# **The MEAN Stack**

## **MongoDB , Express, AngularJS and NodeJS**

MongoDB, Express, AngularJS, and NodeJS. These are the four technologies that make up the **MEAN stack.**



What is the stack of technologies? And **we say stack because one builds upon another. And they all work together in order to actually build the software.** So that's all a stack really means

**stack is often optional. There are pieces that can be interchanged, you might use a different database system for one piece of software, and then another database system for another.**

**So a stack is not, you must follow this set of technologies, but simply which technologies did you choose to use in order to build your software.**

**And in some cases, software technologies work so well together they become a very popular stack, a very popular set of technologies to use to build software. And that's the case with the MEAN stack.**



**Do you see something consistent and similar across all four of these technologies**?

Well, sure. And the word is JavaScript.

NodeJS is JavaScript on the server.

Express is one of those pieces of the stack that you have to have Node along with it, it's simply meant to work with Node. But Express itself is also just JavaScript.

AngularJS is JavaScript as well. It's JavaScript running on the browser, but it's still the same programming language.

MongoDB isn't exactly JavaScript, it stores data in a format that JavaScript loves.

So these four technologies work together very naturally. And that's why they've become such a popular combination that they've been given a name, the MEAN stack.

And that doesn't mean you have to use all four of these. You could choose Node and Express and a different database, like MySQL or others. You don't have to use Angular on the client, you could be doing something simpler and simply be using jQuery, or using other technologies that are out there, like React or Ember, other client-side JavaScript technologies.

But Angular is extremely robust and has all the pieces you need to build the UI portion of a web application. And for that reason, it's extremely popular as well as being backed by Google.

So MongoDB, Express, Angular, and Node as a combination of technologies is a great combination because it works so well.

And you notice, of course, that the order of things was just made in order to spell "mean", not really the order that things would be on the stack.

The order of the stack would really be Mongo, where your data sits, then Node on top of Mongo, which is getting and sending the data to the database, then Express on top of Node, which is making it easier to build the APIs, and then Angular at the top, which is getting and receiving that data from those APIs.

But that's okay because we just want to more easily be able to express the four technologies we're talking about, and so we generally call it M-E-A-N, the MEAN stack.

Now this course is not about the MEAN stack. It's about Node, as we've seen. But since we're talking about Node and want to understand it, it's important to understand how it fits in the MEAN stack, or even other similar combinations, where you have JavaScript running on the client and JavaScript running on the server.

## **How Browsers(Client side) works**

we need to establish a couple of facts about web browsers in general, normally the client side of the client-server relationship, and

we're talking about the internet. Remember how we said that node is written in C++, and the V8 JavaScript engine, and most JavaScript engines in fact, are in written in C++?

Well, here's another thing to know,

**for the most part, browsers are written in C++. Google Chrome, Firefox, popular browsers are written in C++. And while only Google Chrome uses the V8 engine, in general, browsers also have JavaScript engines embedded in them and give those engines access to features beyond what the JavaScript specification says JavaScript should be able to do.**

Now that might sound a bit like what Node.js does,

and your right, it's the same idea,

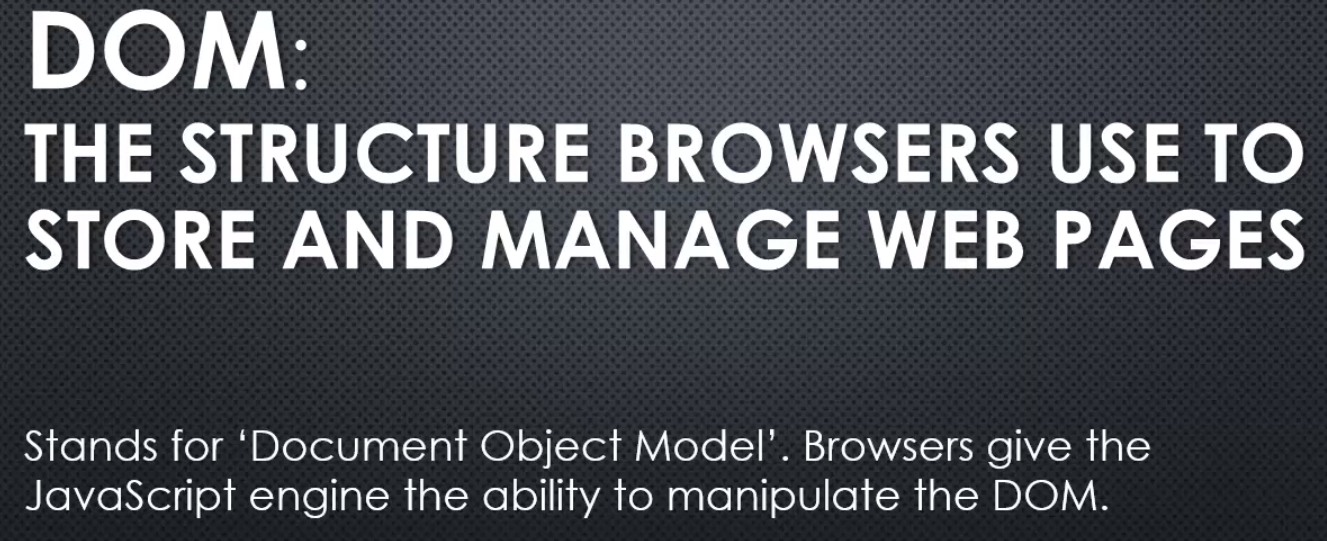
**a programme with the JavaScript engine embedded in it that does more than just what the JavaScript engine does.**



However, the browser is installed in the client, the computer of the person that's using that particular webpage. So the server just has node JS installed but each individual client will generally have a browser.

Now, what happens when the browser receives the text as a result of the HTTP request for a webpage? How does it break it down and know how to then translate that HTML into something that's visible, the webpage that you can see?

Well, that has to do with something called the **DOM.**



**The DOM, this is the structure that browsers use internally, generally a collection of C++ objects, to store and manage webpages.**

**DOM stand for the Document Object Model and browsers give the JavaScript engine the ability to manipulate the DOM, this is often a misunderstanding with web developers.**

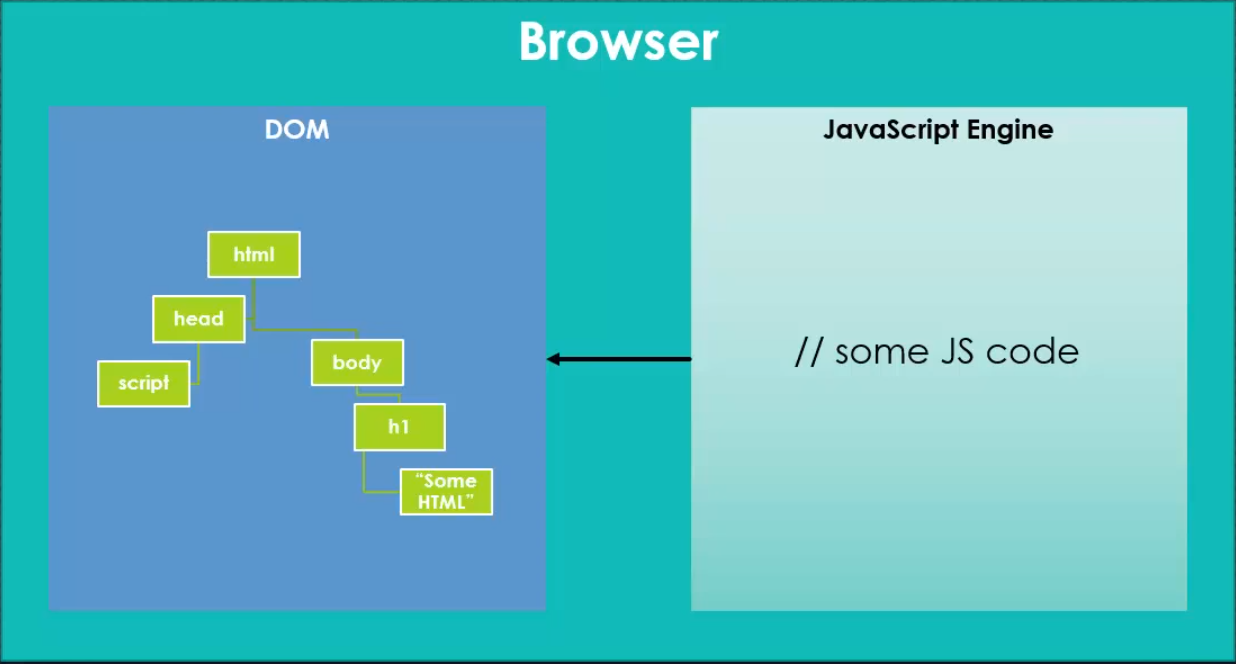
**The DOM doesn't live inside the JavaScript engine really. The browser uses the DOM to render the webpage and gives JavaScript access to the DOM, gives it features, so that it can manipulate the DOM under the hood.**

**And when the DOM is changed, then the browser will automatically regenerate, recreate, re-render, the webpage, and this is the basis for modern web applications.**

Well, **the browser is a programme sitting on the client's computer, and it receives some HTML as the result of an HTTP request. So the HTTP response body contains the HTML**, you've seen this, you've tried doing this in Node.js, using Express, and we're really just building a string and then sending it back as part of the HTTP response.

**But now the browser needs to take this string and decide what to do with it. So it actually doesn't just use this string all the time, it really just processes the string once and builds the DOM, the Document Object Model.**





**And browser takes that HTML and converts it into a hierarchy of objects, which represent the various HTML elements, and it becomes the DOM tree.**

**Now this idea really fits well with HTML itself**

**because HTML is really a sequence of elements with children and grandchildren etcetera, and sibling elements. So, HTML itself is structured in a tree-like fashion and so translates really well into a tree structure like the DOM.**

**And the browser then renders the webpage using the DOM, not using the string that it was given to build the DOM.**

**And then also embedded within the browser is a JavaScript engine, Now if it's Chrome, then V8 is embedded, the same engine as Node.js, but if you're using a different browser then it's a different engine, but the concept is still that it's executing the JavaScript.**

**Now in the browser's case, JavaScript will sit inside a script tag, or a script tag will be a reference to a .js file. In either way, the browser will pass the contents of that JavaScript to the engine and run it, and so it will receive it and execute it, and this again is happening inside the browser,**

we're not talking about Node.js here.

**And then, the JavaScript engine, if that JavaScript code contains some of the extra features to manipulate the DOM, then the code that the JavaScript engine runs will manipulate the DOM, perhaps adding elements, changing elements, removing elements, and when that occurs the browser will re-render the webpage, it will update, reflow, move around things, it will do things dynamically,**

**you click a button and something appears or disappears. That's because when you click that button, the browser caused some JavaScript code to run, to be processed through the engine, and that code contains some references to features that, while they don't exist in core JavaScript, the browser gave JavaScript access to, ultimately, most likely, some C++ code to run that manipulates the DOM.**

**So the JavaScript that was run though the engine actually caused some C++ code to run that manipulates the DOM, the DOM is then changed in some way, let's say an element is removed or added, and the browser automatically knows that it needs to update how the webpage appears. And so, things appear or disappear because that tree underneath that defines how things should appear, has changed.**

This is the core of how modern websites and web applications work, JavaScript code running in the browser, through the embedded engine, manipulating the DOM, making the webpage dynamic, allowing it to change, to respond, thereby making the webpage interactive.

**Now, the problem is that different web browsers can have slightly different JavaScript code that's expected in order to manipulate the DOM. So, if you're writing JavaScript code for the browser and you want it to work across many browsers and many version of browsers, you have to worry about all of the different ways you might have to write your code. You may have to do the same code multiple times, in different ways, so that it works across many browser and many versions, that gets really cumbersome. It's also really cumbersome when your application gets very large, manipulating the DOM, keeping track of what's going on, can result in a whole lot of extra code of what I like to call just plumbing code, code that's just there just to handle doing the work rather than thinking more about what the work is that you want to do.**

**And so frameworks, JavaScript frameworks like AngularJS, help us with that, they keep us from having to deal with code for each kind of browser, and they help us structure our applications and keep things manageable.**

**And really all AngularJS is, like other JavaScript frameworks, is just a bunch of JavaScript code that someone else wrote, much like Express, so, it's just a bunch of JavaScript code that we are taking advantage of.**

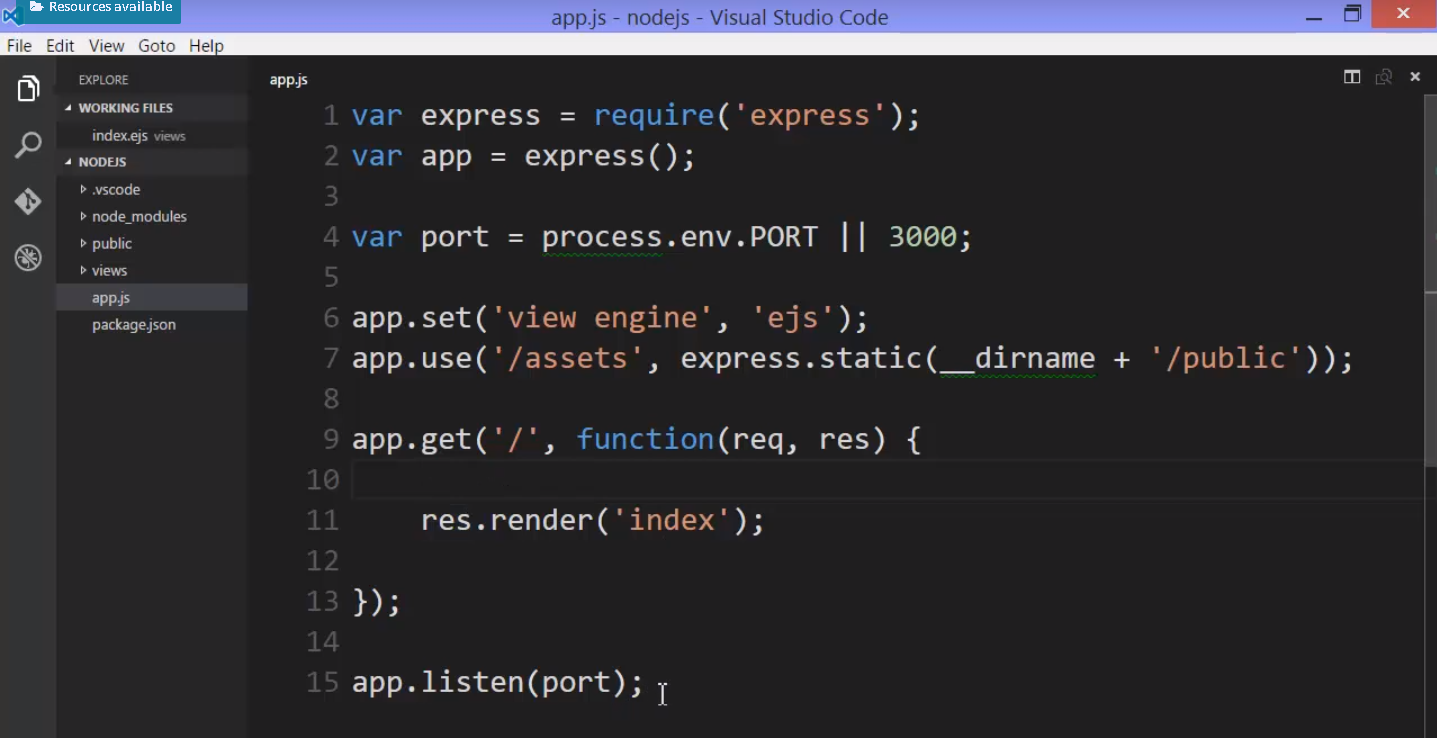
**The browser will load it in through the JavaScript engine and then we can write some more JavaScript code, taking advantage of all the work inside the JavaScript code of a framework like AngularJS.**

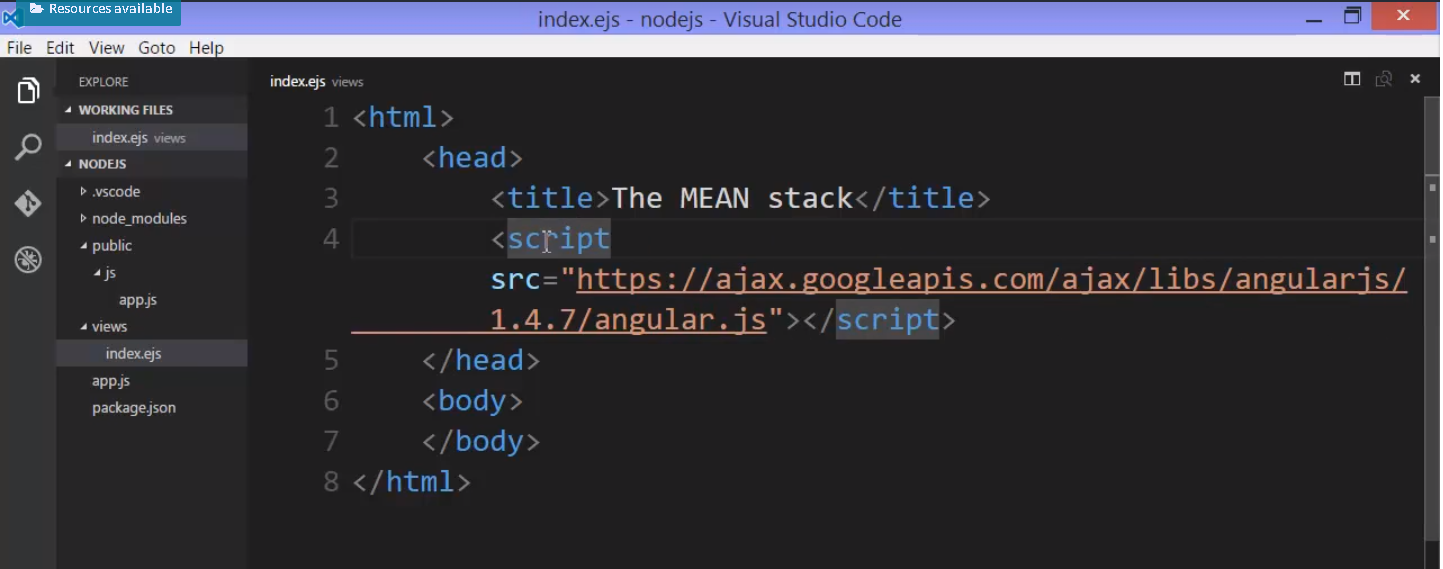
**However, we are going to look at AngularJS from the perspective of the MEAN stack, but also from the perspective of dealing With JavaScript code on the server and on the client.**

**This is the area where you can begin to get confused, you have JavaScript that's meant for the browser, you have JavaScript that's meant for the server, and they're all sitting inside this same application, so you can start to mix things up in your head and make some core mistakes.**

## **AngularJS : Managing the client**

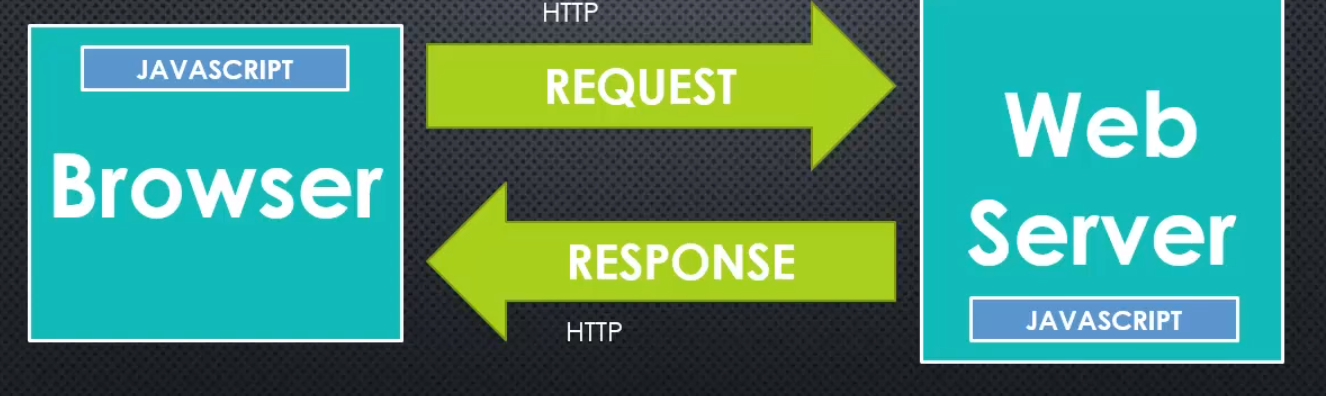
Lets use angular from CDN, or content delivery network. What that means is, I'm going to go out and grab Angular from a different server, and I'm going to go ahead and have my application go out and grab that. The browser will grab this googleapis.com URL.





And why is that good?

**Well, it actually helps my Node server. We won't have as many people requesting and downloading this file from our server. They'll go out and download it from Google. That's also helpful for the speed of the application, because CDNs are designed to be extremely fast for this purpose**



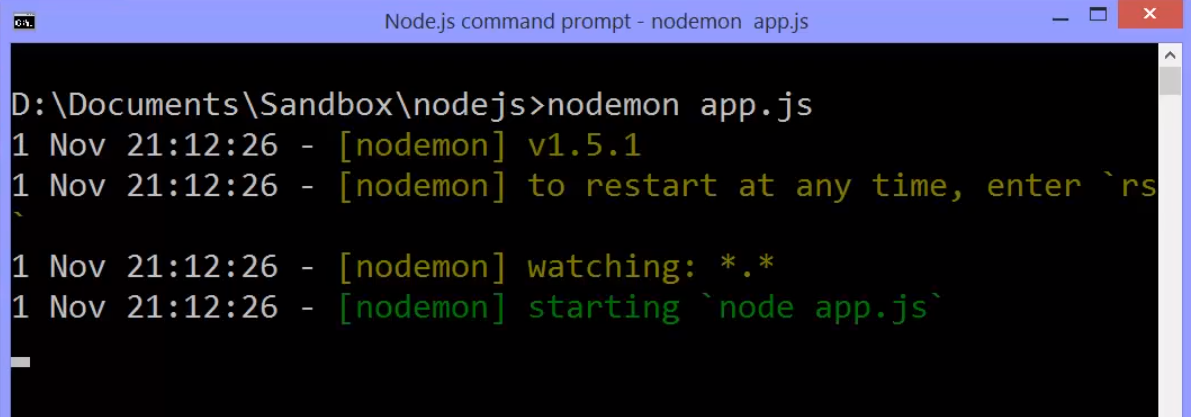
**we're dealing with the two sides of the client-server process, the browser and the web server.** The requests and responses via HTTP, as we've discussed, and the fact that

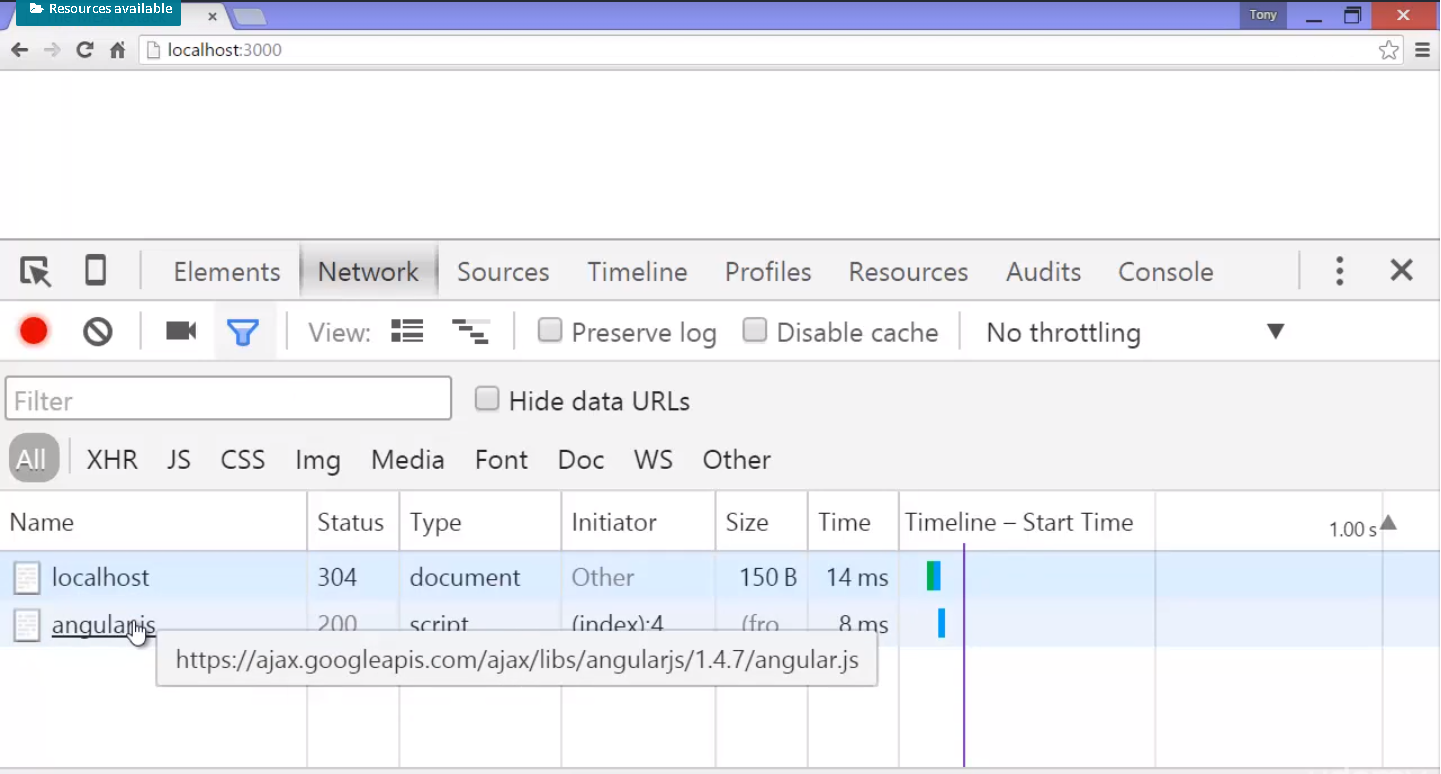
**JavaScript is running on both sides. When you begin dealing with something like the MEAN stack, or any other client-side JavaScript, when you're also writing JavaScripted Node, you can start to get confused between the two sides as you're coding. Now, some things will be more obvious, but some things won't.**

So, you need to keep things straight in your head.

**And so, we've actually requested the browser to download some JavaScript and run it with that script tag. And that script tag is sitting inside our EJS file. But it's important to realise that this JavaScript that's being downloaded in this angular.js file is not being downloaded by Node. It's not being processed by Node in any way.**

So, what's actually happening when we load this? Well, let's go ahead, I'll use nodemon, and run app.js,





**Two things were loaded, the page. So the HTTP request for the page was processed by Node. And here's the response.**

**Node simply provided the HTML string that the browser then interpreted. And when the browser loaded this HTML string, it saw this script tag, and the only thing that Node did was give us this text for the script tag. It returned that in the HTTP response. And then the browser saw that it needed to go out and download this JavaScript file. And so, we see that second download.**

 Angular is just JavaScript. So we've loaded a script file and the JavaScript engine sitting inside the browser has processed that JavaScript. At this point then,

**Angular the JavaScript inside the Angular file is going to deal with the things that the browser makes available to the JavaScript engine. Things to manipulate the DOM, that tree structure that the HTML is processed into.**

So for starters, what Angular is going to do is help us manage updating and manipulating the DOM.

To do that we're going to use the first of what's called a **directive. A directive is generally either an attribute or a value of an attribute that is sitting on an HTML element. Now when this HTML is processed, these directives, these special attributes, cause Angular to do work.**

Because in JavaScript inside the browser

you can go and interrogate, look at,

attributes and values of attributes on HTML elements.

So let's call this TestApp.

And this ng-app

means that this will be the name of my Angular app.

So when I find code in my Angular app

that references this name,

we'll know we're looking at this slice of HTML.

I'll do something similar called ng-controller.

And I'll say MainController as

and we'll just say vm.

What this means is we can actually control

different parts within the HTML of the app.

So this controller means that the code inside

this controller in the JavaScript,

will manage what goes on between these two, or we might say the children

of this particular HTML element.

So in this case the body element.

et's add in our own Angular code

that we're now going to need.

Our code that will use the objects and the features

of Angular made available in this angular.js file.

To do that we need to reference our own file.

So I'll say script source equals, and I'll say assets,

remember we decided that the public folder

would be used

for the static content

and that we would actually respond to HTTP requests

for /assets as things inside the public folder.

So I'll say /assets/js/app.js.

Now again what's going on here?

This app.js file in this public folder

is not processed by node.

Because it's inside the static content directory.

Node is simply going to deliver the contents of this file,

when the browser requests it via an HTTP request.

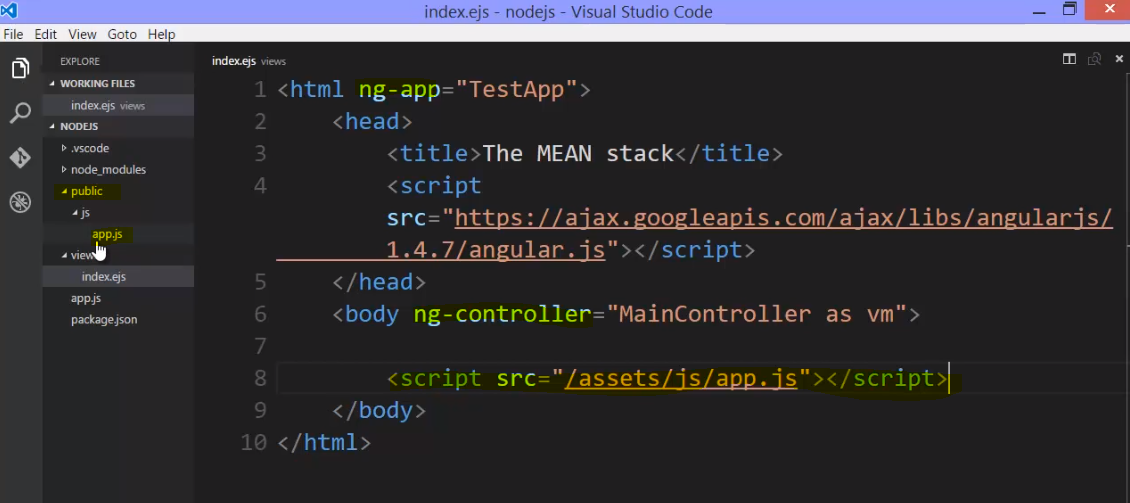
The node won't process, it won't run this JavaScript at all.

So this script tag will then cause the browser

to make that request and the browser,

the JavaScript engine inside each particular user's browser

will do the processing.



by the way, in our index page

we loaded the angular.js script tag in the head

and this other script tag in the body.

You don't have to do it quite this way,

but I like to do things this way because

I want Angular to be completely loaded

before the browser renders,

or outputs the visual things

that we see on the screen.

And these scripts in the head tag

will be processed before that happens.

in my app.js file,

because my angular.js file was loaded already,

I now have a variable, an object called angular.

And we'll use this a lot.

It provides all the functionality that I need.

I can say angular.module and I'll give it the same name

that I put in ng-app.



So what this does is it adds a controller of this name,

to this module, to this app,

and executes this function.

And I'll just call my controller function here.

I'll create it and it will be executed,

it'll be called here by Angular itself.

And then I'm going to say this.message hello.

So what I've done is I've added a property

to the this variable.

Which because I said as vm,

what Angular actually did

was make the this variable point

to the same object that this variable here,

which I call vm or view model,

that is the model or the data and properties methods

that I want for this particular view,

or that is to say this particular screen,

is going to be made available

because it's on the same object.

So Angular then will attach message

and I can actually access message inside my HTML.

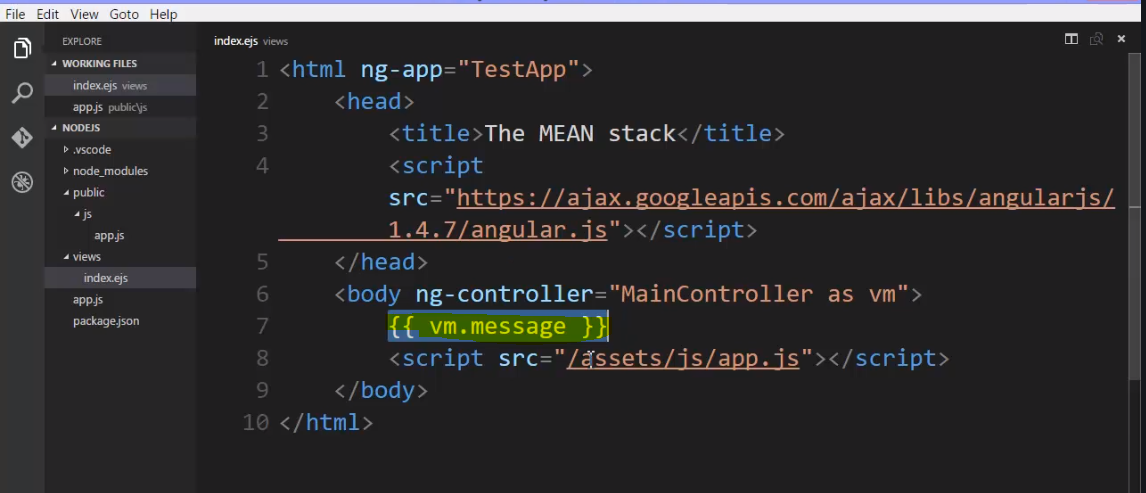
I can do that with, for example,

a double set of curly braces.

Two curly braces and then two ending curly braces,

and I can say vm.message.

So what's happening here?



Well, this string is what will be given

from node to the browser.

But once the browser has this HTML

and processes all this JavaScript,

it's going to see these attributes in the DOM,

it's going to find those same things

in the JavaScript code

where Angular is expecting them to be

in the way Angular is expecting them to be set up.

And then it's going to see this property on the object,

and it's going to see these curly braces in the DOM,

and it will know that means replace this

with the value

that is actually on the variable.

Now again this is kind of like what EJS was doing

when it was replacing values,

but that was really just replacing strings.

Angular is doing something similar

but it's using JavaScript's ability to update the DOM,

the DOM tree, not the string of HTML that was downloaded,

but the DOM tree that was generated

when stored by the browser after the fact.

So this will be in the DOM tree,

the screen will want to render this

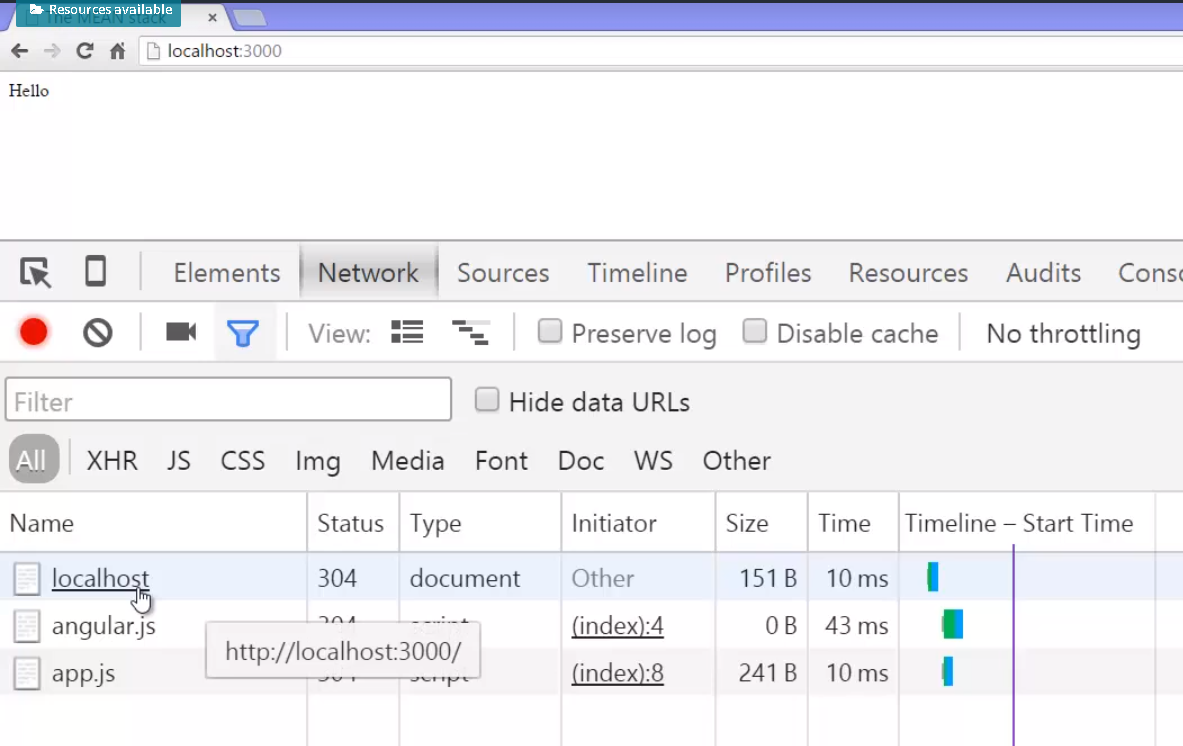
and Angular will actually change what's in the DOM

and the screen will render instead

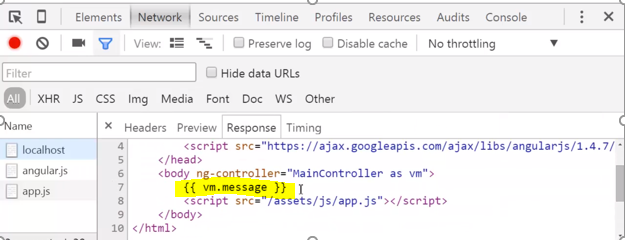
what we're asking it to

via the Angular code and my code.

**When I run/refresh this**

Output – 

 go to localhost. What was downloaded was vm.message with these curly braces.



**That will pulled into the DOM, the JavaScript was run and  then when the JavaScript ran in the browser, it replaced that with Hello. At that point, that had nothing to do with node.**

**But everything to do with Angular. Angular makes it very easy for us to manipulate the DOM.**



Okay, just recap what’s happening here-

**node.js delivered this entire HTML page, AngularJS was then loaded by the browser in its JavaScript engine. We attached this ng-app attribute which Angular looks for, the JavaScript code that is Angular looks for, and we gave it this name which matched the module that we're working with. And so that meant that that module and the code inside of it are going to be managing all of this HTML. Then ng.controller has a name, that was the name of the controller inside that module and it will execute this function that we give it which then makes things available to use inside this particular HTML element. And we used double curly braces in order to output a value. We used the this variable inside the function to attach a property and then we used the as, what's called controller as AngularJS to give ourselves a variable name that matched up and is the same object that the this variable was pointing to, and we were able to output our value because when the AngularJS code ran it saw this in the DOM, these double curly braces, and then replaced that value in the DOM,with the value here stored in the JavaScript.**

## **Angular1 , Angular2, React and more…**

**When we talk about Angular, when we talk about React, we're talking generally about JavaScript that's running in the browser**

When we talk about client side javascript framework**. There's something very important to remember. It's all just JavaScript. Angular is just JavaScript. React is just JavaScript. Angular 2 is just JavaScript. BackBone, Ember, all of these frameworks that people talk about and use, they're not adding anything to the JavaScript Engine. They're not adding anything to the browser. They're just JavaScript code that other people have written for you, and they're just using what the browser and JavaScript Engine make available to JavaScript.**

**This is important because we're talking about the Mean Stack, which is a very popular stack, but it's not the only stack you have to use. It's not like for some reason Angular works best with Node. Angular is just JavaScript running in the browser. So you could use any other JavaScript framework or not even use the JavaScript framework. Just write JavaScript in the browser yourself, and that's just fine.**

**Node.js has a bunch of JavaScript code that's written for you, and C++ code that gives you more features to the language. Angular, React, these others designed for the client for the front end don't do that. They're just JavaScript code written so that you don't have to write as much.**

## **Very Important:Being a full stack developer**



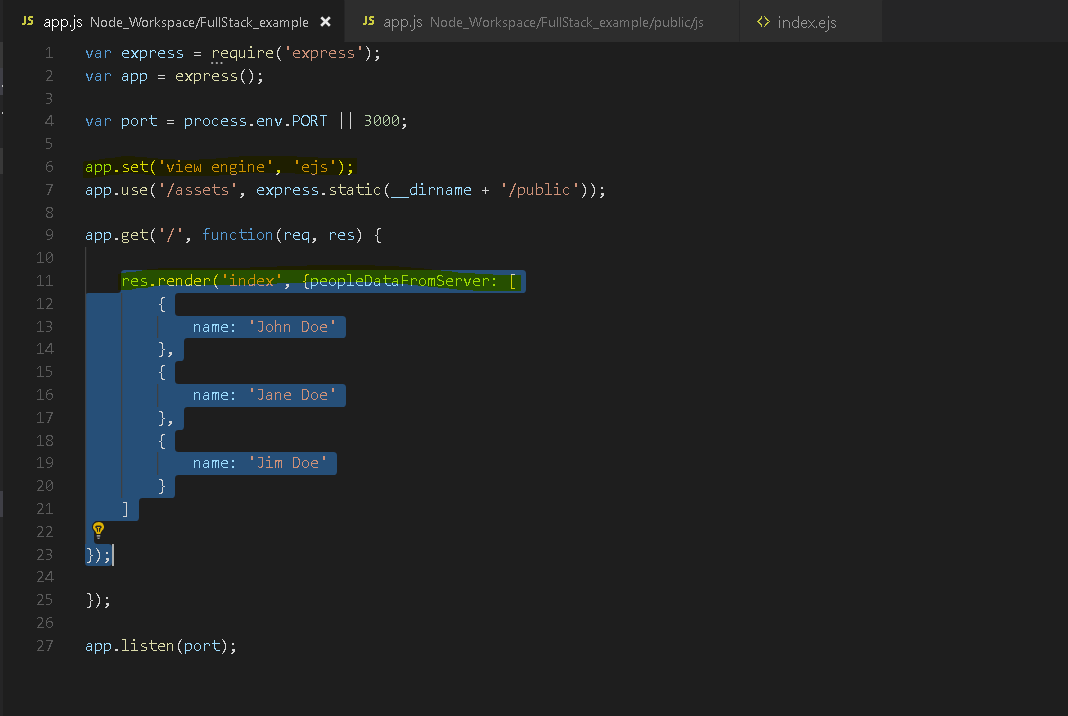
A full stack developer. That basically means a developer who knows all the pieces of a particular software stack. And thus, if they really had to, could build an entire piece of software by themselves.

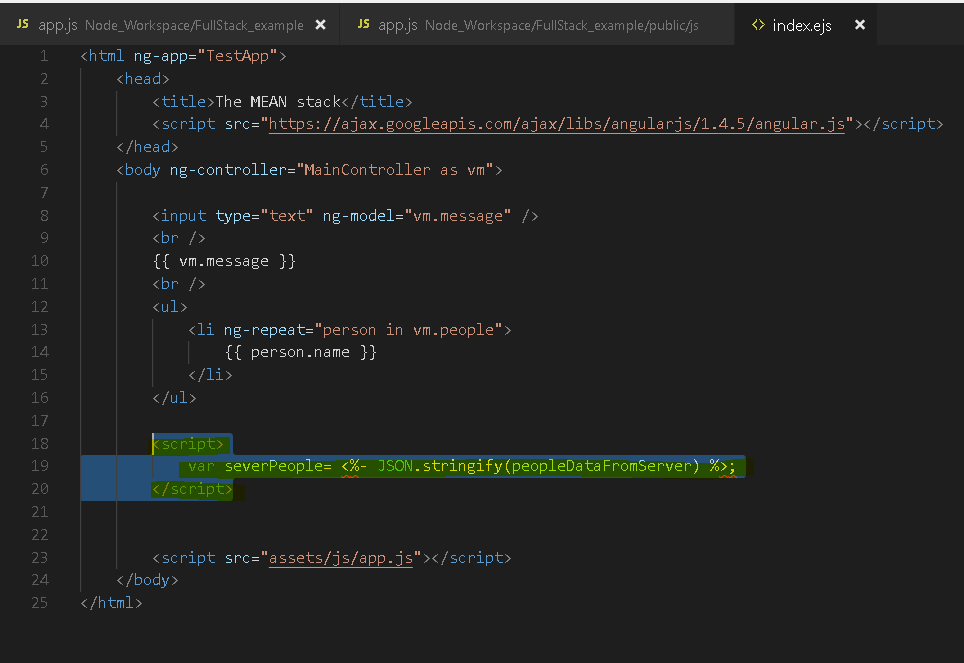
So for example, with the MEAN stack, a full stack developer would mean someone that knows MongoDB, knows Express.js, knows AngularJS, knows Node.js.

Generally speaking, it's someone that knows the client, the server and the database side of software, and perhaps as well, design in UX. But either way, being a full stack developer means that you are a more useful developer to a software company.

### **Very important example**

sending data from server to client by Template Engine







So lets understand what just happened?

When I loaded the page,

then Express and NodeJS responded to that

by rendering the dot EJS file

using this data and this was all server side, NodeJS side.

When indexed that EJS was then rendered.

What was the dynamic piece in here

that was actually updated by EJS?

Only this.

It took serverPeople and converted

that to JSON and replaced this area

here with the result of JSON dot stringify.

In other words, a JSON string.

And then return that to the browser.

That was the HTTP response.

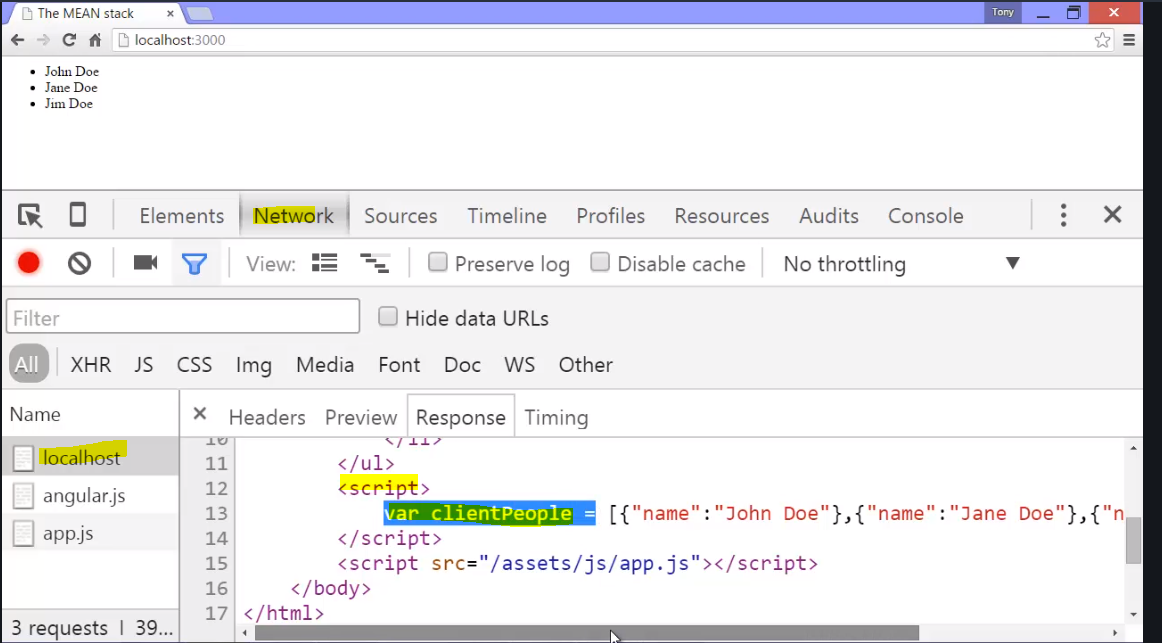
So if we go over to the browser

and I just refreshed this.

Go to the network tab.

Click on localhost to see what we got back.

And what we got back, let's look at this.



See this script tag?

There it is.

Var clientPeople equals, that wasn't

adjusted by EJS or processed because

that's just part of the page,

but the part that I did tell the template

engine to process, it did.

It ran that JSON dot stringify on the server.

And so I ended up with some JSON

that was directly placed inside

the JavaScript code that was then run

on the client, on this browser.

And so clientPeople, this variable

ended up with this data inside of it.

And then my normal AngularJS code ran.

Now let's think about this.

I outputted data directly from the server

onto the client via the template engine.

Now, there are some circumstances

where you would might wanna do this