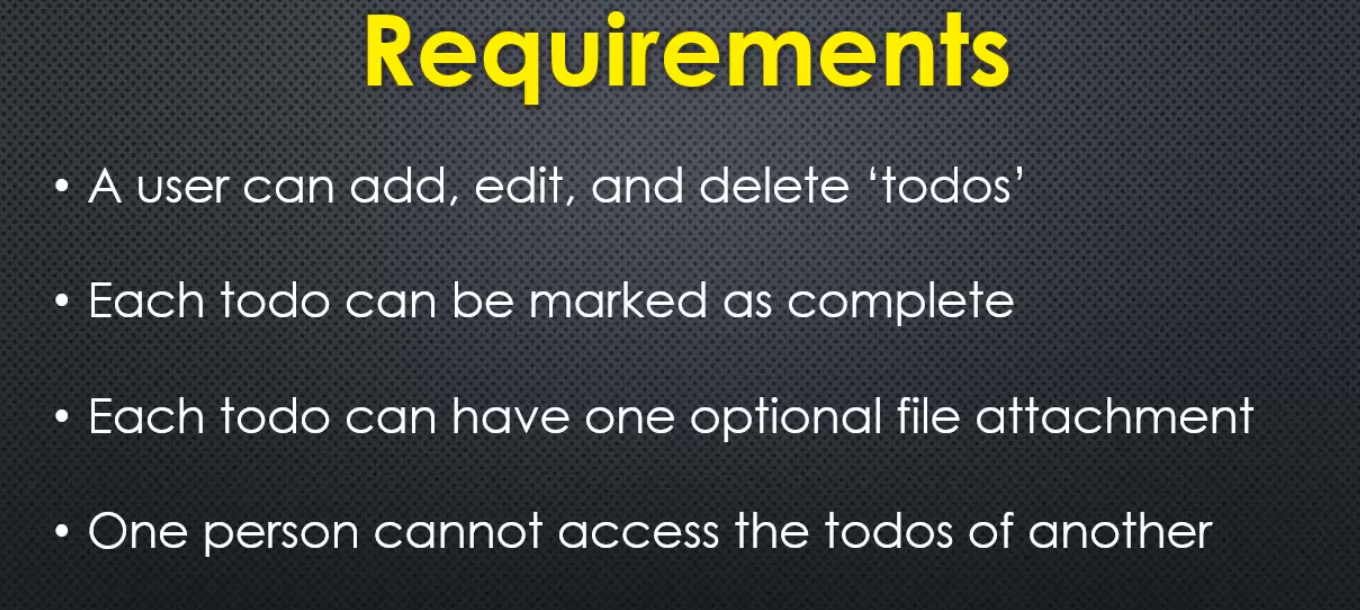
# **Let’s build an Application**

## **NodeTodo : Software requirement**

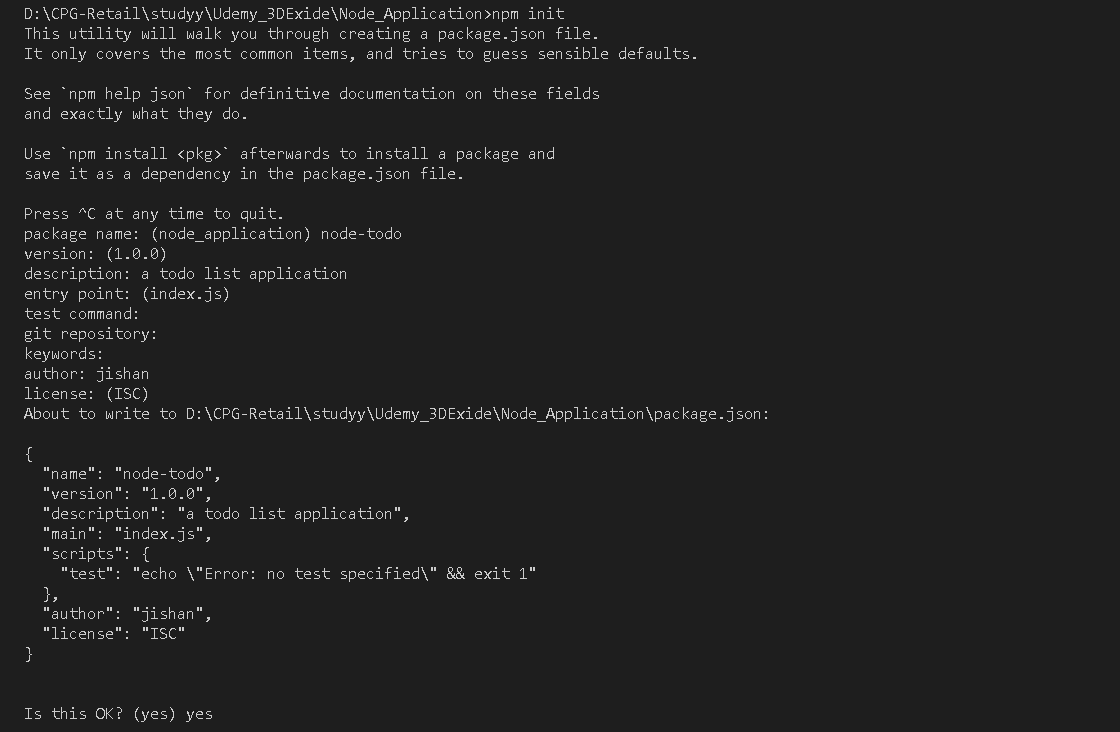


## **Initial Setup**

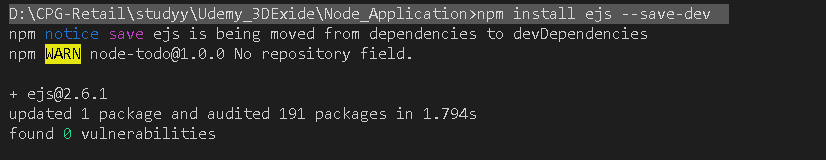
I'm starting off with an empty folder, that's it. So I'm going to use NPM init to create my package.json

and then instal some NPM packages

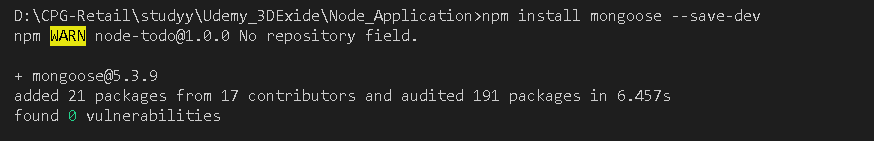
I know I'm going to need.

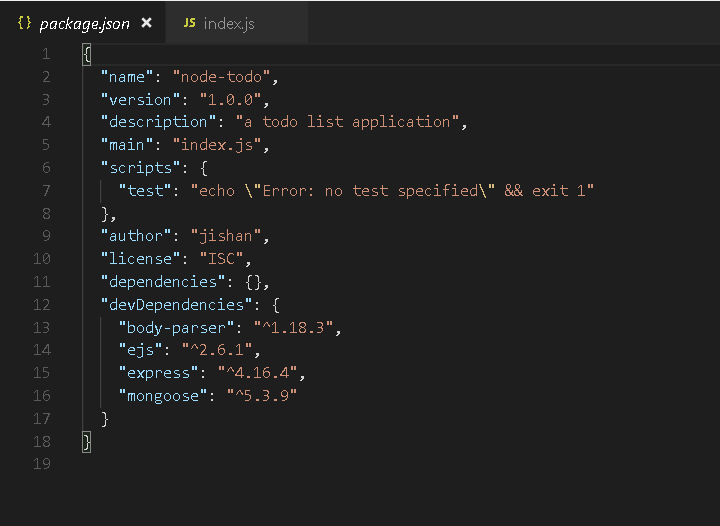


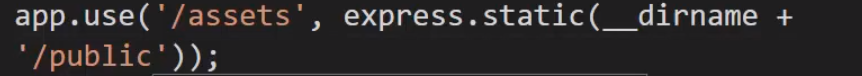








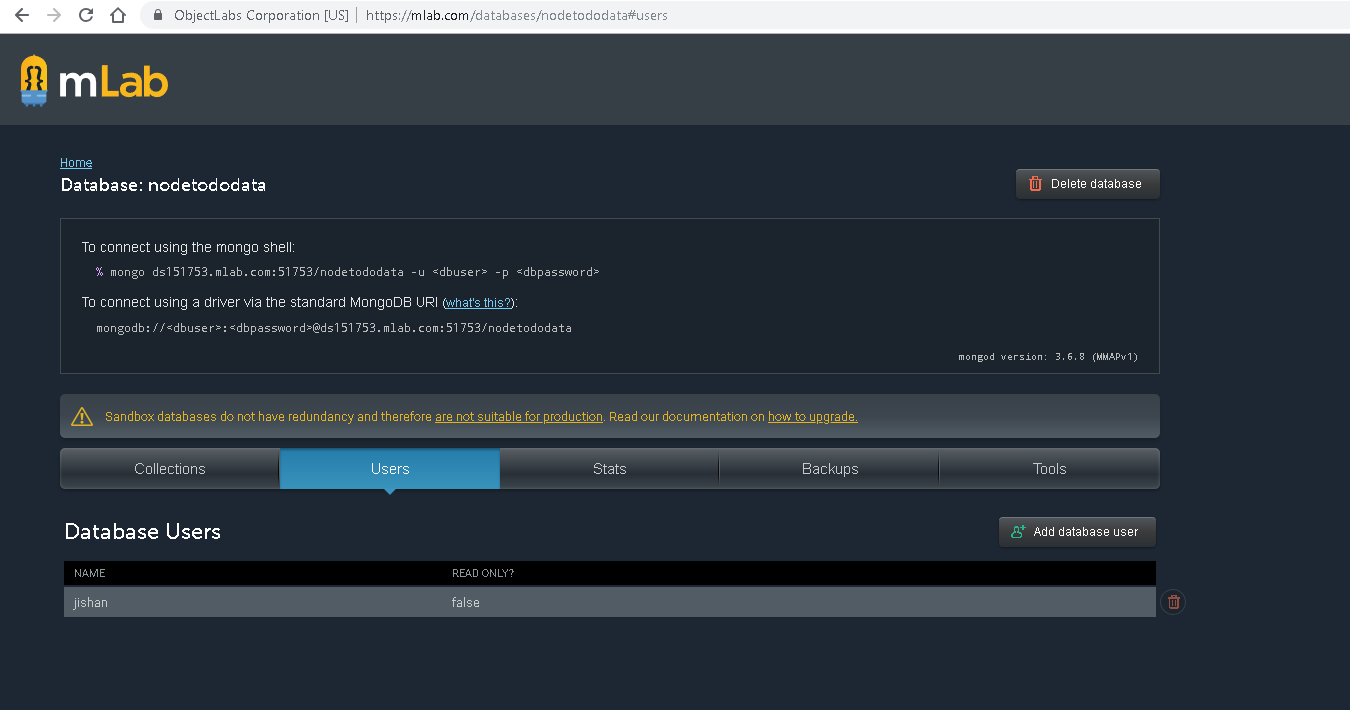


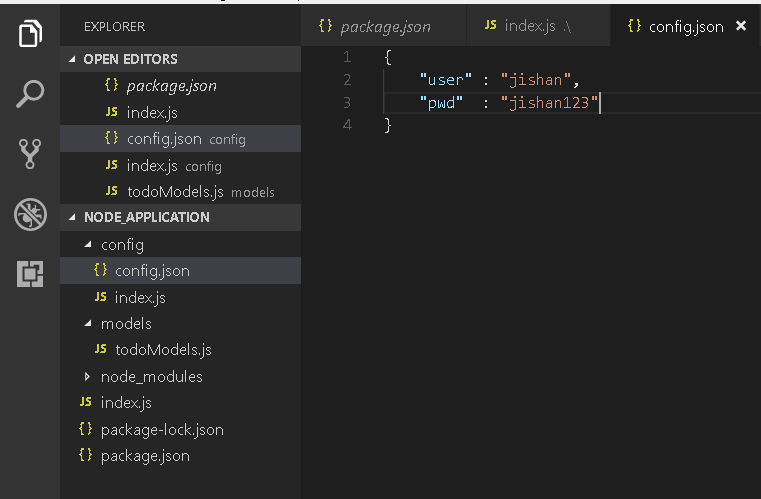


**I also know that I'm going to be building some code for the browser, my front end, so I might as well already set up my public assets folder, which will just be delivered straight to the browser.**



## **Setting up mongo and Mongoose**







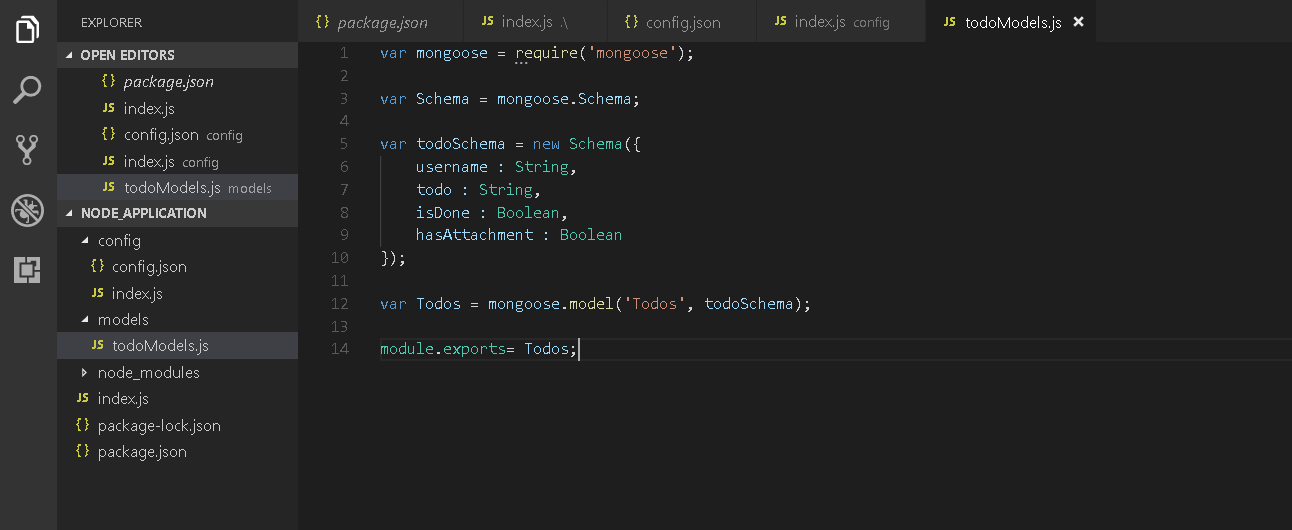
What else do I need to do?

Well before I actually make the connexion to the database,

what I'm going to do is define what my data looks like.

So I'll make another folder.

We'll call this models.



So now I have a reusable model.

I can create to-dos.

I'll be able to delete them and update them

and find them using the to-dos object being exported

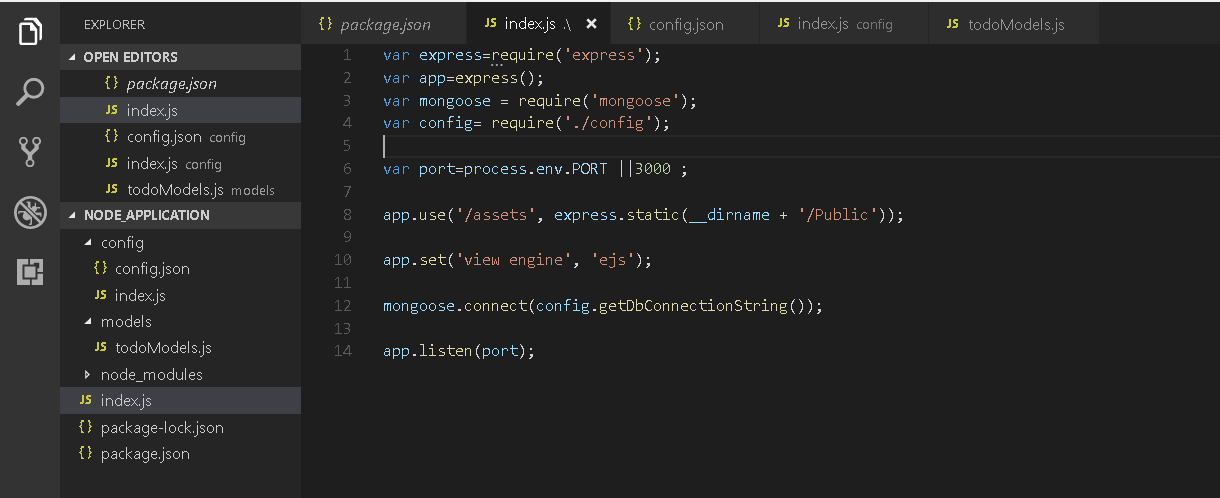
because mongoose.model will give that to me.

It'll give me the schema

and then it will provide all the methods

to be able to create, update, delete

records inside those files inside the MongoDB database.



'm set up now to connect to my Mongoose database

## **Adding Seed Data**

Documentation remaining….

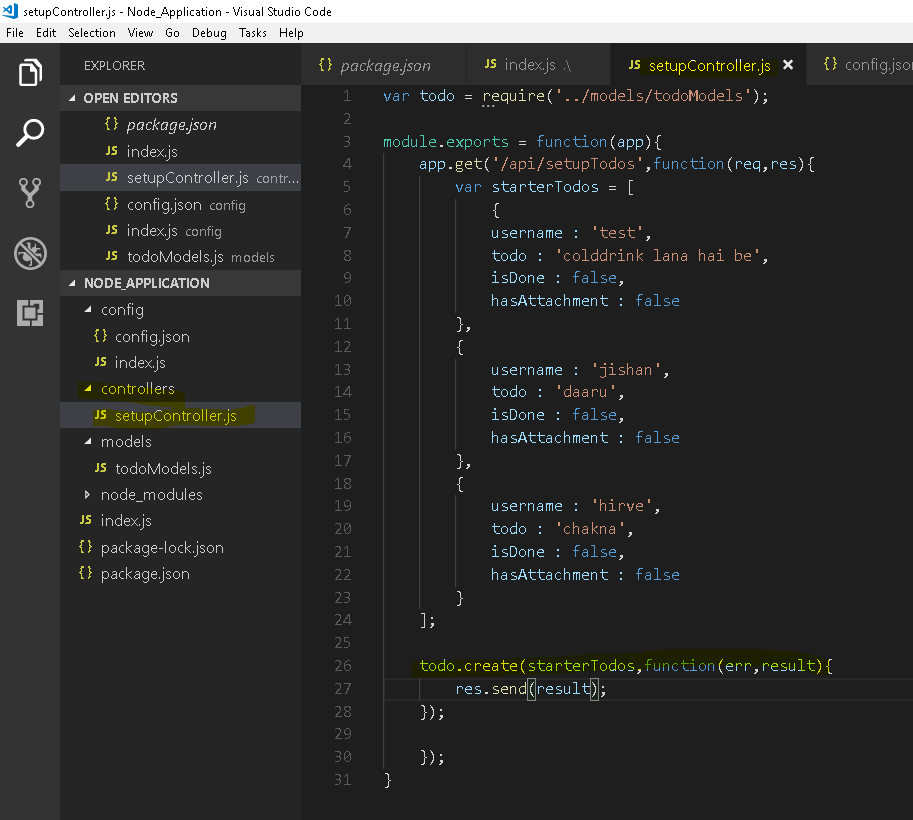


you also may want sample user data, a seed routine, so that you can spin up a new database. You just run this routine, get some data in it, and start working on your application.

in order to set up the seed data?

Well, this is Node and Express, so I'd go to some URL, run that URL and it would set up my seed data.

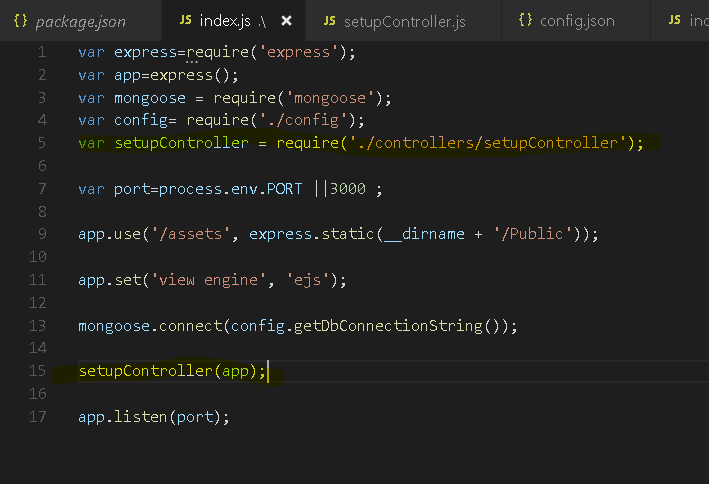
So, I'm going to go ahead and create a controller's folder. Remember, we did this earlier. I'm separating out different parts of my application to make my code a bit easier to manage.



I have todomodels available to me. Because, remember, that's what I called it

getting back from require. And **this was a Mongoose created model, which means I have certain methods available to me like create, and create will accept an array.**

**Now we have to pass app to controller**



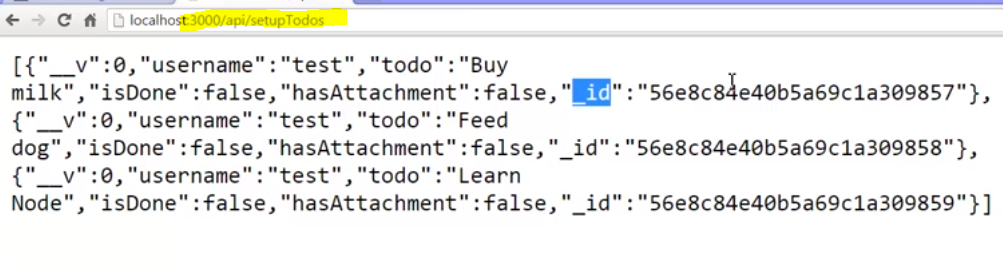
I'll run that endpoint.

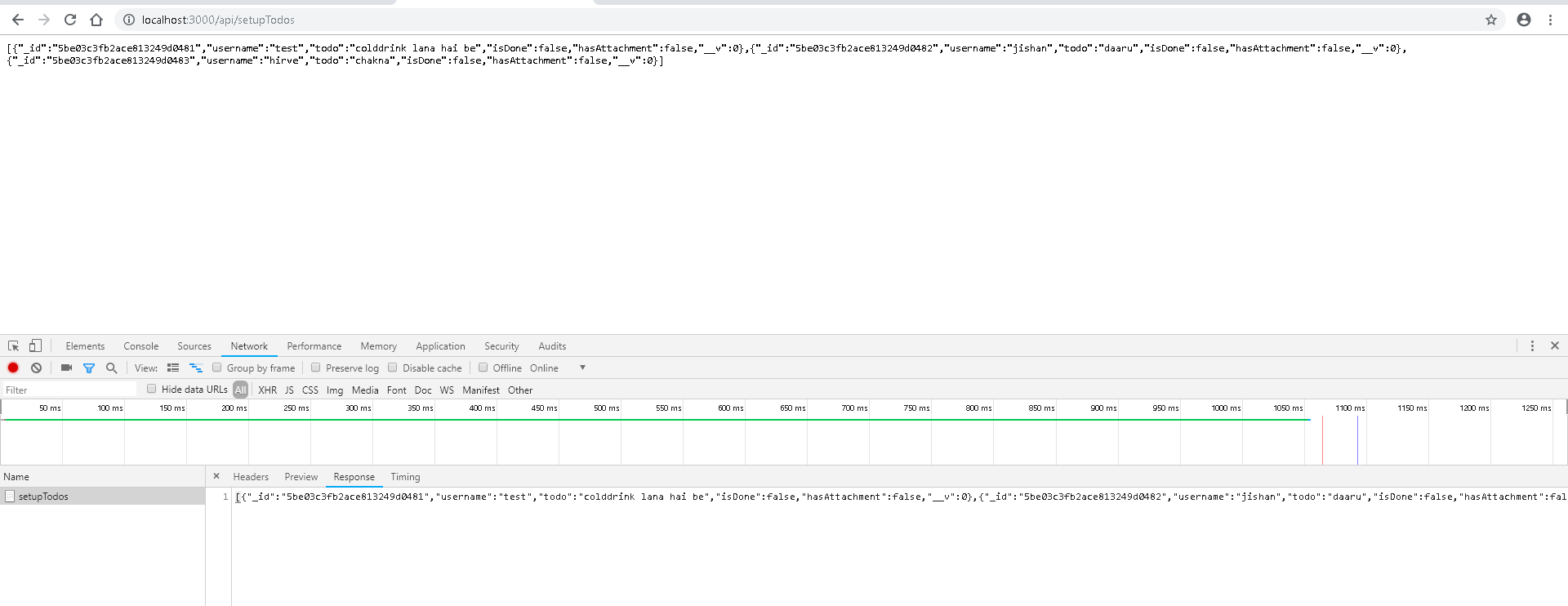
It should connect to Mongoose

and MongoDB will see that schema for the first time,

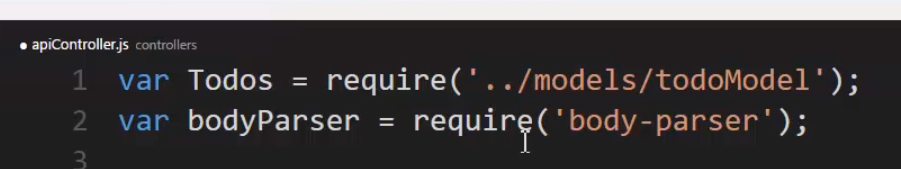
set it up, and then via the Mongoose create method,

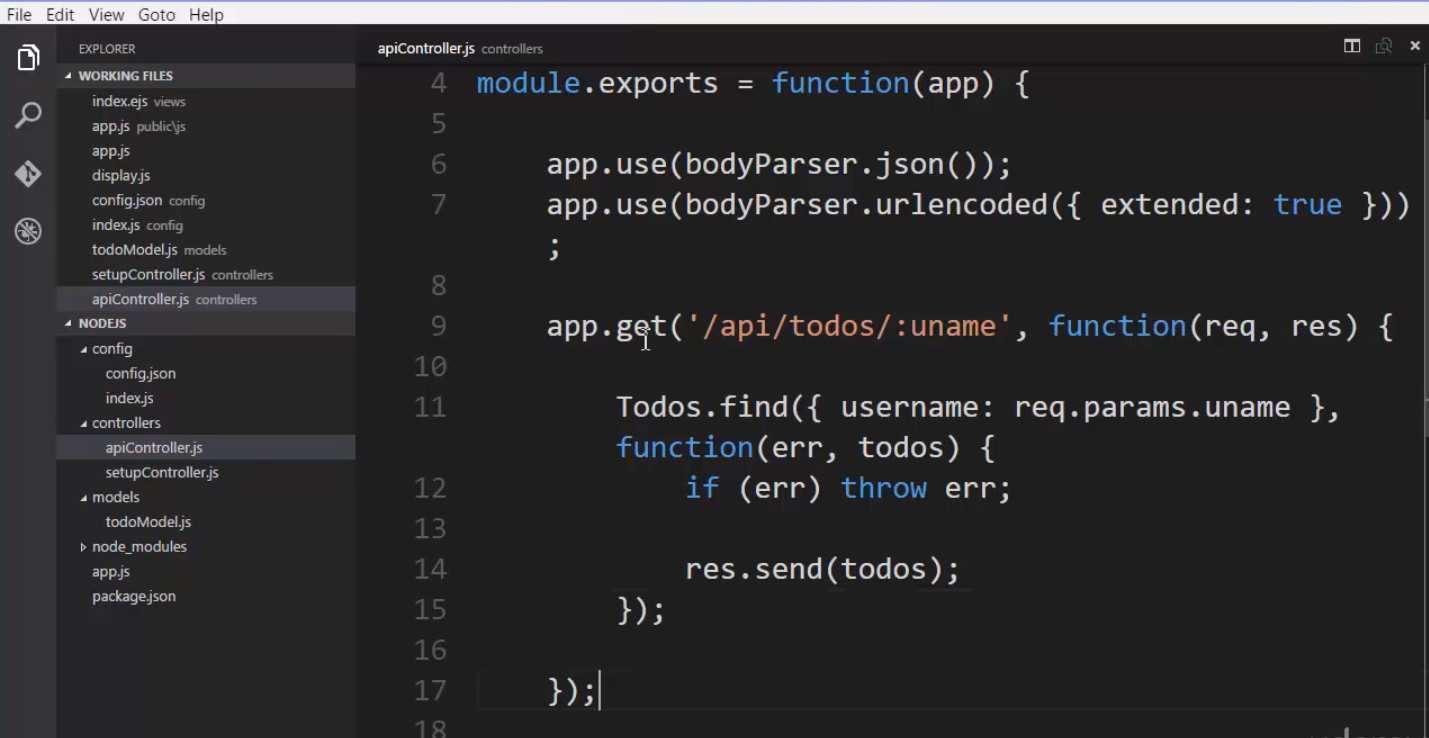
we'll add that data into it.

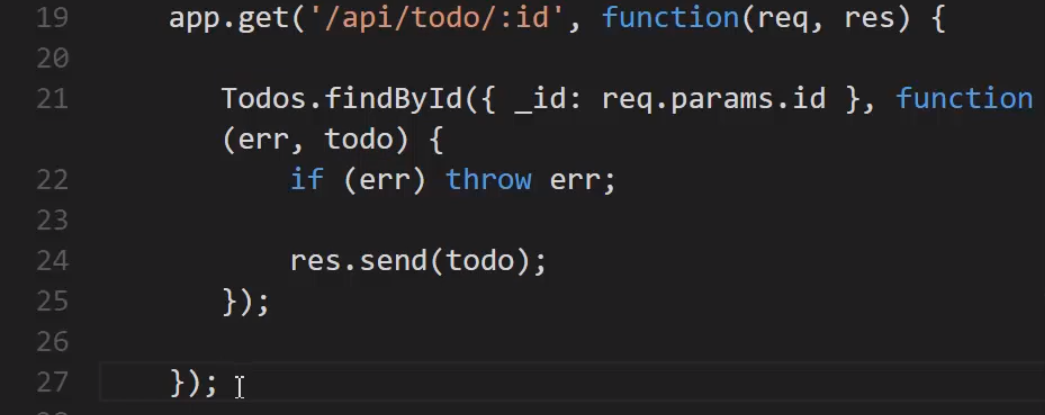


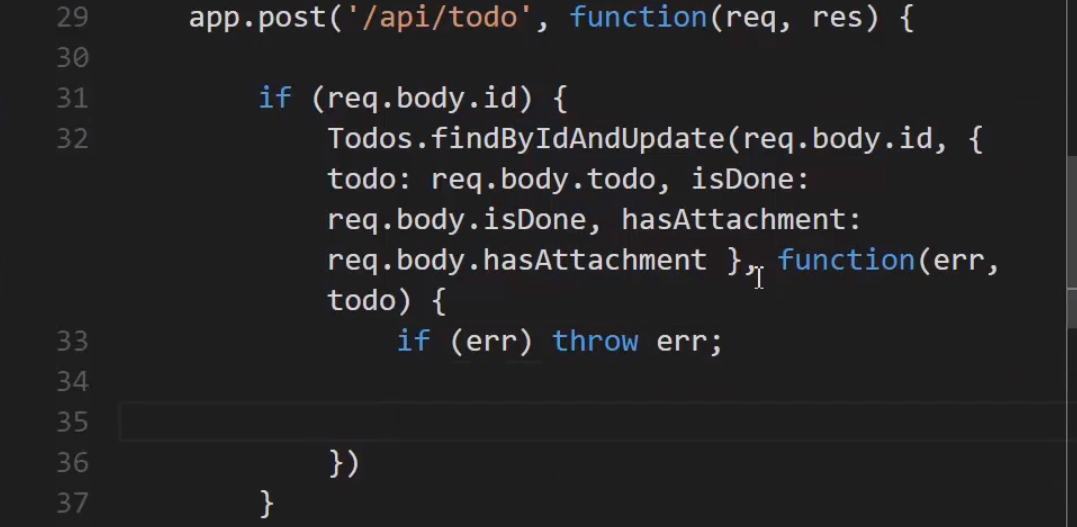


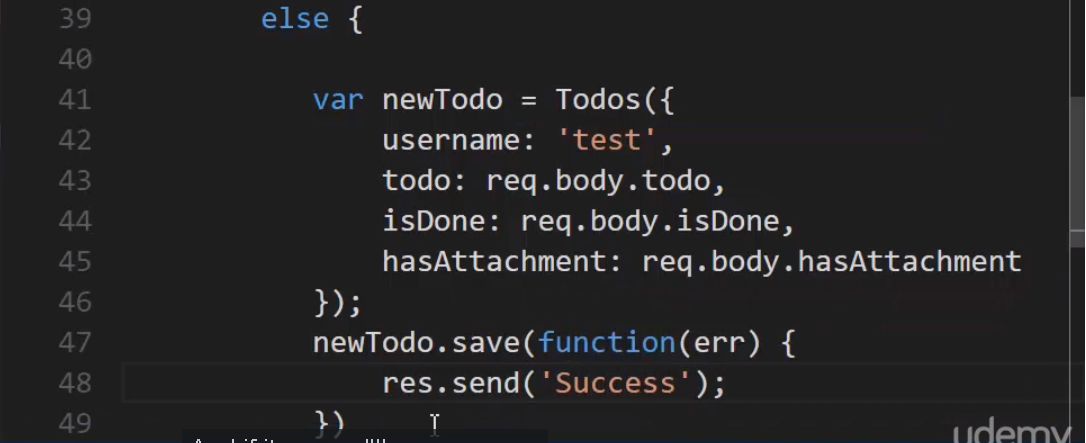
## **Creating our apis**

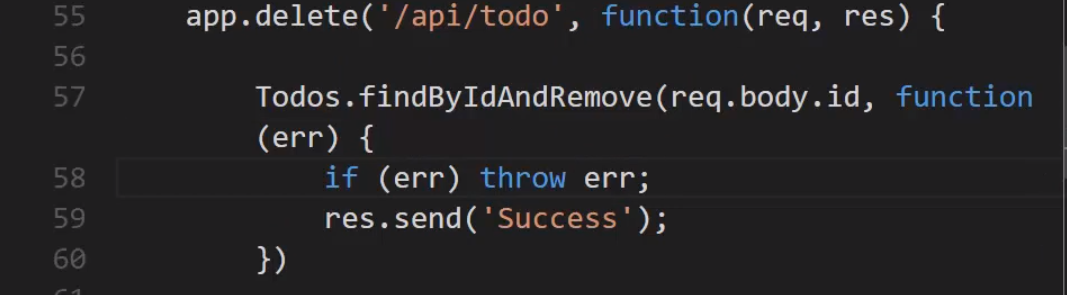
Lets develop CRUD Apis (endpoints)

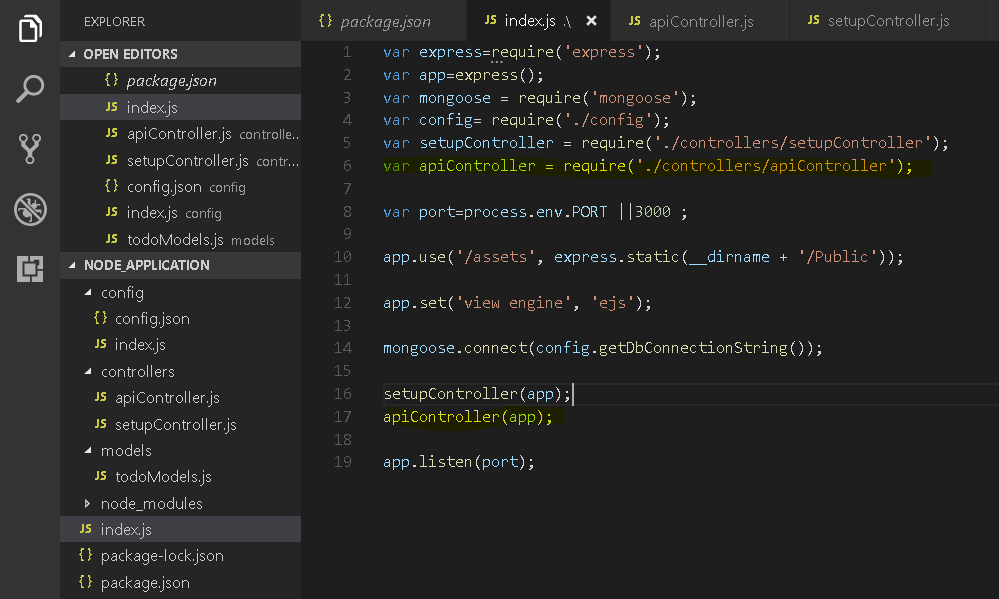


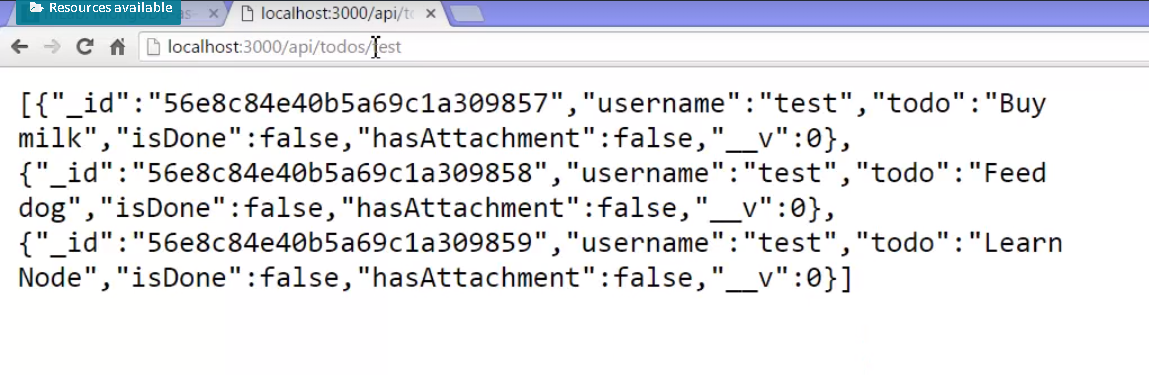


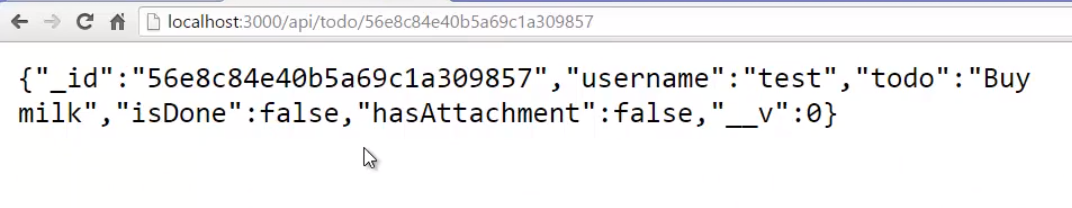








Output - 



Now this is easy to do with the browser to do gets

but post and delete is a little tougher

and I'd really rather not have to

build a whole front end just to test my API.

How am I going to post some data?

How am I going to set up an http post or delete?.

Well there's actually utilities out there

to help you with that.

So let's do that real quick.

Let's fully test our API before we even

start building the rest of our application.

## **Testing our Api**

So let's move on to see how we can test

our API without having to build a whole bunch

of interface in the browser to try it out.



we're going to use postman to test our API.

Without, again, having to build anything.

it lets you do is generate HTTP requests.

So I can create get requests, post requests,

delete requests, and see the responses come back

from the server.

I'll say localhost:3000/api/todos/test

which we did it in the browser previously.

And we saw those results come back from the browser

but Postman will do the exact same thing.

But notice that I get the status,

how long it took, and the actual results.

And I see my JSON results.

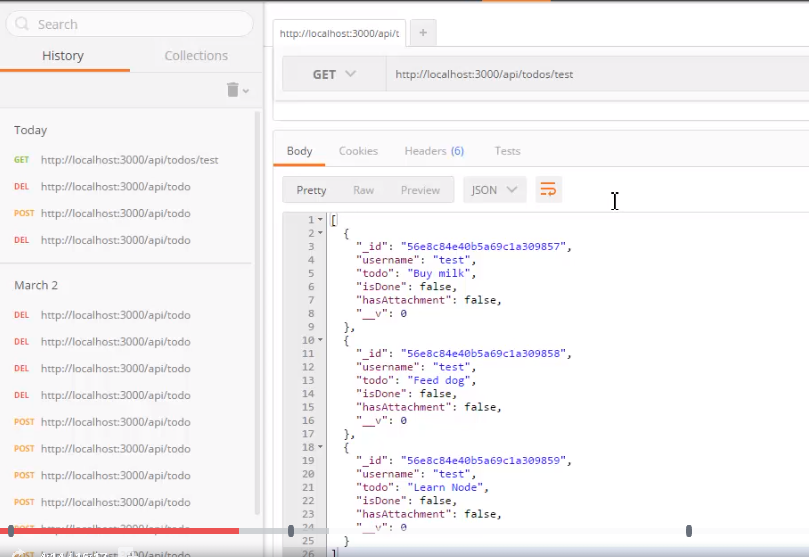
And I can see the headers that came back.

The HTTP headers.

And the format that I'm seeing.

So, I've just tested my API.

I did exactly the same thing we did in the browser.



Now, of course, this was easy to do in the browser. But post and delete are a little bit harder. But here I'll be able to do them quite easily.

Before we look at posting and deleting,

let's take another look at our endpoints

to make sure we understand what they're looking for.

Again we're posting to /api/todo.

And we're using request dot body.

And that means it's going to look at the HTTP request body

and assume that it's JSON,

convert that to a JavaScript object and then let us look at

the properties sitting on that object.

So body will be a JavaScript object created from

the JSON sitting in the body of the HTTP request.

And if there is an id

on that object then it's going to do an update.

And if there's not

then it's simply going to create a new todo,

for now with the username 'test',

it's going to expect a field called todo.

Our property called isDone.

A field of property called hasAttachment.

And then save it to the database

and give us back 'Success' as an HTTP response.

For the delete it's going to expect just the id

and that's still going to be a property

on a JavaScript object created from some JSON.

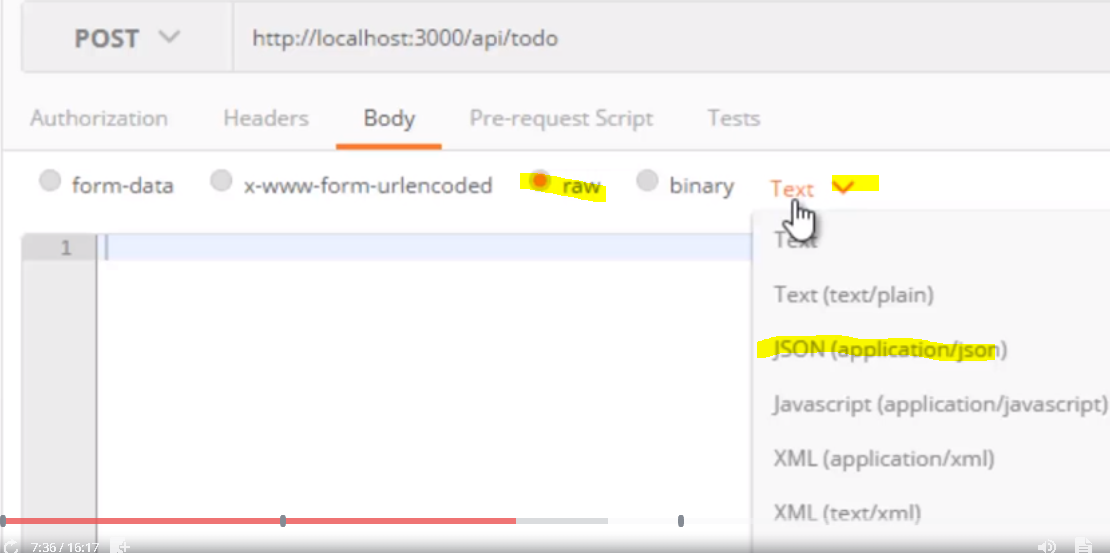
So we'll send JSON, as the HTTP body,

for both delete and post.

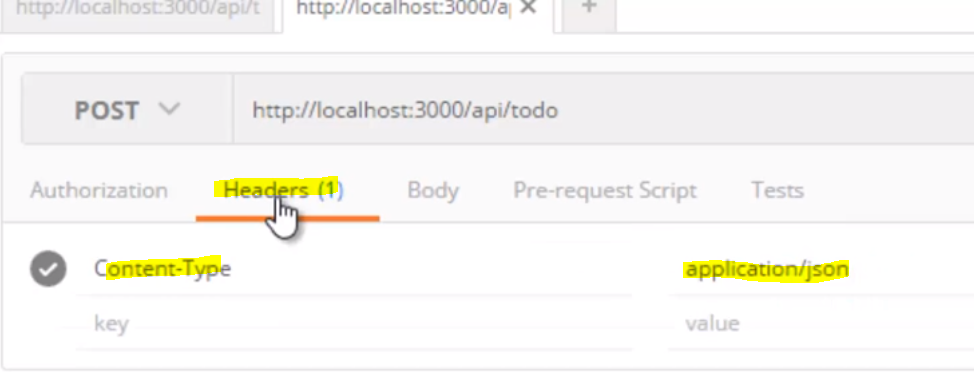
And then we should expect to see a success message come back.

Postman lets me specify the body. If I said form-data this would be basically like a query string. But if we had posted a form in our browser then this would be of the same format. A name and value. Name value pairs.

Instead I'm going to say raw. Which means I'm actually just going to type straight in some raw data. But instead of text I'm going to say, right here where I can specify the type of raw data, JSON. That's what Node.js is going to be expecting.



And you'll notice that what that does, if I go to the headers tab, is it added an HTTP header of Content-Type specifying that this is application/json.



So this is a standard way of saying, "Hey, this is an HTTP request and this header, "which tells us more information about the request, "tells us that the content in the body "of the HTTP request is JSON." And that's what Node expects so it'll handle it properly.



I sent an HTTP request with the verb post.

Node.js handled it.

I sent it to localhost:3000 so that was the port.

Node.js was the process on the computer

that ended up handling, then, that request.

And after we posted it,

Express took that HTTP request,

Node and then Express helped it in order to be able to say,

"Well now this is the function that should run."

In response to the URL that was requested,

and that's part of the HTTP request, what URL it was.

Then it looked at the body and took this JSON and turned this into a JavaScript object that was made available via request, or R-E-Q in our case, dot body. So this thing became an object, in JavaScript dot body, on the request object.

And that's why we were able to say,

request dot body dot todo dot isDone dot hasAttachement.

So that's very powerful.

Now let's try updating this.

Remember that the same endpoint, /api/todo,

for the post handles both creates and updates.

Now we could have done this differently

but that's the way I chose to do it.

And the only difference being that it will do an update

if it finds a property dot id

inside this JSON.

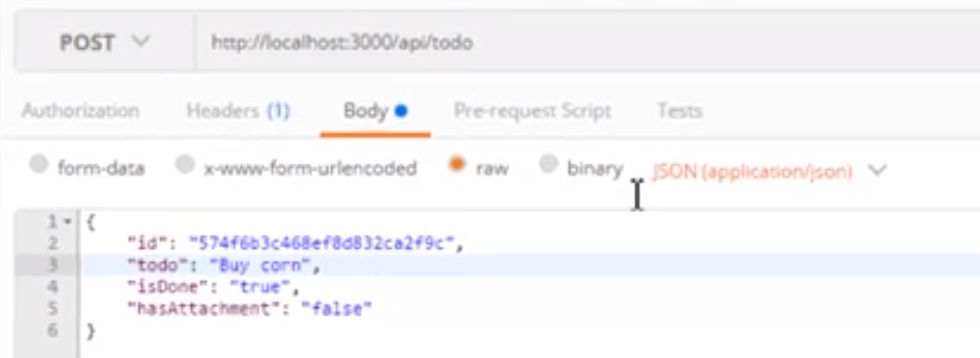
And it will say, "Ah, you mean to update

"this particular object already created,

"this particular todo that's already created,

"and we'll update it instead."

That's how we coded it.



Let's try deleting this todo. We created code to do that.

And remember that all it expects in the body

is a JavaScript object, created from JSON,

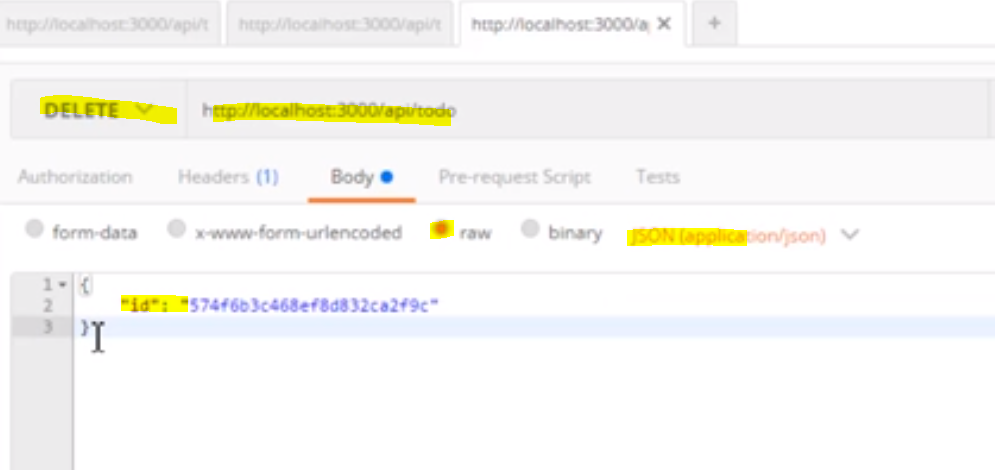
that has the id in.

Nothing else. It doesn't need any other information.

It just needs to know the id

of the todo that we want to delete.

That's how we coded it.



Some of you might call that CRUD.

Create, read, update, delete.

And it's the foundation of most of

the applications we build, isn't it?

Creating, reading, updating and deleting data.

Usually storing it in a database.

Now, if you think about it,

what we've built is not a web application.

What we've built is a web API.

And that means that even though we planned

on using AngularJS, in order to access the data

and update the data via the API, we don't have to.

We could build an iOS application.

We could build an Android application.

We could build a Windows application

that uses the API to update, read, create, delete,

todo's that are stored in the database.

In fact, we could create all of those

and have multiple ways of accessing our data.

Multiple platforms or front-ends.

In our case we're going to use AngularJS.

But the power of creating your API first

is that now you've expanded your options for the future.

Maybe you're creating a web application,

that's accessed in a browser, but one day

if it gets really popular you'd like to create an app.

So build an API first.

That just seems to make sense, doesn't it?

That way you only have to worry

about building something that accesses the API.

And all that issue of updating the database

is already taken care of for you.

So we've checked our API,

without having to build any AngularJS code,

and it works great.

## **Adding a front-end in Angular-2**

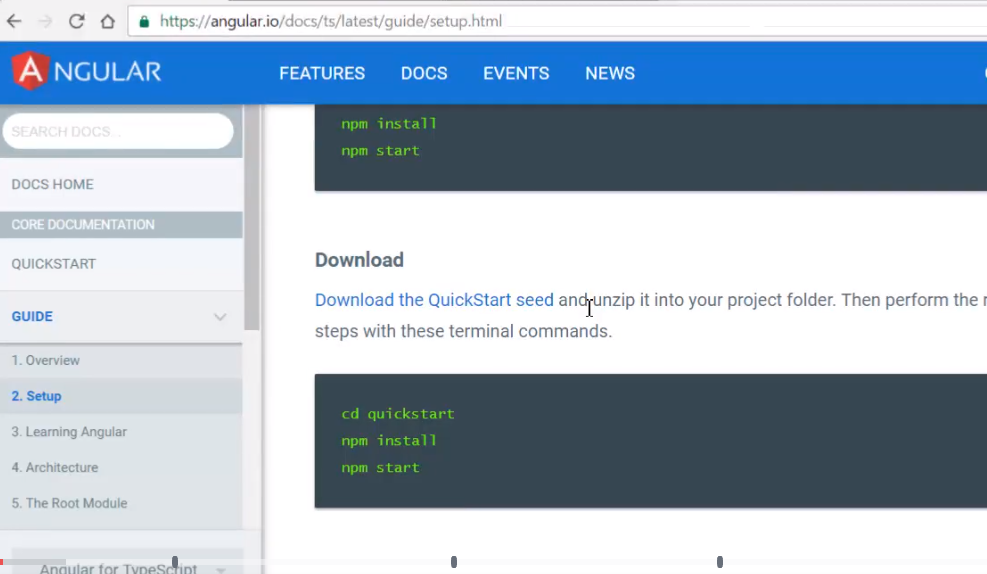
Node.js APIs are just web APIs

that can be used by any technology,

but we are going to focus on AngularJS 2

as part of the MEAN stack, so let's take a quick look.

We can go to angular.io, which is the Angular 2 website.



 I've already downloaded the zip file.

I've extracted it and put it into the public folder

of my Node.js application.

Let's take a look at what that looks like

and what we had to do to get everything up and running.

 I've added a public folder which we've used before.

But instead, I unzipped the Angular2 QuickStart

and dropped the entire thing in the public folder.

Now, this is frontend development using Node and npm

which means it has its own package.json,

but that's being used to do things that generate code

for the browser that generate JavaScript files

that the browser will download.

So there are a few ways that you can set this up.

Some people might like to mix the package.json

for Node with the things that are going on

with the frontend web development like AngularJS 2.

However, in this case, to make things clear,

I'm not going to do that.

I'll just keep the two things entirely separate.

That means I have my own node\_modules folder

in the public folder,

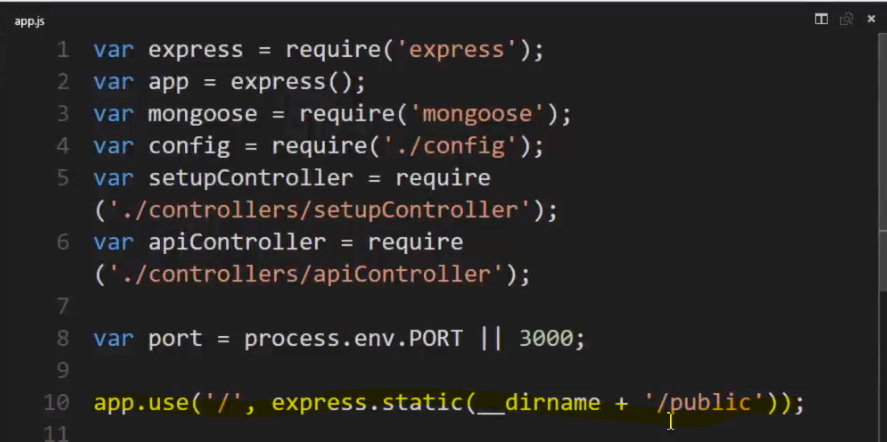
in particular, just for my frontend application.

And my node\_modules in the root is for my package.json

that we've already been using for my Node.js server side

of the application.

So that means that I need to tell Node.js that I'll be serving my application from the public folder, and I do that in app.js. We might remember this, I used app.use and told it that the public folder is actually now just /, meaning if I just go to the home of the website, then serve whatever is inside the public folder.



And in the public folder, there's an index.html file,

this came along in the QuickStart.

I could make this an ejs file if I wanted to do

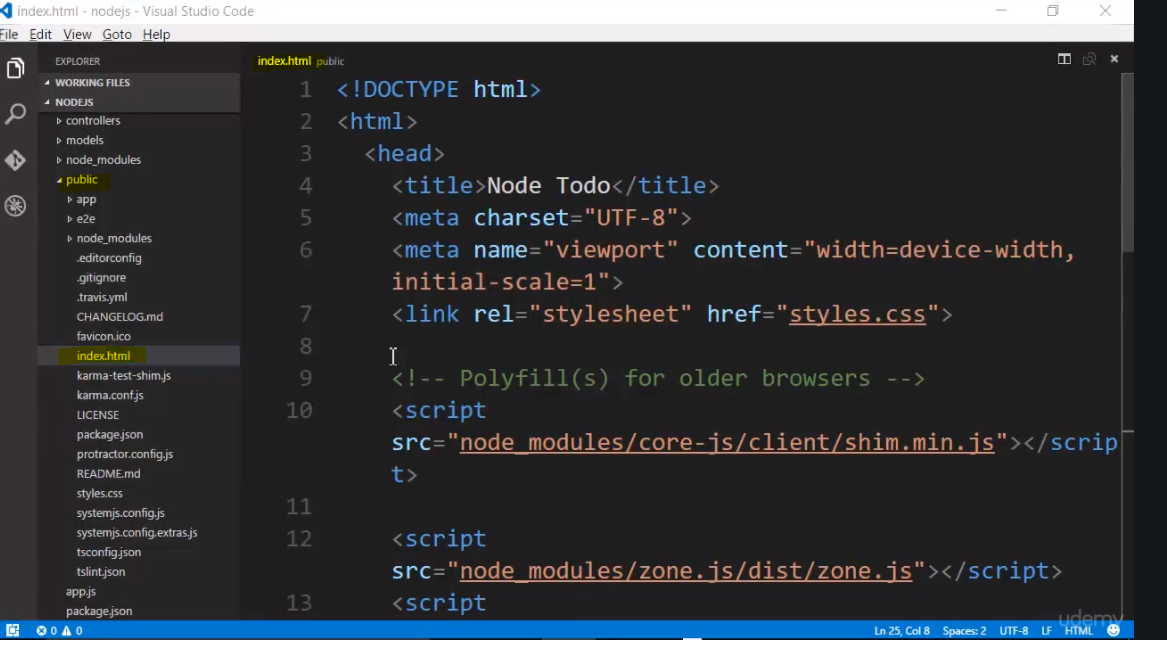
some server-side code against it.

But in this case, we're going to build a pure web frontend

that only uses the APIs, so it's fine for it to just be

an HTML file, it'll all be client-side JavaScript

that will be building everything.



.ts TypeScript is transpiled into JavaScript.



 some languages are purely transpiled languages.

That means the language you're writing in

is never actually run anywhere.

It doesn't really exist, essentially.

It's there to make it easier for you to write your programmes

and to maybe write some tools

to help you write your programmes,

and then that's converted into something else

that there is software to actually make it run.

TypeScript is one of those languages.

AngularJS 2 applications are most easily written

when you use TypeScript, and so Google,

on its documentation for AngularJS 2, primarily shows you

how to write code using TypeScript,

although it also shows examples, and you can write code

using pure JavaScript, it's much more cumbersome.

TypeScript is an entirely different thing to learn, however.

It's from Microsoft, it's a purely transpiled language.

You can go to typescriptlang.org to learn more.

TypeScript is an entirely different thing to learn, however.

It's from Microsoft, it's a purely transpiled language.

You can go to typescriptlang.org to learn more.

For example, you can go to a playground

which actually shows you the output of the transpilation,

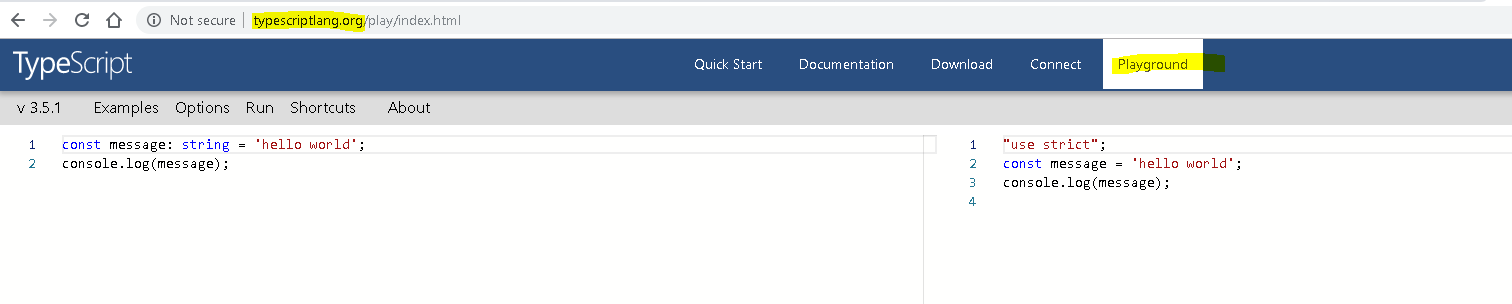
meaning here's code on the left that you write in TypeScript

and here's the code in JavaScript that will actually

be generated by the code you write in TypeScript

and this JavaScript code on the right will be

what's actually run in the browser.



That means that when you're writing .ts files,

there's a method, there's software

that's running the transpiler

that every time you make a change to your .ts file,

it can detect it and generate the updated .js file,

the code that you'll actually deliver to the browser.

Now, why would you want to use TypeScript?

Why does Angular want us to use TypeScript?

Well, because it could add language features

that are missing in JavaScript.

Now, I love JavaScript, it's a fantastic language.

However, there are some areas of other programming languages

that JavaScript lacks that can be good when dealing

with large teams or when dealing

with extremely complex structures,

and you need to have tighter control of your code.

TypeScript provides some of those things

as well as provides some features that are coming

in the next version of JavaScript, ES6 or ES 2015.

So that way, we can use what's coming now

when we're typing our code, when we're writing it,

even though it's not actually in the code

that's actually running in the browser.

**Please see complete example code**

So our AngularJS 2 frontend, which is completely separate

from our Node.js code that we're writing,

the only thing that we're doing is letting Node.js

serve the pages, deliver this JavaScript and HTML code

when we request it from the public folder.

But then, that frontend is actually calling

the Node.js code that's then running on the server

to actually deliver the data.

So I have a frontend and a backend,

the client side and the server side.

They're completely separate, but they're both

utilising features of Node.js and npm.

