Section 2: Introduction to Node.js and NPM

22 May 2025 19:43

Now before diving into some NodeJS code, let's do a high-level overview of what NodeJS actually is, what we use it for, and why we use Node instead of other technologies.

So, the official definition is that NodeJS is a JavaScript runtime built on Google's open-source V8 JavaScript engine.
Now, what does that actually mean?

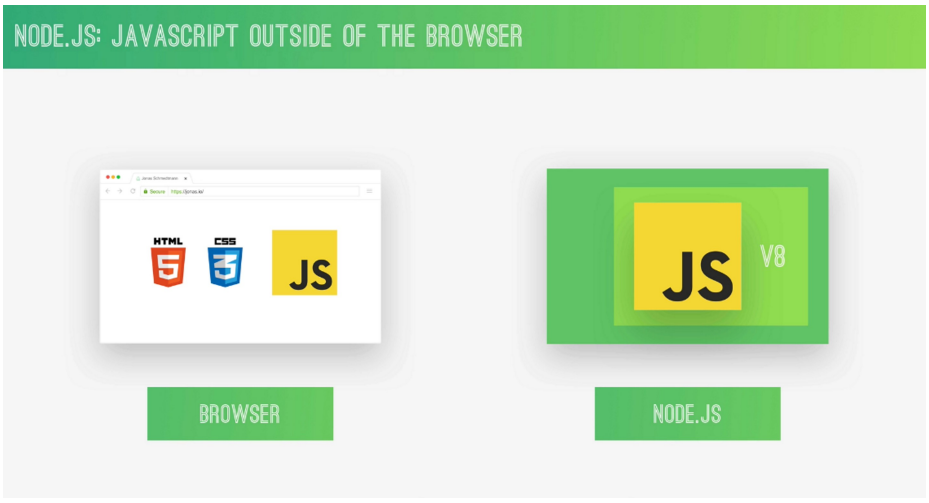
Well, let's start by trying to understand what the JavaScript runtime and the V8 engine actually are.

So, you have probably already used JavaScript before, and it was probably always just inside a browser, right?
Because any browser natively understands HTML, CSS, and JavaScript. And no matter if you write vanilla JavaScript or use a framework like React or Angular, that's all just JavaScript that gets executed right in the browser.

So, in this case, the browser is the JavaScript runtime.
But what if we could take JavaScript out of the browser and simply execute our JavaScript code somewhere else, without all the restrictions that we have in the browser?
Well, it turns out that we actually can.

And the solution for this, as you can guess, is called NodeJS.
NodeJS is just another JavaScript runtime. It's like a container—an environment in which a program written in JavaScript can be executed, but outside of any browser whatsoever.
All right, it's actually a bit more complex than this, of course, but for now, this is enough.

Now, who actually does execute the code if not the browser?
That's where the V8 engine developed by Google comes into play.
Because that is exactly where JavaScript code will be parsed and run in NodeJS, okay?
So, I hope that now the definition of NodeJS being a JavaScript runtime on the V8 JavaScript engine makes a bit more sense.



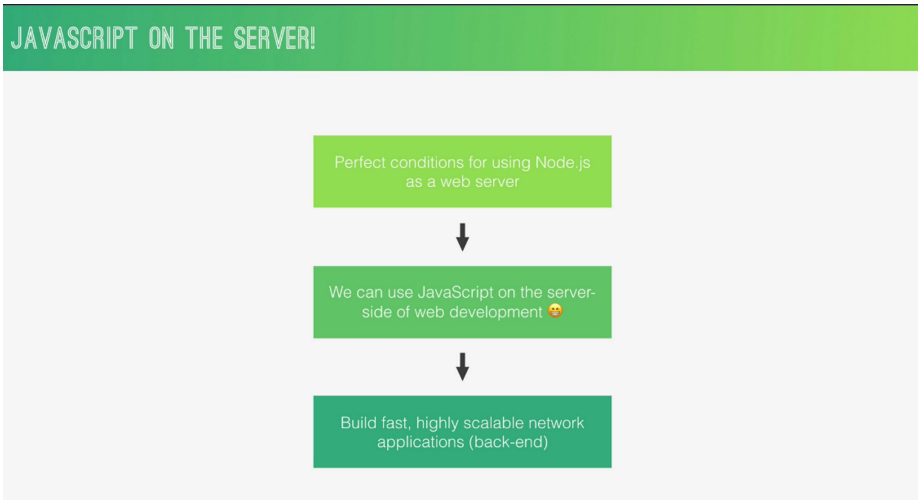
We are gonna go into a lot of detail about how Node really works behind the scenes right in the next section.

Anyway, now that we have JavaScript outside of the browser in a kind of standalone environment, which is NodeJS, we can do so many more things with JavaScript that were completely impossible before.

Like accessing the file system or having better networking capabilities—these are now possible with NodeJS.
All these factors together give us the perfect conditions for using NodeJS as a web server.

This means we can finally use JavaScript on the server side of web development to build fast, highly scalable network applications that power the back-end for websites or web apps.
And this is absolutely fantastic—and honestly, game-changing—for web development.

So, let's now take a look at some use cases for Node and why it's such a great fit for back-end development.



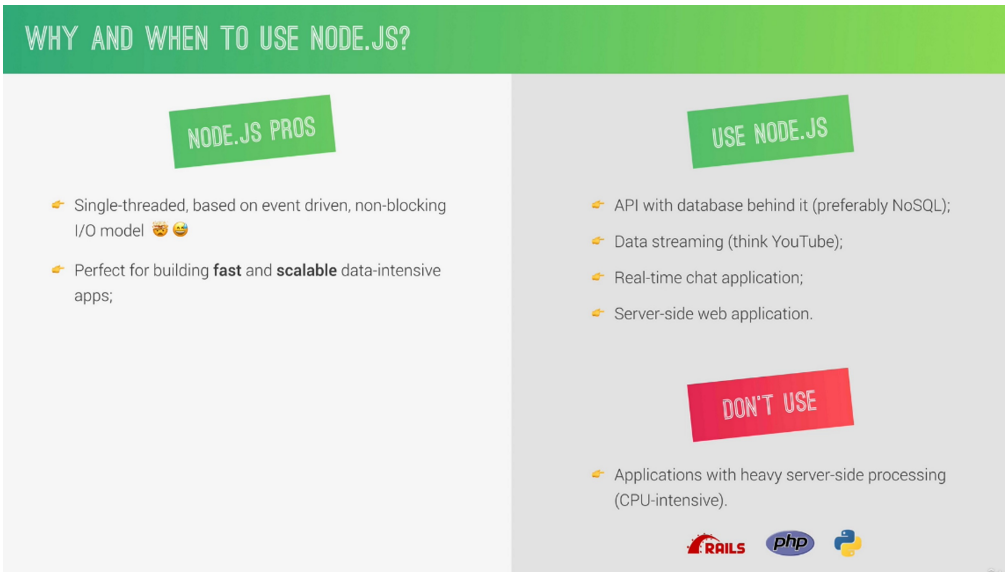
And the first thing we need to talk about is the fact that Node applications are so fast and so scalable because NodeJS is single-threaded, based on an event-driven, non-blocking I/O model—which makes NodeJS very lightweight and efficient.
Now, that of course sounds super complicated, I know. But don't worry—we're gonna break down exactly what all of this means a little later in the course.

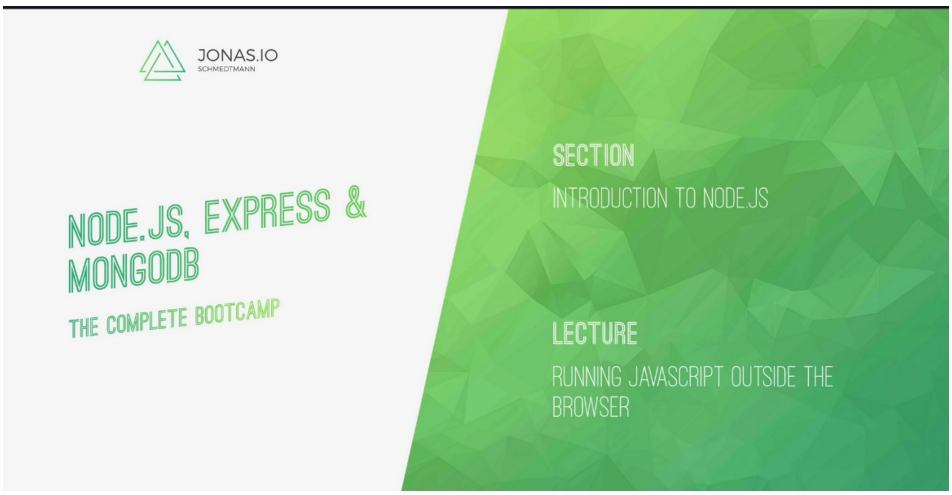
For now, just keep in mind that Node is perfect for building super fast and scalable data-intensive web applications.
That makes NodeJS a perfect fit for building all kinds of applications—like building an API with a database behind it, preferably a non-relational NoSQL database like MongoDB.
And this is actually *exactly* what we're gonna do later in the course as we dive deeper and deeper into

NodeJS.

But there are, of course, all sorts of other apps we can build—like data streaming apps (think YouTube or Netflix), real-time chat apps, or even server-side web applications where the entire content is generated right on the server.
So as you can see, the possibilities are kinda endless.

But there's actually also a type of app that we *shouldn't* build with Node.
And that's when your app needs some super heavy server-side processing—like image manipulation, video conversion, file compression, or anything like that.
You'll understand why that's a problem once we dive deeper into how Node actually works.

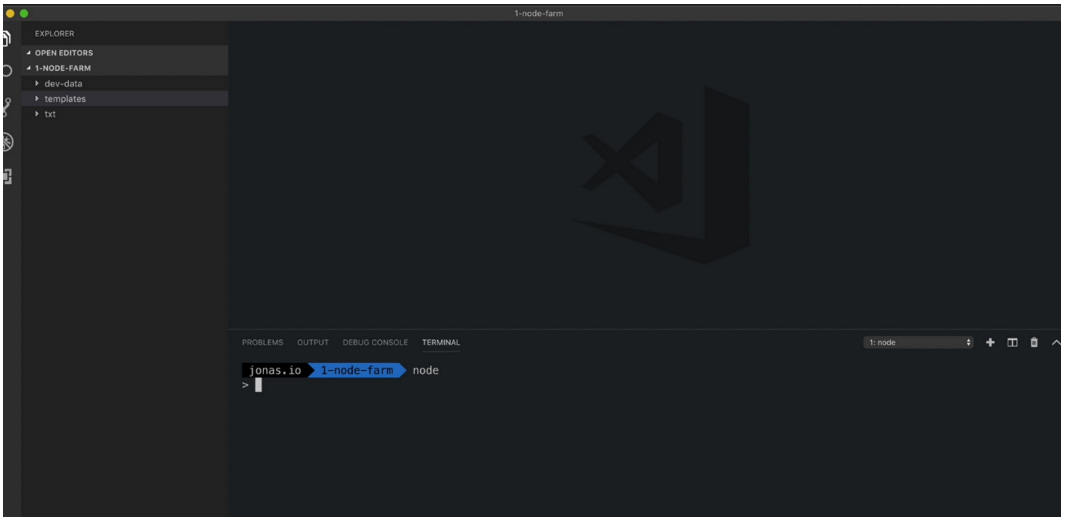




Running JavaScript Outside the Browser

Open up starter project. To start writing some node code here in the console, type

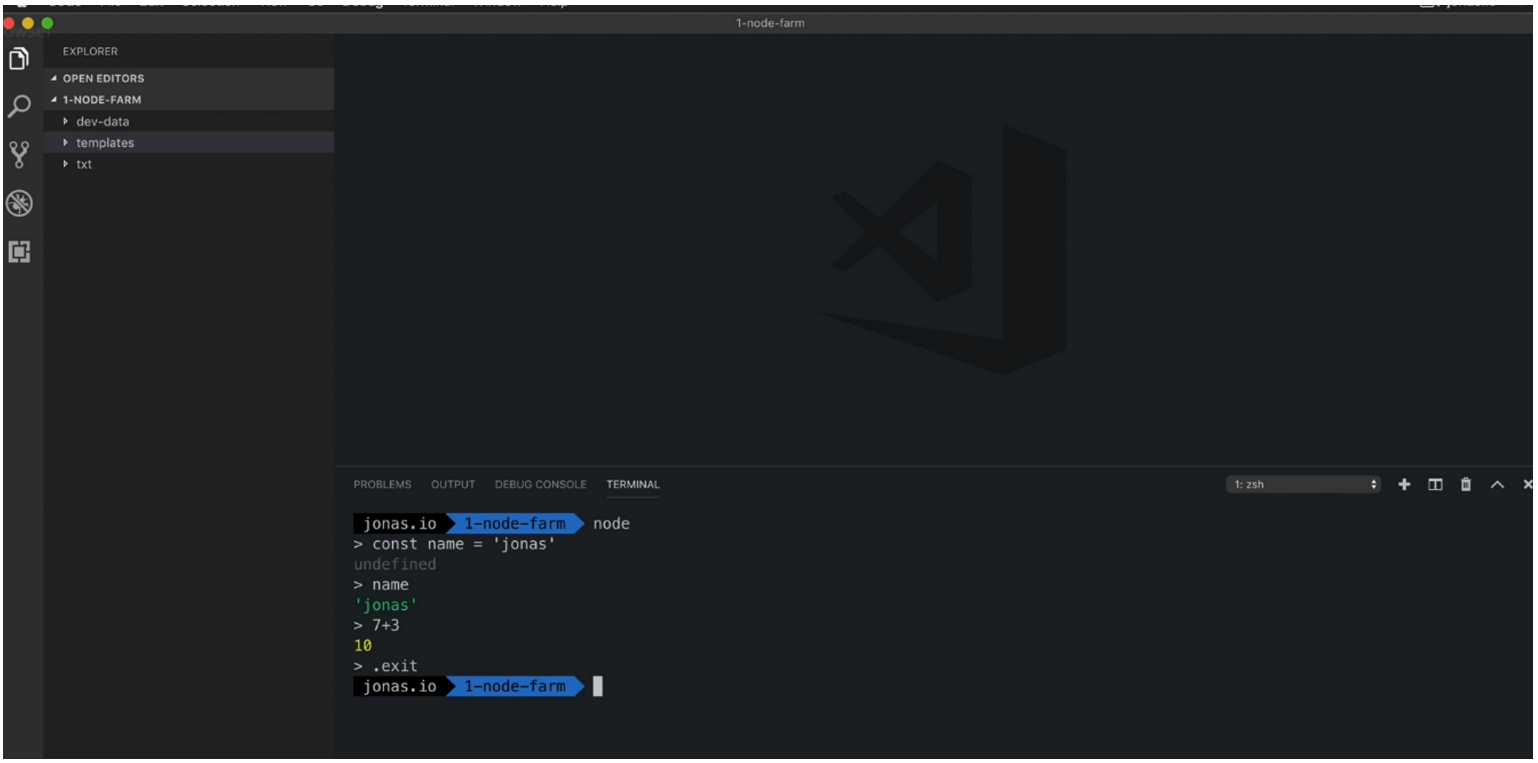
"node"



Which will then open up "Node REPL". Repl means 'Read Eval Print Loop".

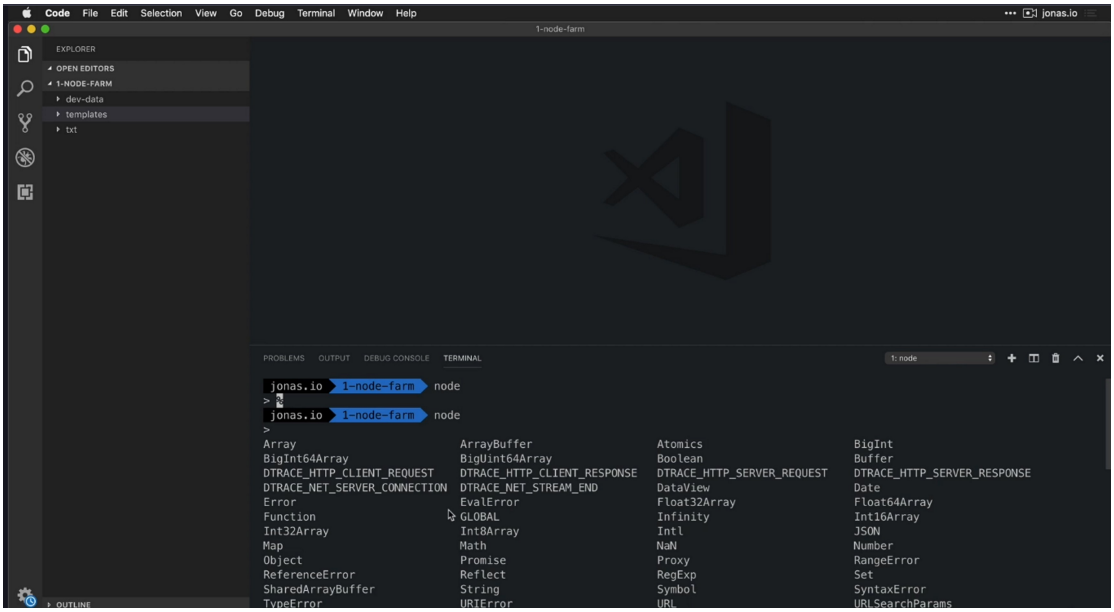
Which means we can now write javascript code in the terminal.

Any javascript will work here. Because at the end of the day, NODE JS is a JavaScript runtime.

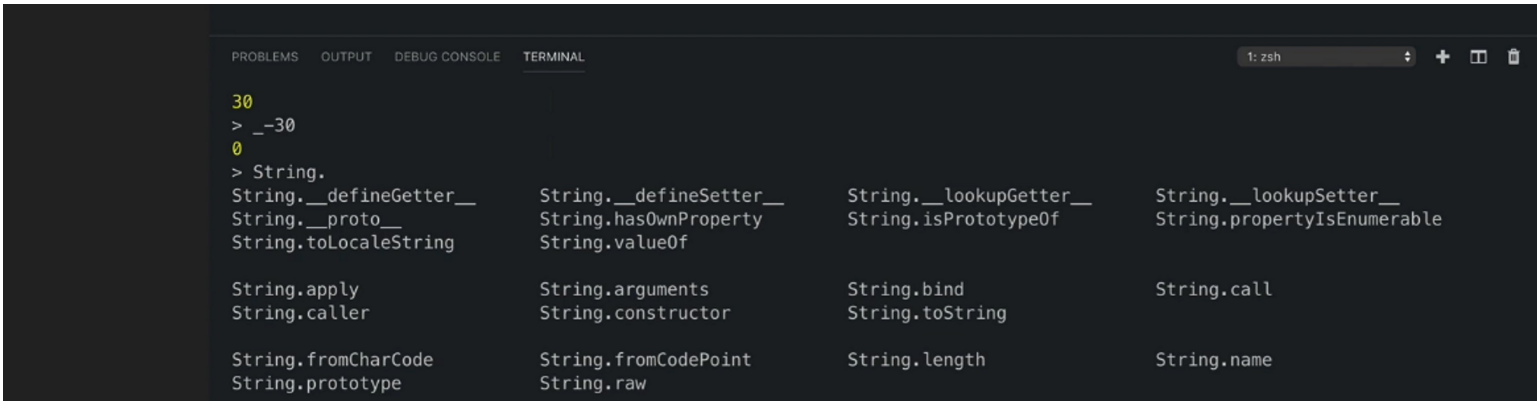


We use .exit to exit
We can also use CTRL +D

If we hit tab right now, you can see all the global variable that are available in Node.

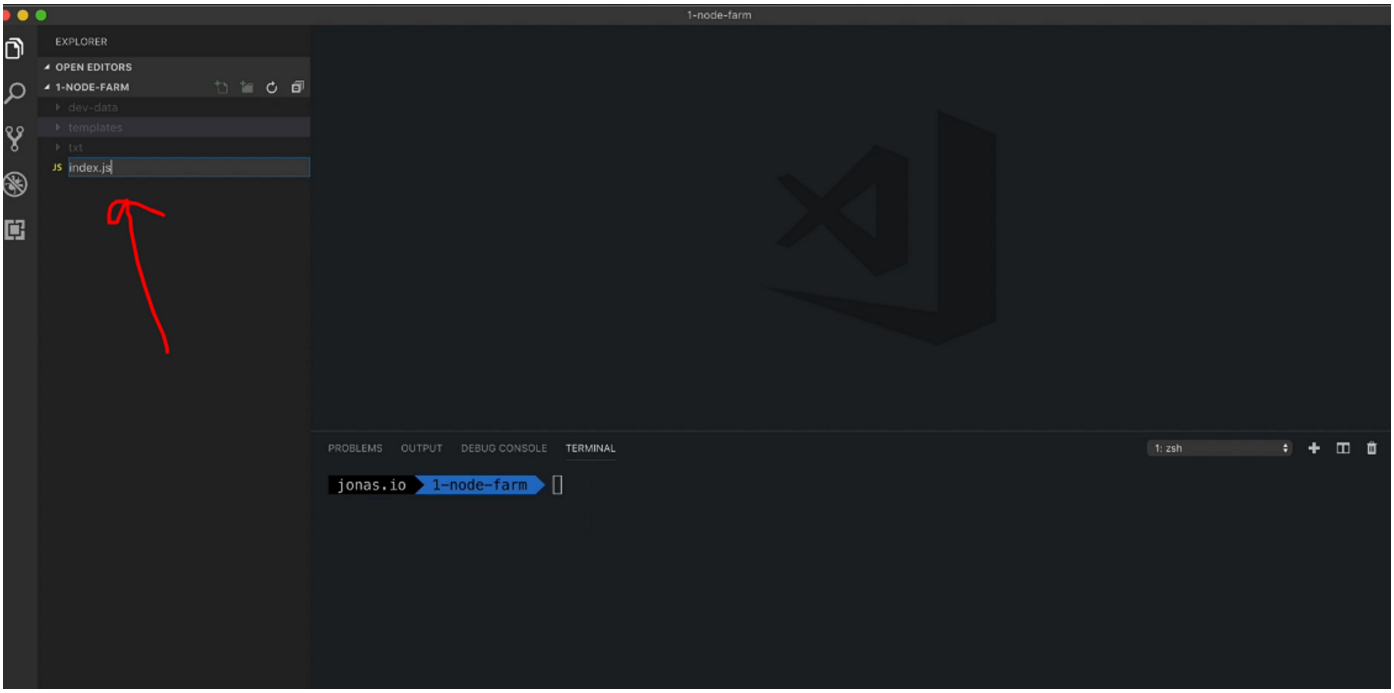


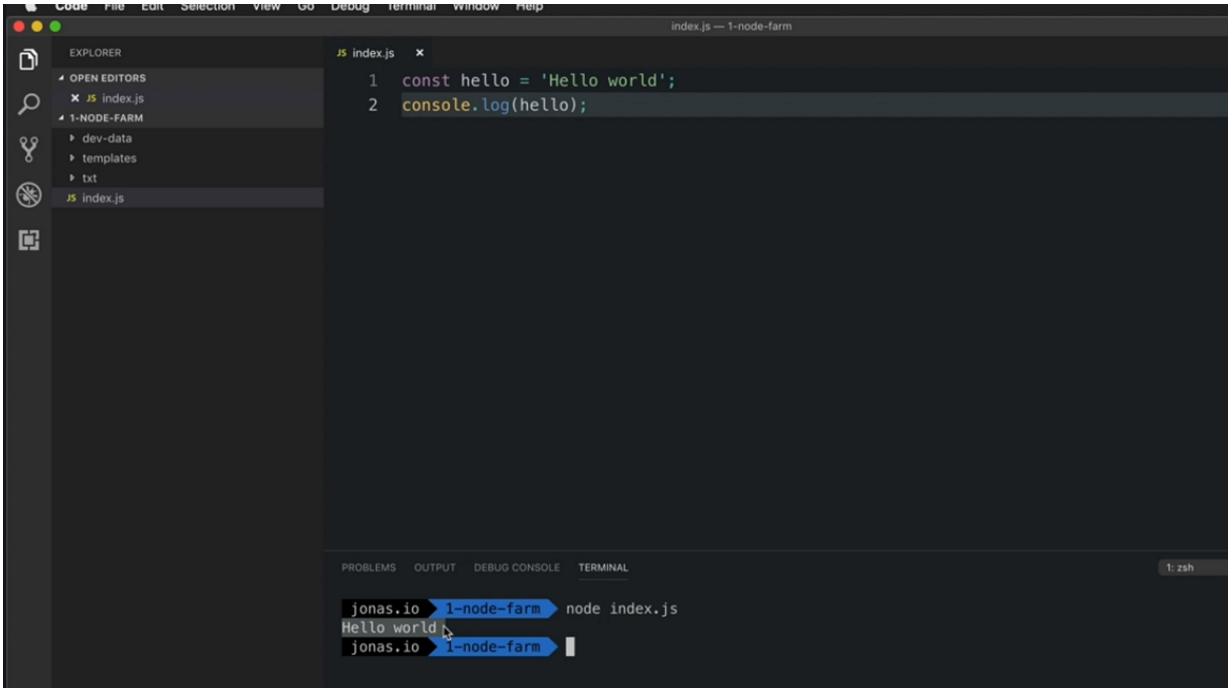
_ => underscore is basically your previous result.



CTRL+K to clear the console.

Let's create a javascript file.



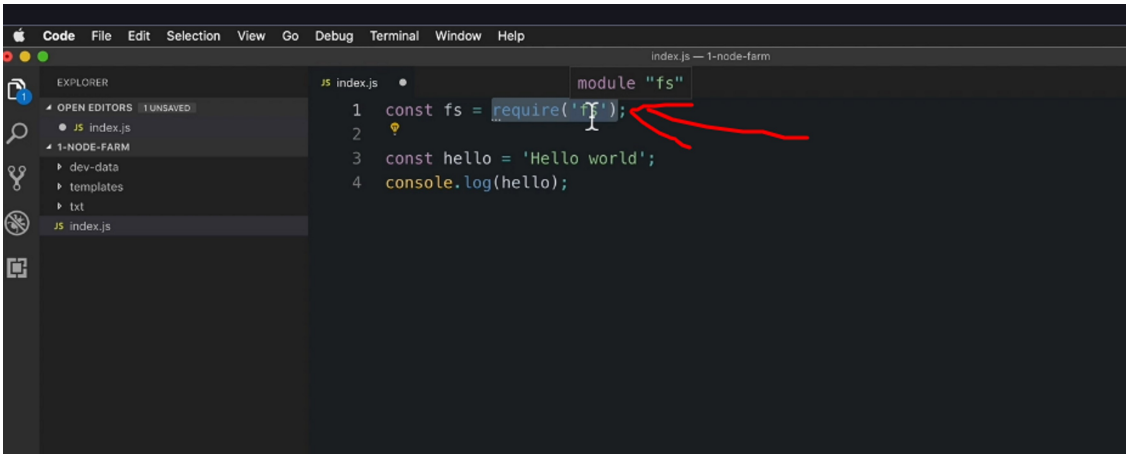


With Node Js we can do some amazing stuff which is not possible on the browser.

Like reading file from the file system. In order to do that, we need to use a Node module.
So Node.js is really built around this concept of modules where all kinds of additional functionality are stored in a module.

And in this case for reading files, that is inside the FS module.

Here FS stands for file system. By using this module (FS) here, we will get access to functions for reading data and writing data to file system.

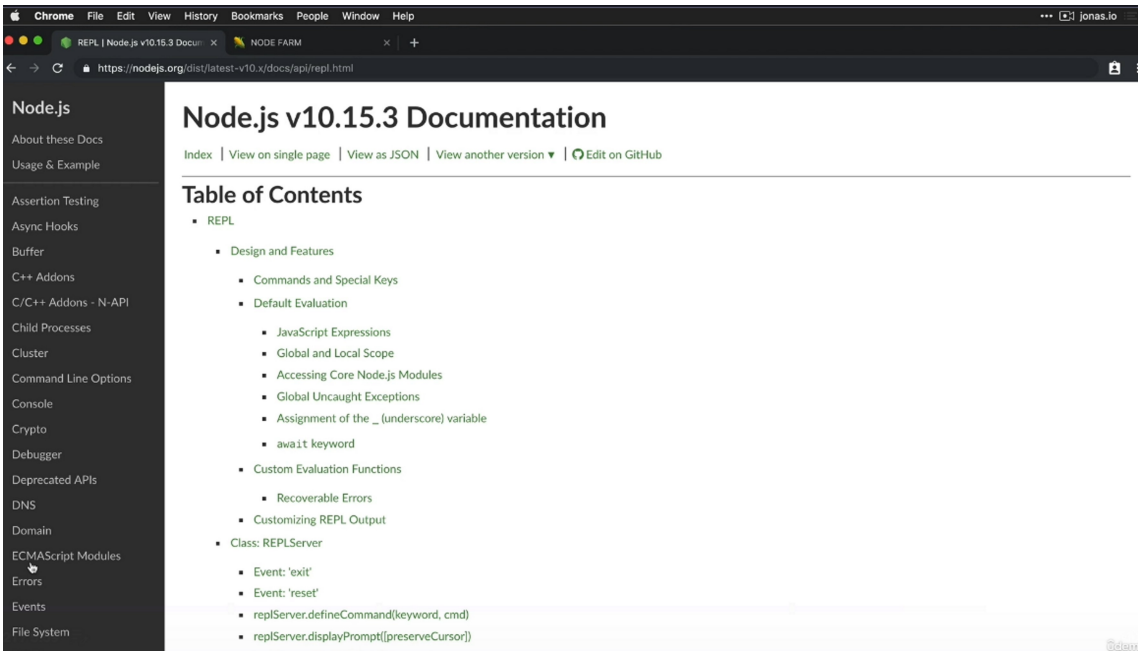


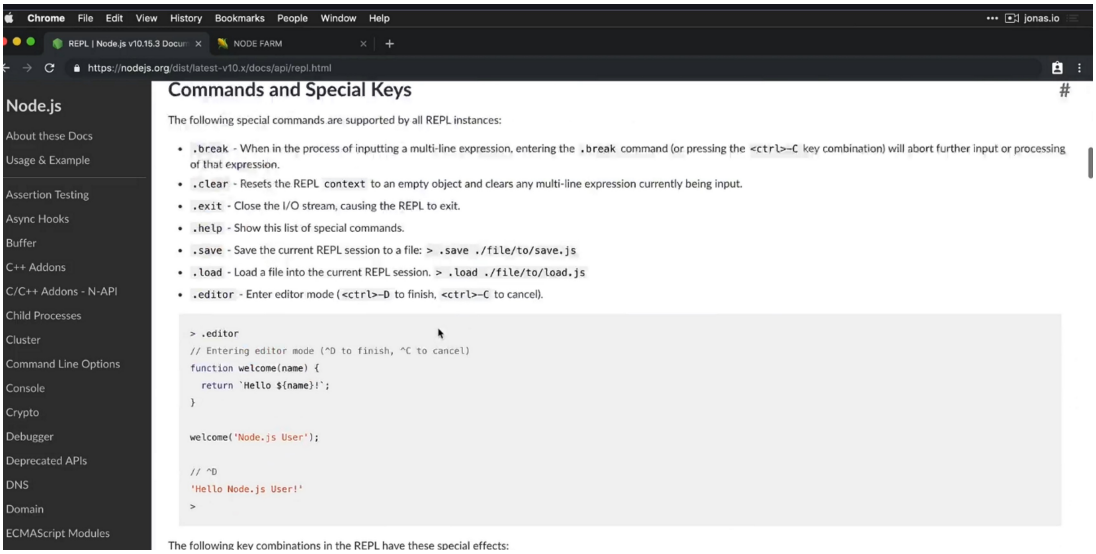
So calling this function here will then return an object in which there are lots of functions that we can use.

In case you need some other module, you always know look for Node documentation.

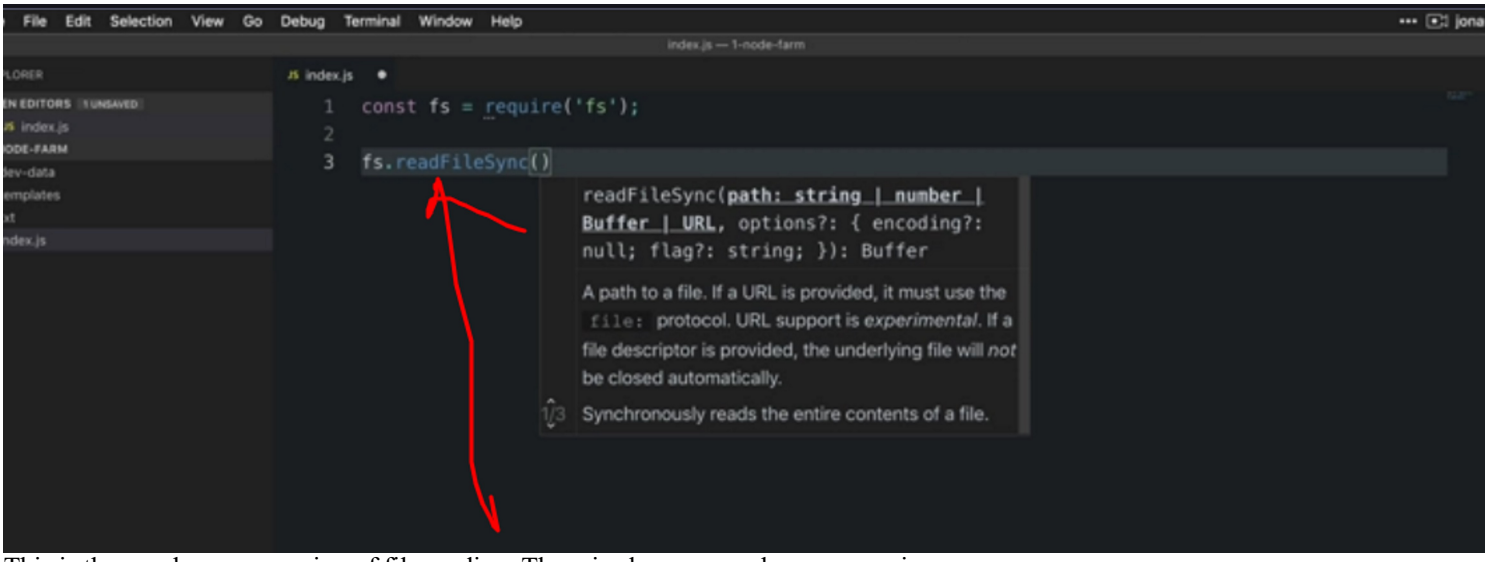
nodejs.org

All the modules are listed on the left side



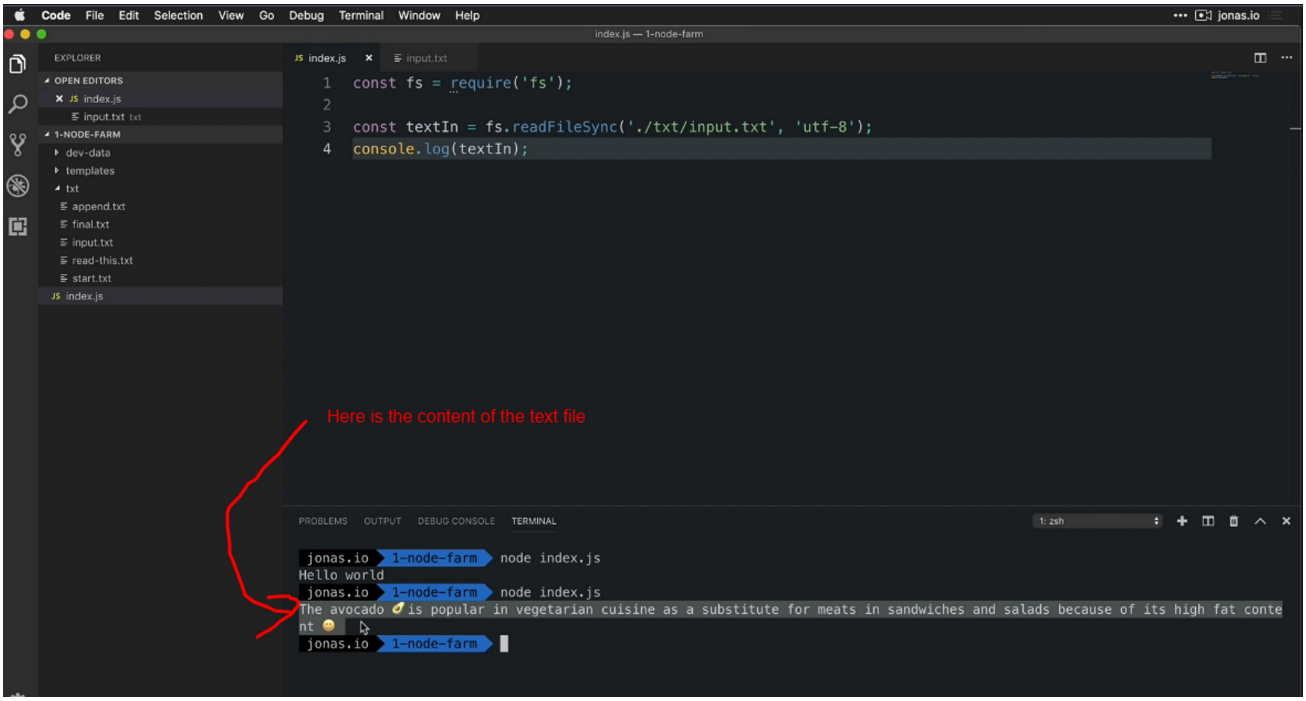


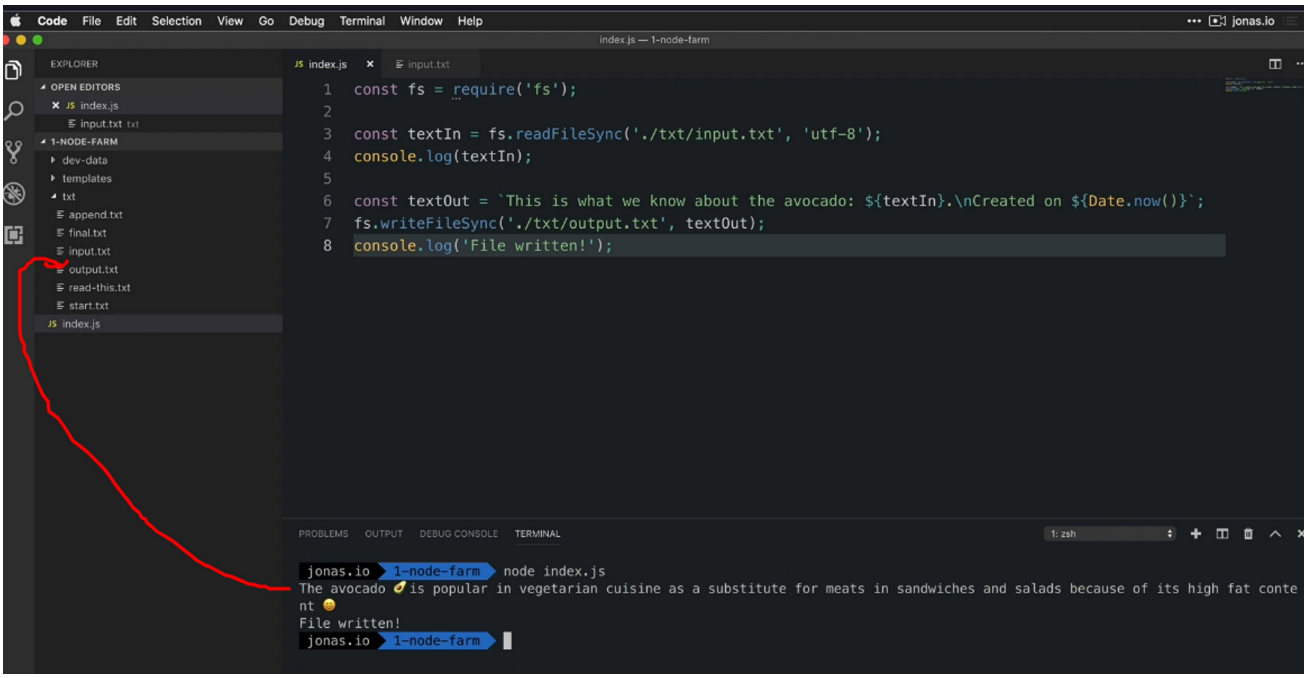
This is really exciting lecture, cause you're gonna learn how to read data from files, and also how to write data into files.



This is the synchronous version of file reading. There is also an asynchronous version.

This method takes 2 parameter. First one is the file path and second one is the 'character encoding' .


Here is the content of the text file
index.js — 1-node-farm
const fs = require('fs');
const textIn = fs.readFileSync('./txt/input.txt', 'utf-8');
console.log(textIn);
jonas.io 1-node-farm node index.js
Hello world
jonas.io 1-node-farm node index.js
The avocado is popular in vegetarian cuisine as a substitute for meats in sandwiches and salads because of its high fat conte
nt
jonas.io 1-node-farm

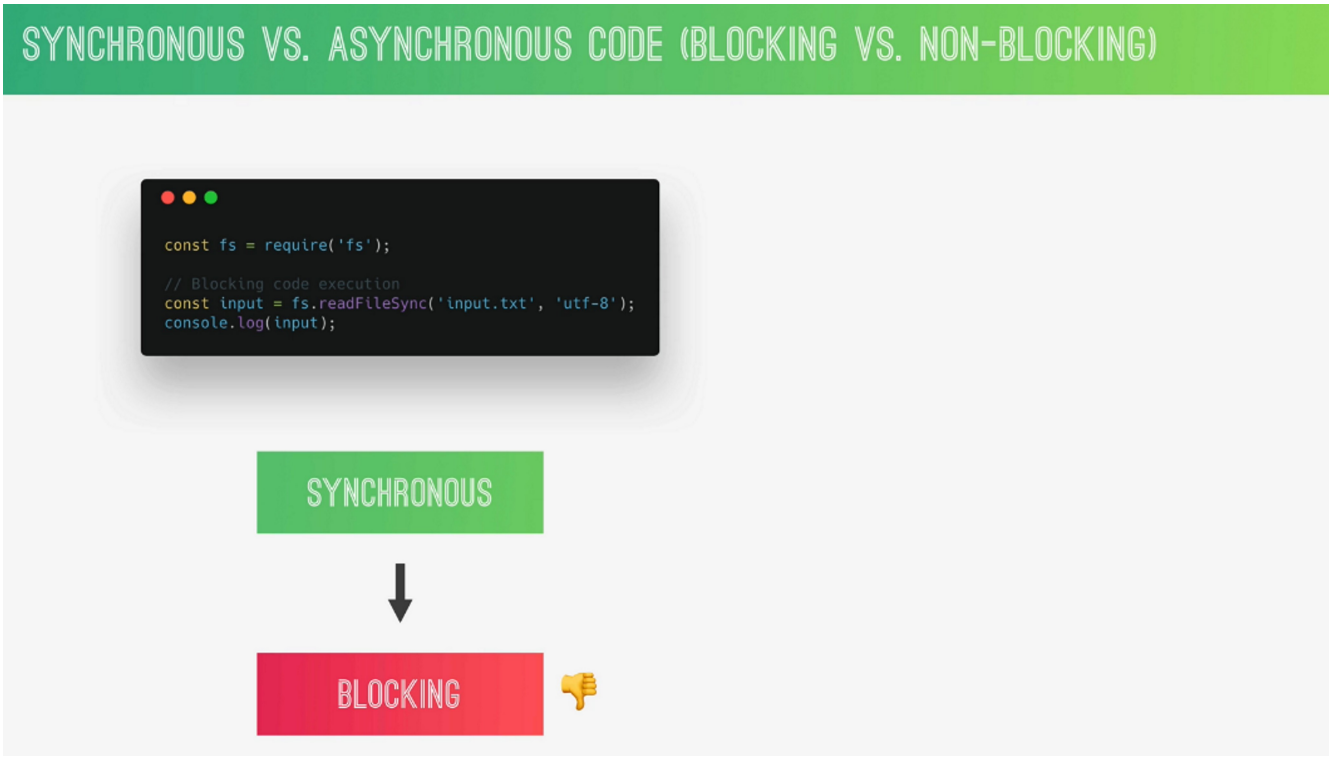

index.js — 1-node-farm
const fs = require('fs');
const textIn = fs.readFileSync('./txt/input.txt', 'utf-8');
console.log(textIn);
const textOut = `This is what we know about the avocado: ${textIn}.\nCreated on ${Date.now()}`;
fs.writeFileSync('./txt/output.txt', textOut);
console.log('File written!');
jonas.io 1-node-farm node index.js
The avocado is popular in vegetarian cuisine as a substitute for meats in sandwiches and salads because of its high fat conte
nt
File written!
jonas.io 1-node-farm

Let's take a moment to learn about the **asynchronous nature of Node.js**, which includes some absolutely fundamental concepts — like **synchronous**, **asynchronous**, **blocking**, and **non-blocking** code.
Understanding these will be super important for everything that's coming up in this section.

The piece of code we wrote in the last lecture — where we read a file and saved its content into a variable — was done **synchronously**.

That just means each statement is processed one after the other, **line by line**.
First, we require the file system module → then read the file → and finally log the result to the console.
So yeah, **each line waits** for the one before it to finish.
Now, this can be a real issue — especially for **slow operations** — because one line **blocks** the rest of the code from running.

That's why we call synchronous code **blocking code**: stuff only runs after the previous thing is done.
And because of how Node.js is built, this blocking behavior becomes a major problem.
(We'll dig into that more in the next slide.)



The solution to this problem in Node.js is to use **asynchronous, non-blocking code**.
With async code, we offload heavy tasks to be handled **in the background**.

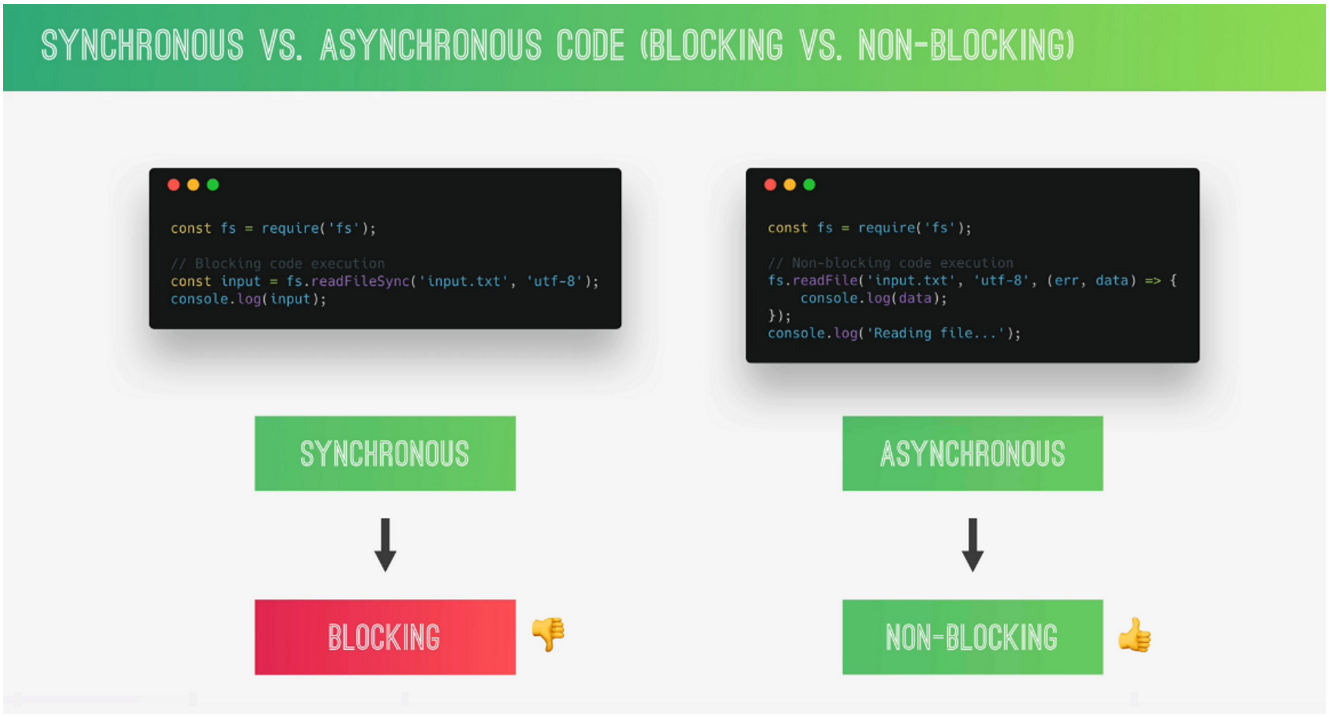
And once that task is done, a **callback function** we registered earlier gets called to handle the result.
💡 Meanwhile, the rest of the code keeps running — it **doesn't wait or freeze** — because that heavy task is chillin' in the background.
Basically, we're **deferring** our reaction to the result until it's ready — and that's what makes async code **non-blocking**, which is way more efficient.

Let's look at the example:

We use the async readFile() function and pass it a **callback**.
This starts reading the file in the background and then **immediately** moves on to the next line — printing "Reading file..." to the console.
So yeah — no blocking going on here.
Then once the file is fully read, the callback gets triggered, and the file data is finally logged to the console.

Totally different vibe compared to the synchronous version, right?

Now, here's the real question:
Why does it *have* to be this way in Node.js?
Why is blocking code a big deal?
And why are callbacks used so often?
Let's break that down next... 👀



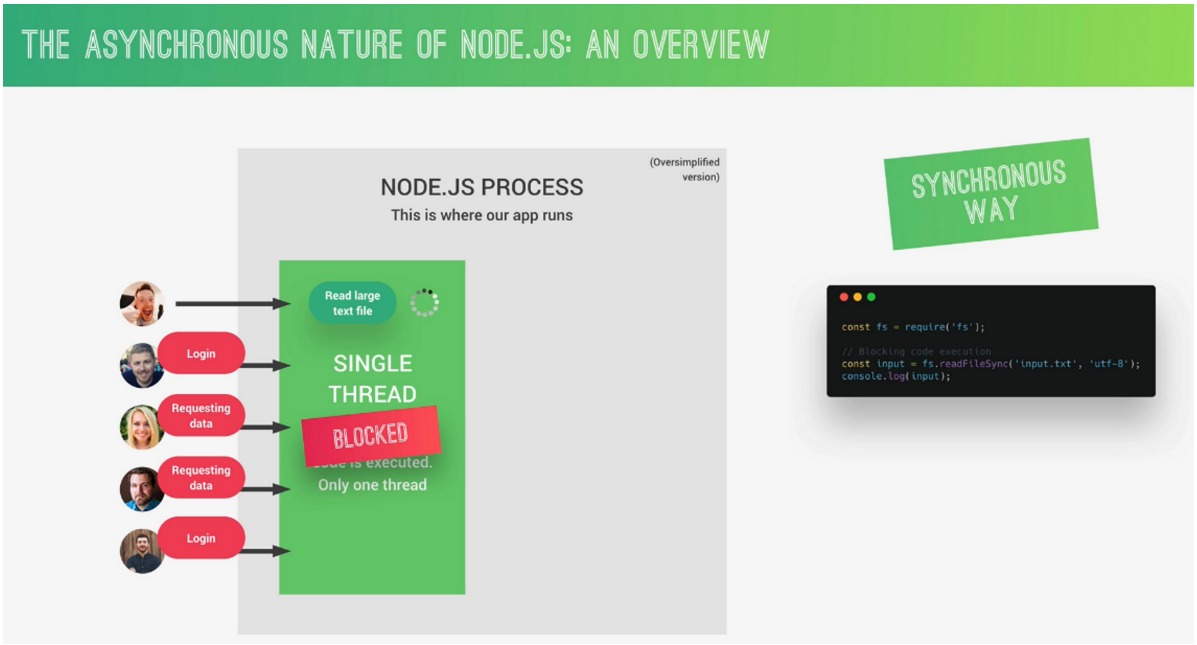
To really understand **why** blocking code is a problem in Node.js, we first gotta understand one core fact:

👉 **Node.js is single-threaded.**
That means when your Node.js app runs, it only uses **one thread** — just *one* set of instructions being executed on the CPU.
So basically, all your app's code runs on that **one single thread**. That's how Node was built — by design.

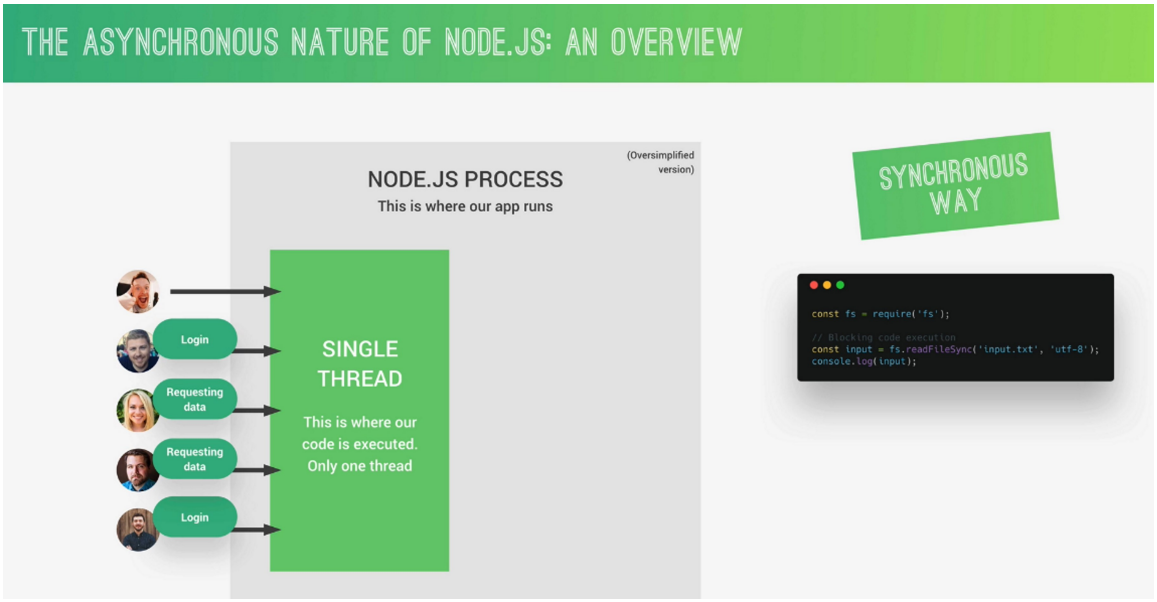
Now here's the kicker:
Everyone using your app — whether it's 5 users or **5 million** — they're **all sharing that same thread** 😳
Yep. Every user interaction, every request, every line of code — it's all running **on the same thread**.
So if just **one user's** code is running something **synchronous and slow** (like reading a huge file).

🚨 Boom — the **entire thread** is locked.
That means **all other users** have to *wait* until that task finishes before *anything else* can happen.

That might be okay if you only have a few users...
But if you've got thousands — or millions — of users?
Yeah... now you've got a problem 💀
Imagine a user hits a route that does a synchronous file read that takes 1 second.
For that 1 second, **nobody else can log in**, request data, or even ping the server.
Total traffic jam 😬
That's why we avoid blocking code in Node.js.
That's why we go async.
That's why callbacks (and later Promises, async/await, etc.) are a thing.



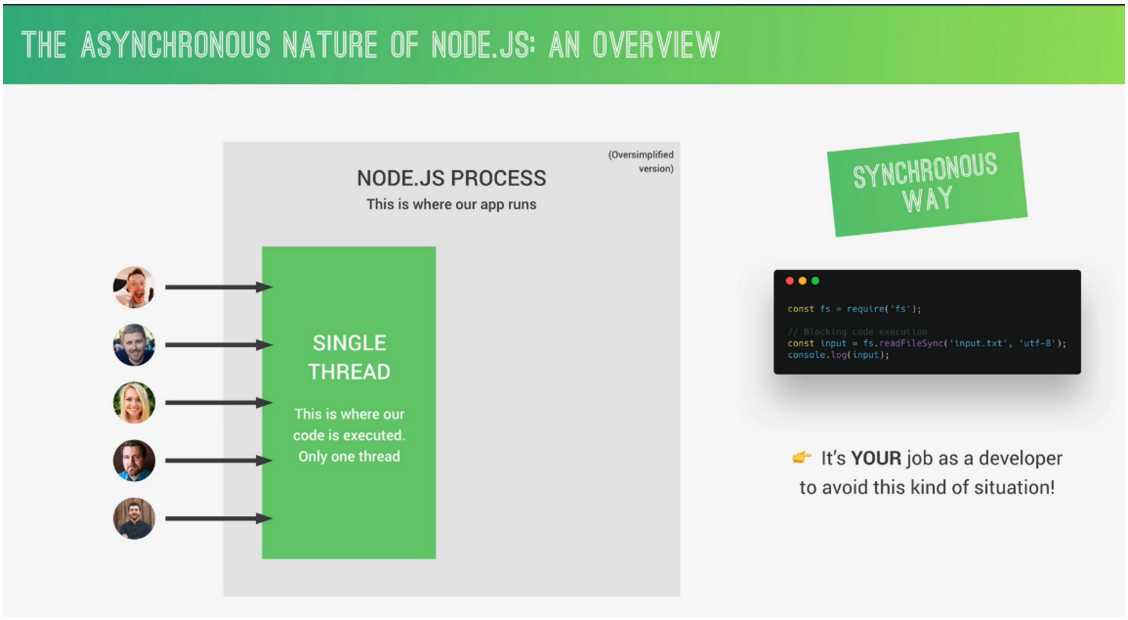
They will have to wait until the file is finished reading. Only when that happens they will finally be able to perform the simpler tasks, one after another.



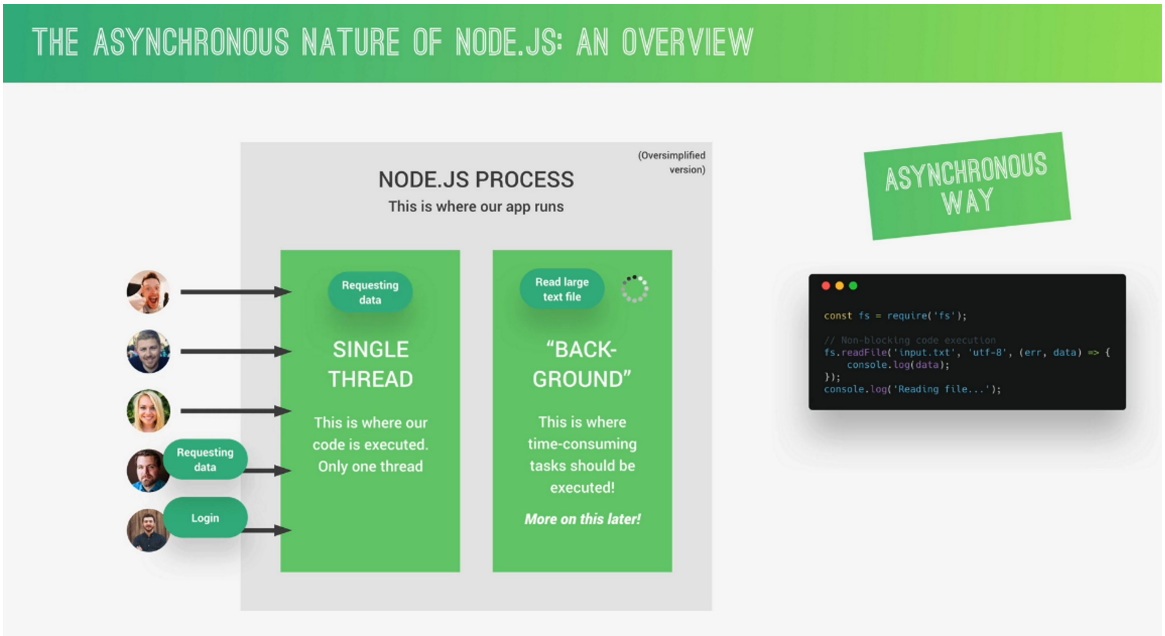
This should be enough to understand the concept. It's better to go step by step and keep things simple from the start, okay?

Now, here's how the situation would play out with synchronous, blocking code—which is obviously a terrible user experience.

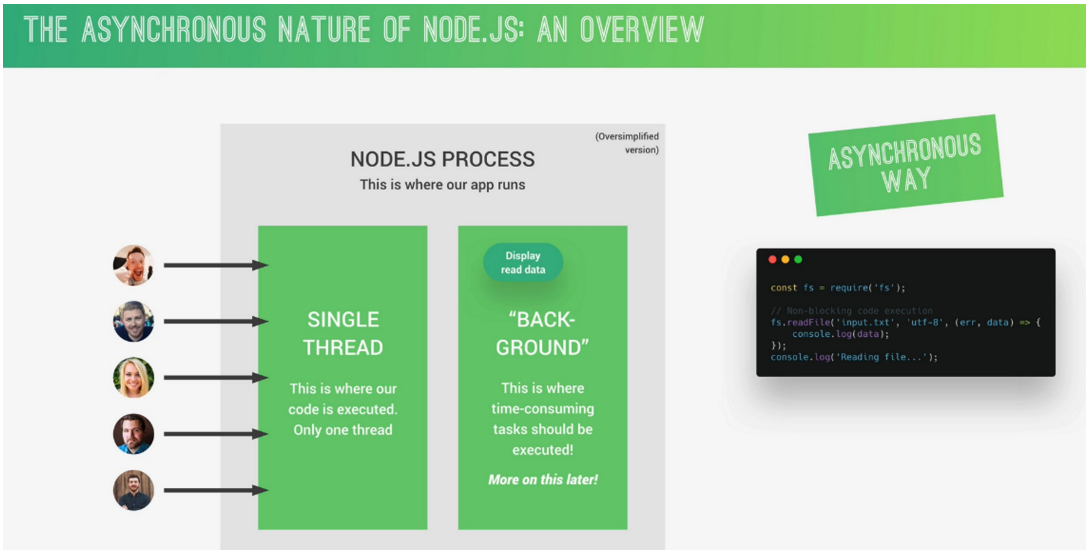
As a developer, it's your responsibility to avoid these issues by using asynchronous code.
So, in this case, we should use the asynchronous file read function, which doesn't block the single thread.



So, in this case, we should use the asynchronous file read function. Instead of blocking the single thread, it handles the heavy work in the background and waits until the file data is fully read.
We also register a callback function to be triggered once the data is available. Meanwhile, other users can continue their tasks on the single thread, one after another, while the file is still being read in the background.



Now once the data is read, our callback function will get executed.



Let's take a minute to understand the **asynchronous nature of Node.js**, which includes essential concepts like synchronous vs. asynchronous and blocking vs. non-blocking code. This is super important for everything coming up next.

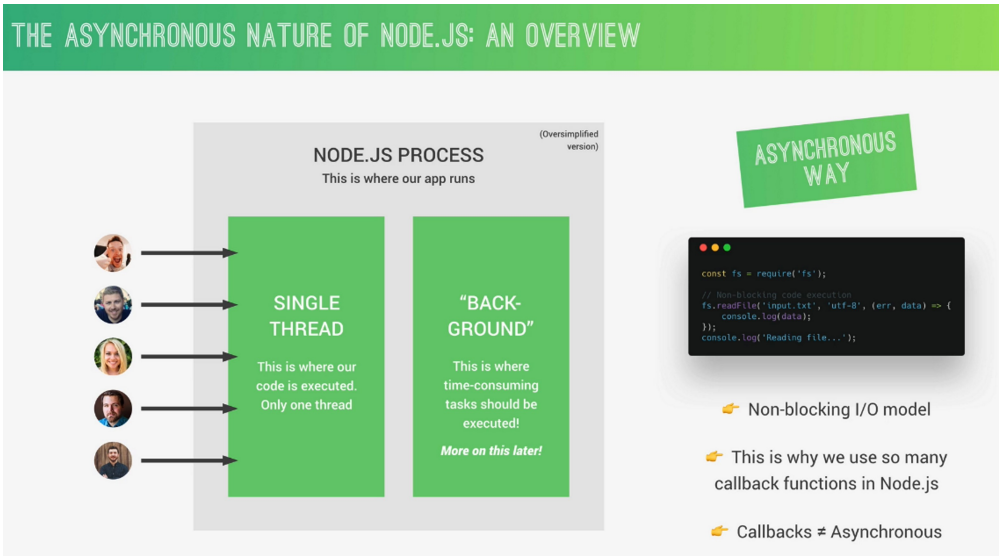
In the last lecture, we wrote code to read a file and store its content in a variable. That was done synchronously—meaning the code runs line by line, and each line waits for the previous one to finish. First, we require the file system module, then read the file, then log the result. Simple, but this causes problems with slow operations because each line **blocks** the next.

That's why synchronous code is also called **blocking code**—nothing else can run until the current task is done. In Node.js, that's a huge issue because the entire app runs on **a single thread**.
Yup, **Node.js is single-threaded**, meaning all users share the same thread. If one user triggers a slow, blocking operation like reading a large file, every other user has to wait. That's manageable with five users but a disaster with 5,000.

So, the solution? **Asynchronous, non-blocking code**.
With async code, heavy work is offloaded to the background. While that work runs, Node.js moves on. Once the task finishes, a registered **callback function** gets triggered to handle the result. This keeps the thread free for other users and tasks.

For example, readFile() is an async function that starts reading a file in the background and immediately moves on to the next statement—like logging something to the console. When the file is done loading, the callback runs and prints the data. No blocking. Clean. Efficient.
Now, why does Node work like this? Because **there's only one thread**, and blocking it even for a second affects every user. That's why Node leans heavily on callbacks and async behavior—to keep things fast and scalable.

Other languages like PHP spawn a new thread for each user, which is a totally different model. But Node's creator designed it this way to build **high-performance, scalable web apps**, and it just works brilliantly for that.



Now, just as a final note here, it's important to know that,
when we use callbacks in our code,
that doesn't automatically make it asynchronous, all right?
So, passing functions around into other functions
is quite common in JavaScript,
but of course, again, that doesn't make them
asynchronous automatically, okay?
It only works this way for some functions in the Node API,
such as the readFile function
and many, many more, as people explore in the future.

So, this callback model that we just discussed,
where one function is called
once the one before has finished its work,

can quickly lead to some hard-to-read and unmanageable code.
Just take this example where the second file read
depends on the first one,

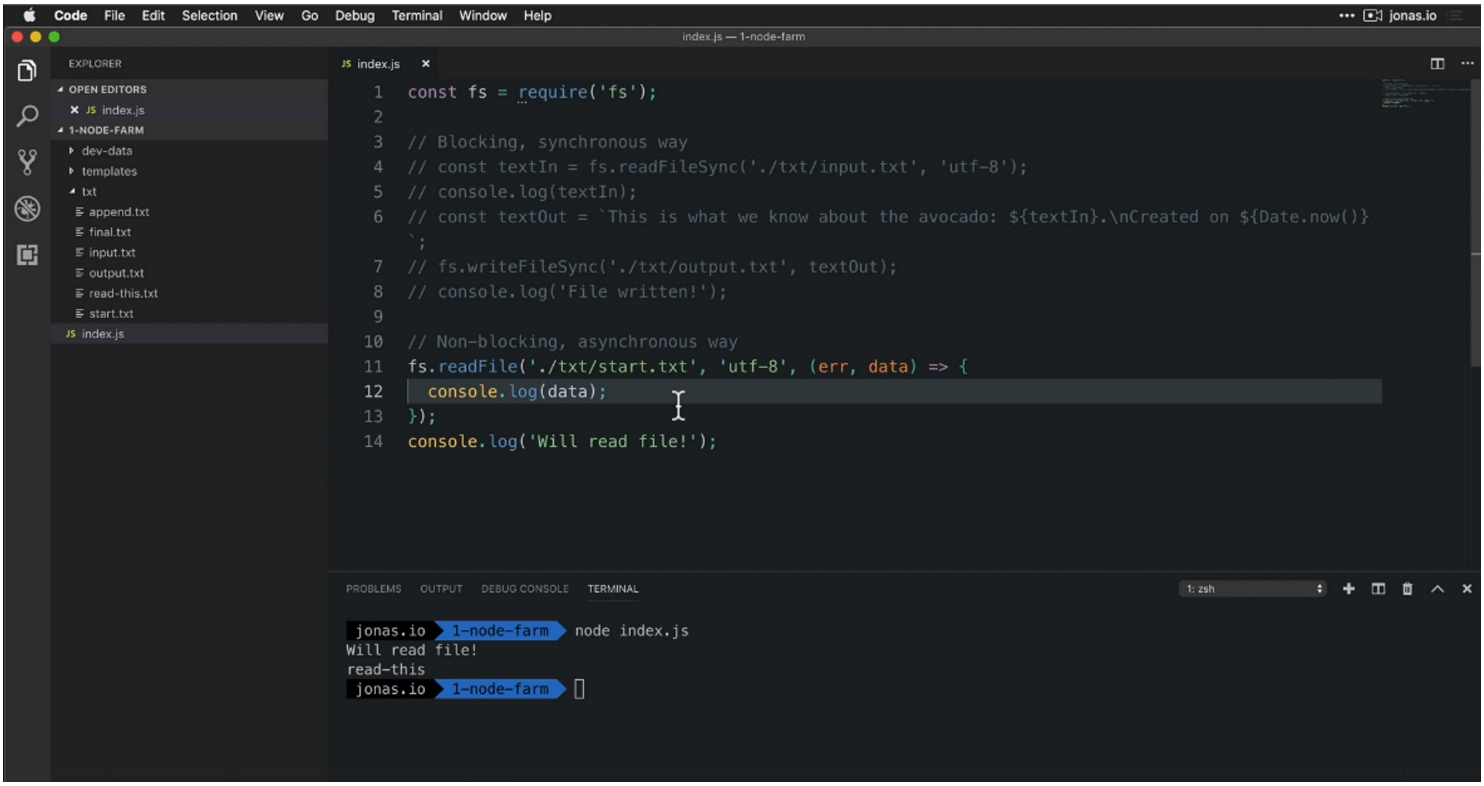
then, the third file read depends on the second one,
and finally, we want to use the final data
to write a file as a result.

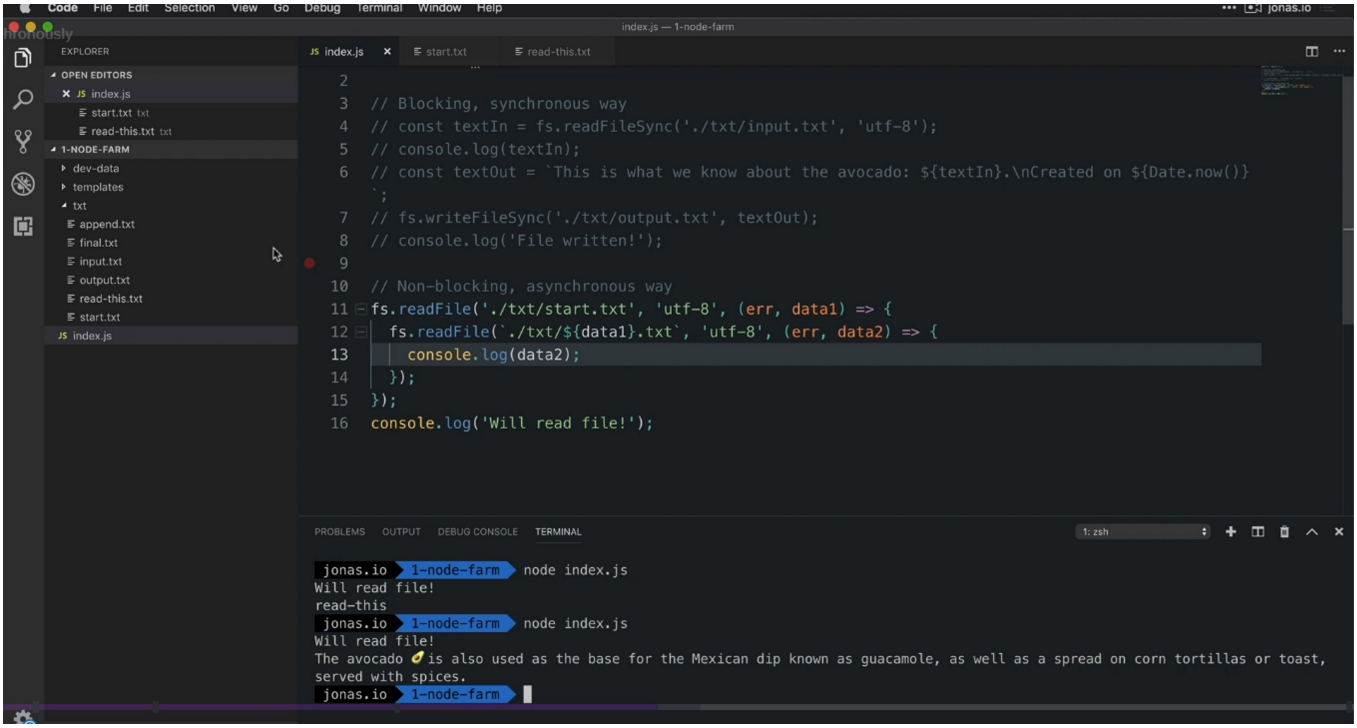
That looks quite confusing, right?
I mean, it's gonna work just fine,
but it's just hard to reason about—
and that is just with four levels deep.
Imagine you had like 10 or 20 levels,

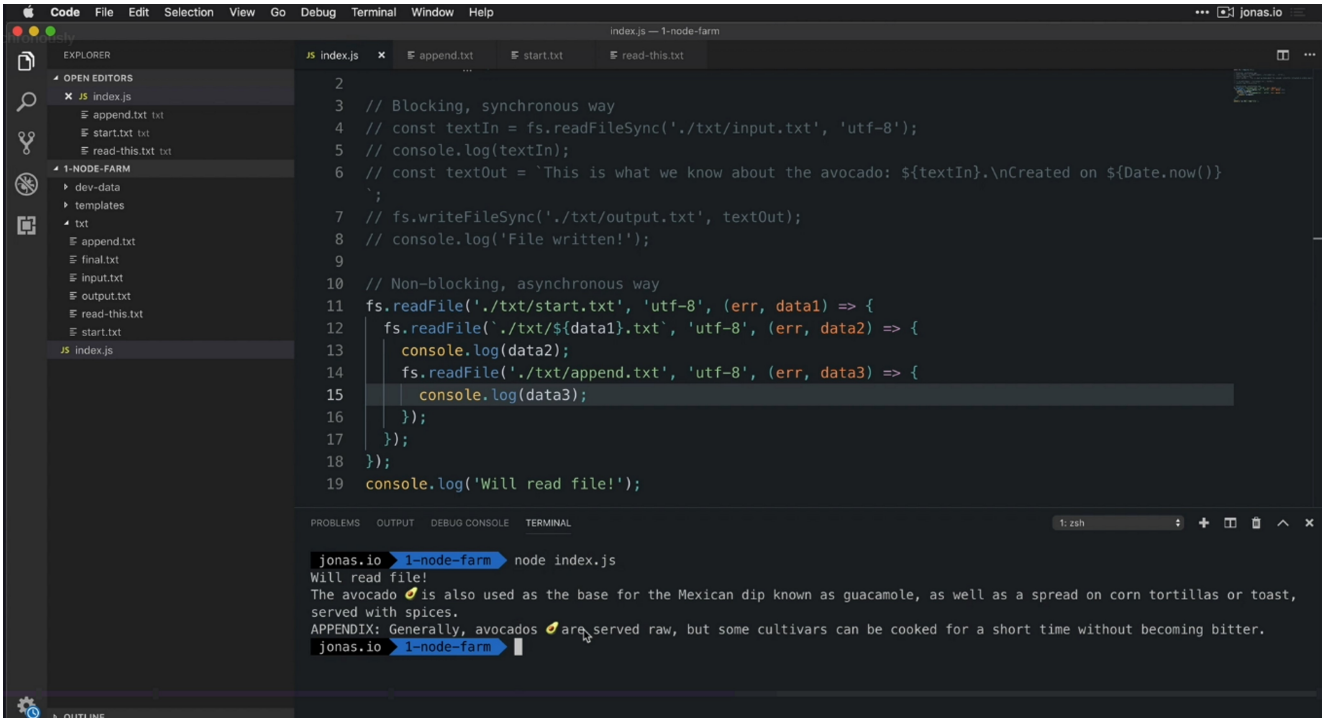
which is actually not that uncommon.
Anyway, this is what we call the **callback hell**.
It's such a common problem,
that it already got its own name.
And do you notice this triangular shape here?
That's a very clear sign that you're in callback hell.
Now, how do we actually escape callback hell?

Well, we can use more advanced tools for handling
asynchronous code, like ES6 **promises**
or even better, ES8 **async/await**.

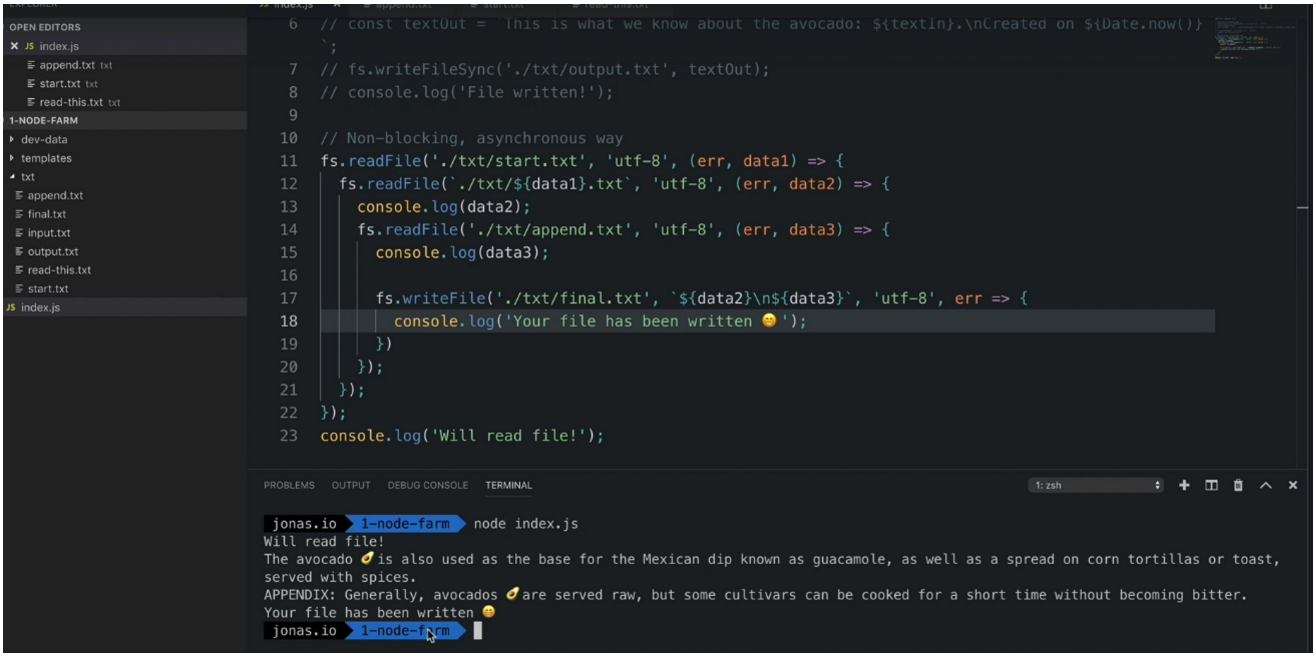
In this video, we are gonna read files and write two files just like before. But instead in an asynchronous way.

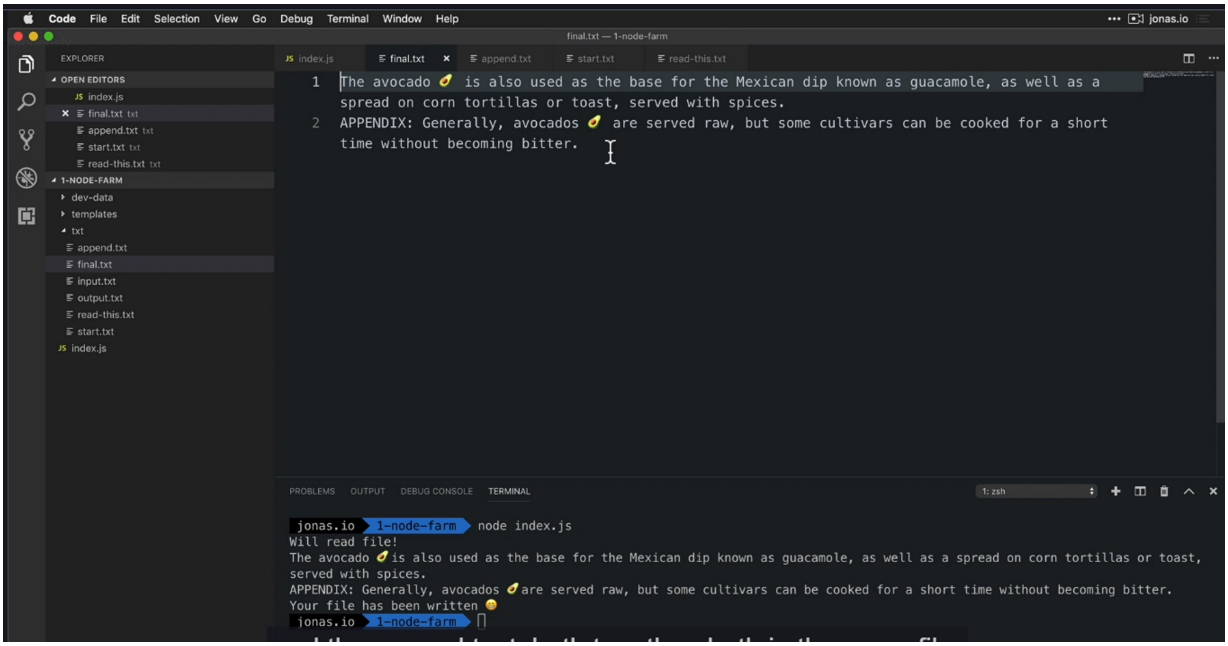


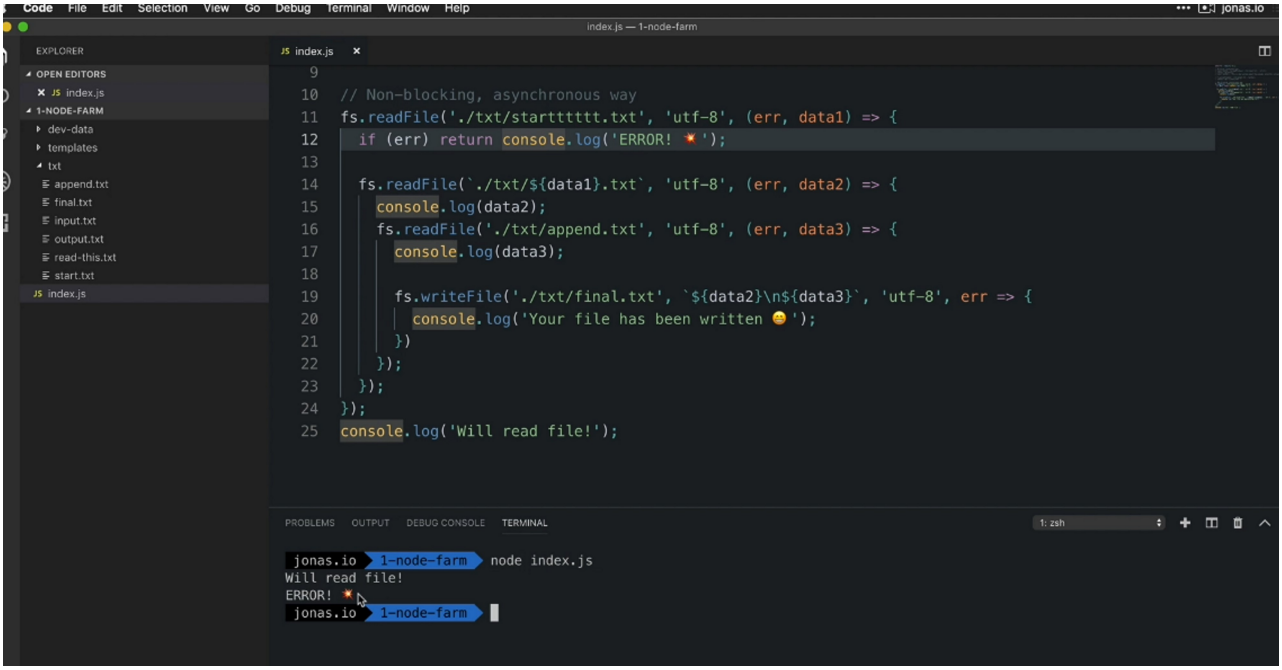




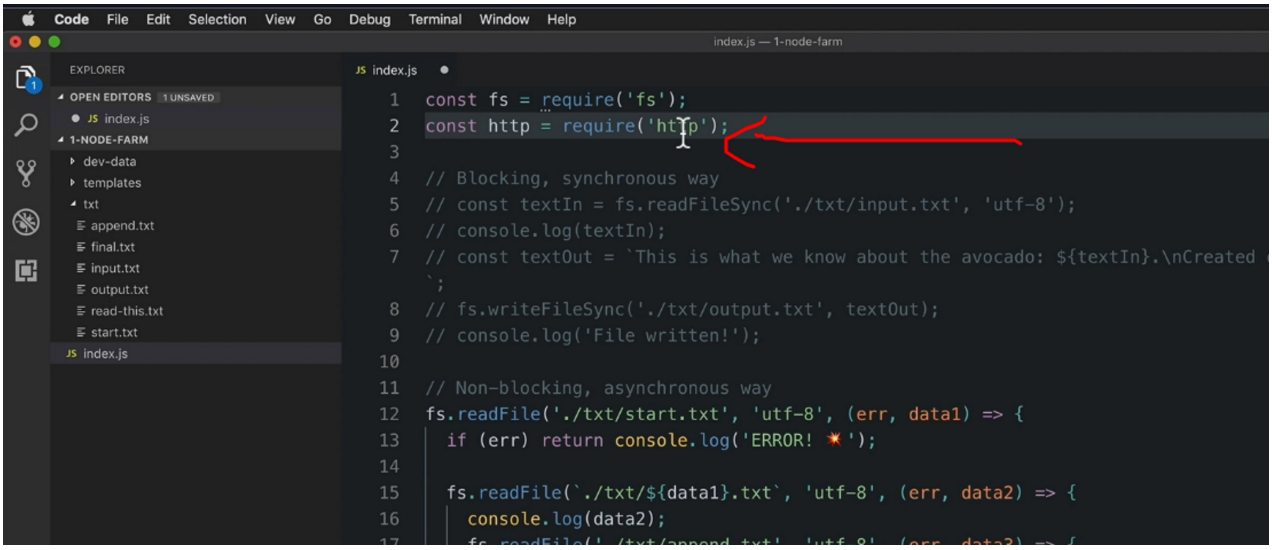
But this code is callback hell!


OPEN EDITORS
index.js
append.txt
start.txt
read-this.txt
1-NODE-FARM
dev-data
templates
txt
append.txt
final.txt
input.txt
output.txt
read-this.txt
start.txt
index.js
// const textOut = `This is what we know about the avocado: ${textIn}.\nCreated on ${Date.now()}`;
// fs.writeFileSync('./txt/output.txt', textOut);
// console.log('File written!');
// Non-blocking, asynchronous way
fs.readFile('./txt/start.txt', 'utf-8', (err, data1) => {
fs.readFile(`./txt/${data1}.txt`, 'utf-8', (err, data2) => {
console.log(data2);
fs.readFile('./txt/append.txt', 'utf-8', (err, data3) => {
console.log(data3);
fs.writeFile('./txt/final.txt', `${data2}\n${data3}`, 'utf-8', err => {
console.log('Your file has been written 😁');
})
});
});
});
console.log('Will read file!');
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: zsh
jonas.io 1-node-farm node index.js
Will read file!
The avocado 🥑 is also used as the base for the Mexican dip known as guacamole, as well as a spread on corn tortillas or toast, served with spices.
APPENDIX: Generally, avocados 🥑 are served raw, but some cultivars can be cooked for a short time without becoming bitter.
Your file has been written 😁
jonas.io 1-node-farm


Code File Edit Selection View Go Debug Terminal Window Help
jonas.io
final.txt — 1-node-farm
EXPLORER
index.js
final.txt
append.txt
start.txt
read-this.txt
OPEN EDITORS
index.js
final.txt
append.txt
start.txt
read-this.txt
1-NODE-FARM
dev-data
templates
txt
append.txt
final.txt
input.txt
output.txt
read-this.txt
start.txt
index.js
1 The avocado 🥑 is also used as the base for the Mexican dip known as guacamole, as well as a spread on corn tortillas or toast, served with spices.
2 APPENDIX: Generally, avocados 🥑 are served raw, but some cultivars can be cooked for a short time without becoming bitter.
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: zsh
jonas.io 1-node-farm node index.js
Will read file!
The avocado 🥑 is also used as the base for the Mexican dip known as guacamole, as well as a spread on corn tortillas or toast, served with spices.
APPENDIX: Generally, avocados 🥑 are served raw, but some cultivars can be cooked for a short time without becoming bitter.
Your file has been written 😁
jonas.io 1-node-farm

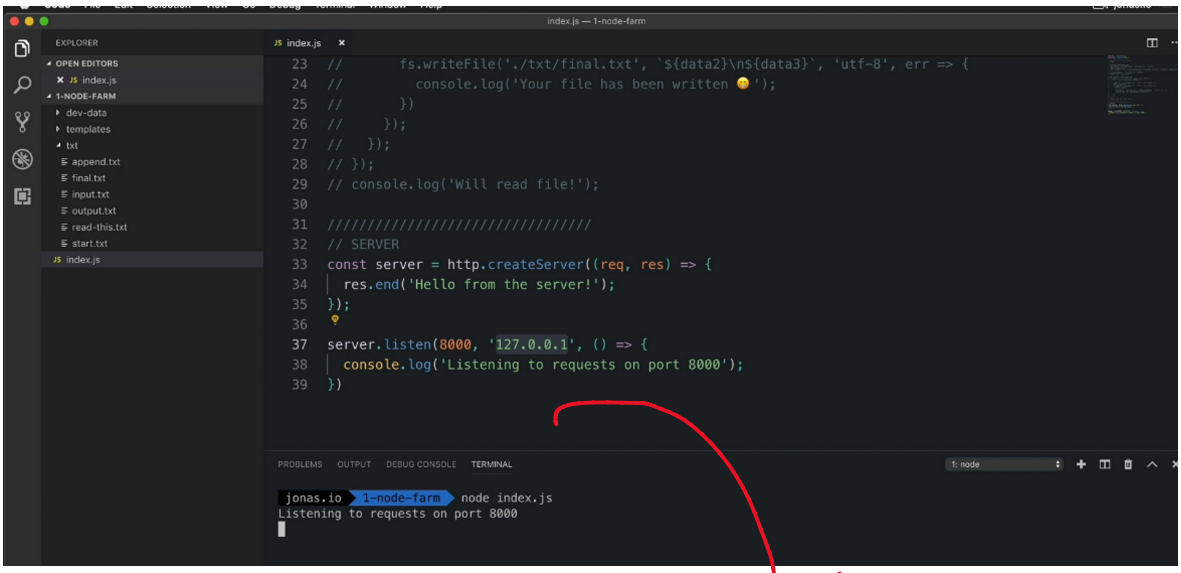


This video is yet another really exciting one,
because now we're gonna create our very first
real web server capable of accepting requests
and sending back responses.
And the first step is to include yet another package
or another module,
and this one is called **http**.



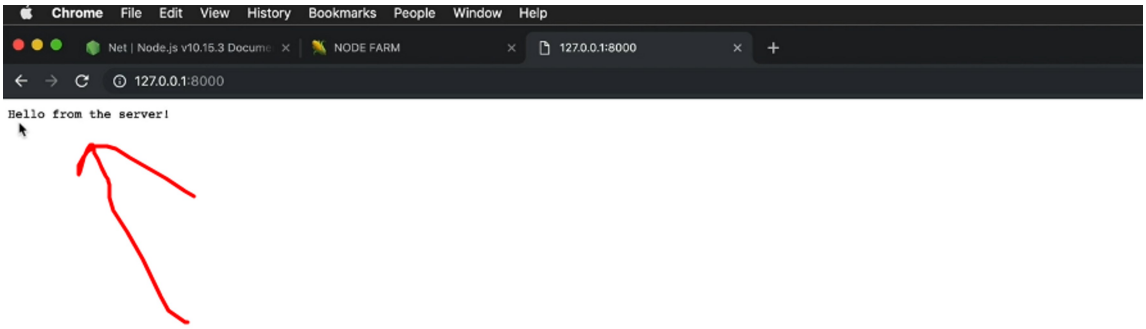
In order to build our server, we have to do two things. First we have to create a server then we start the server.

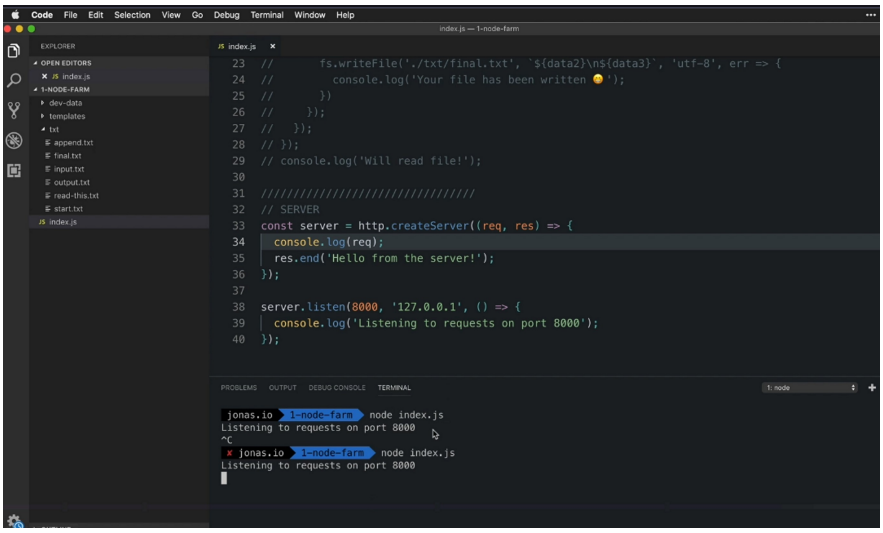
So http.createServer will accept a callback function, which will be fired off each time a new request hits our server, And this callback function gets access to two very important and fundamental variables. It is the request variable and response variable.



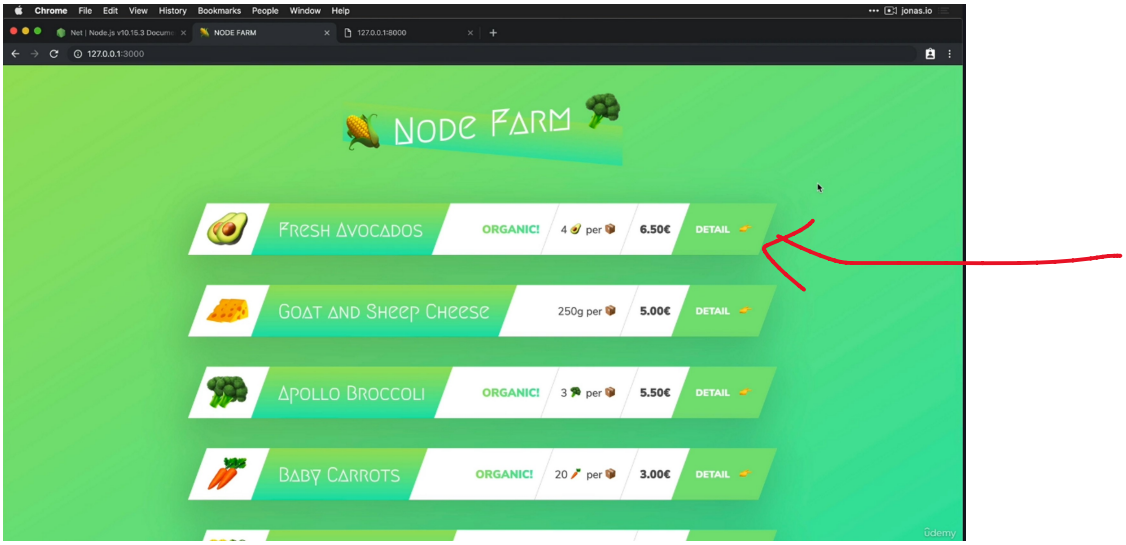
Default IP address. This is the host which is localhost.

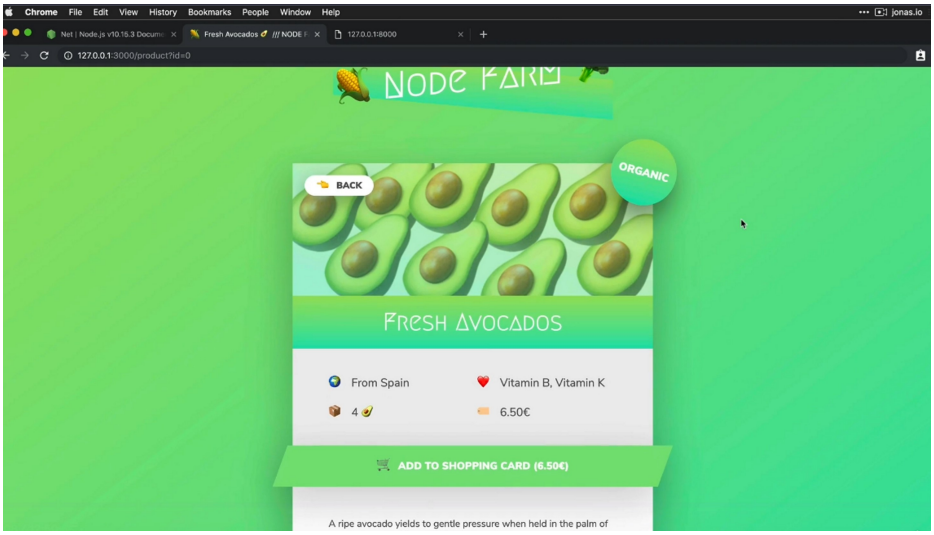
As you can see the app keeps running, because that's because of something called the **event loop.**



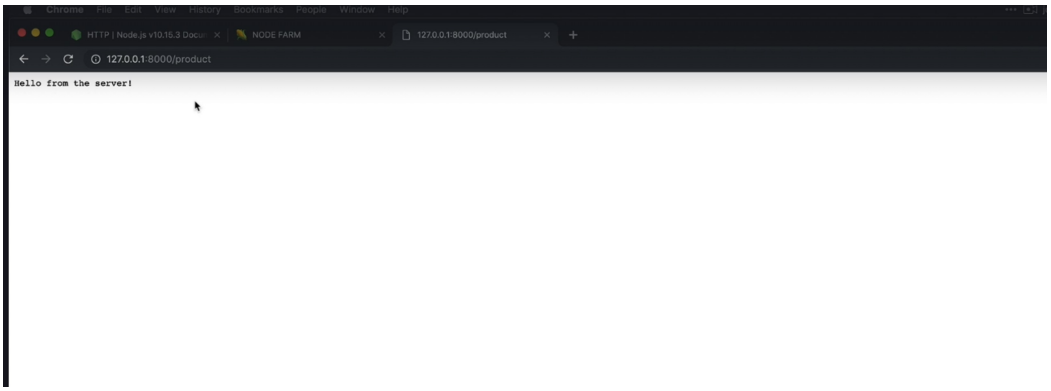


We are going to build this project in the next lecture.

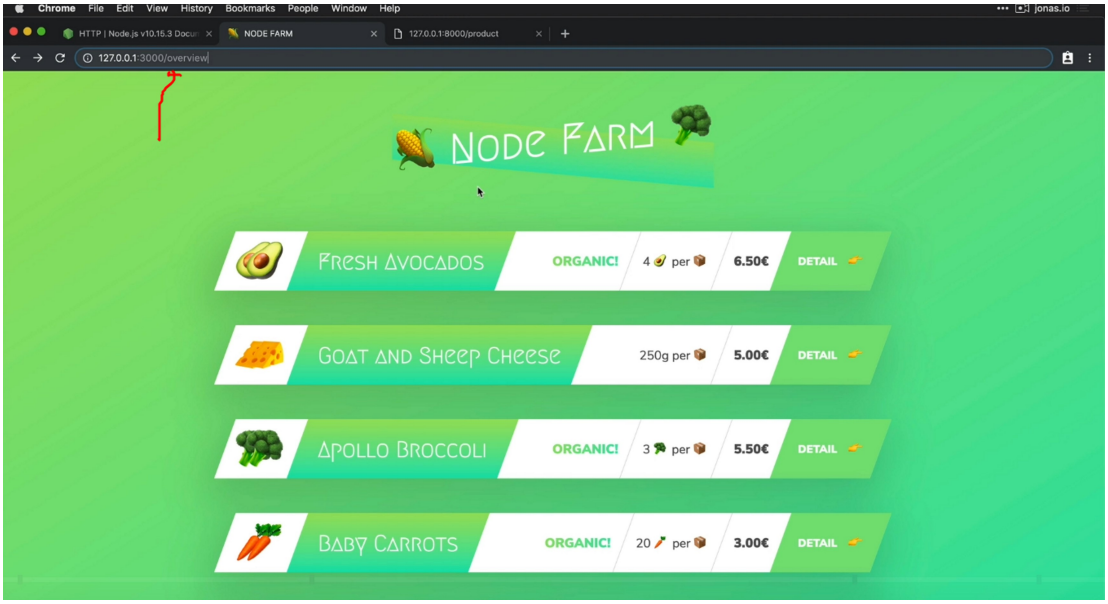


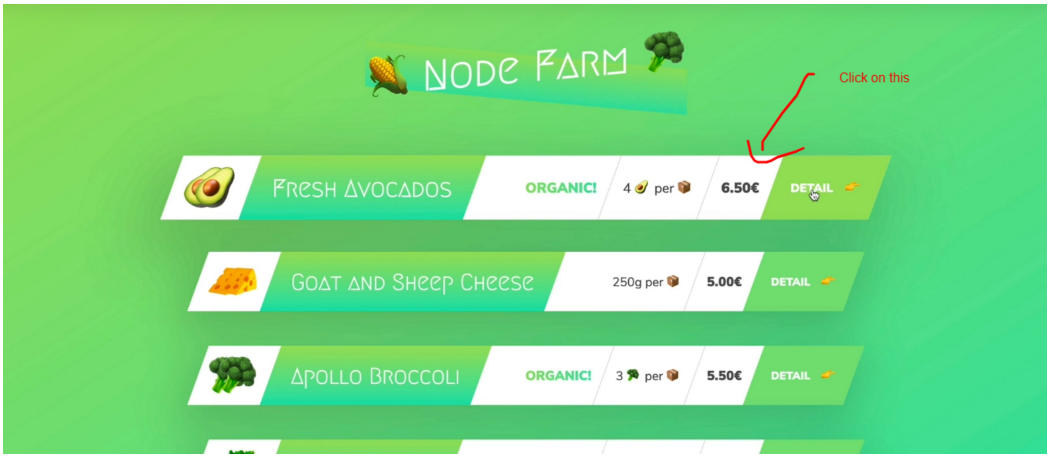


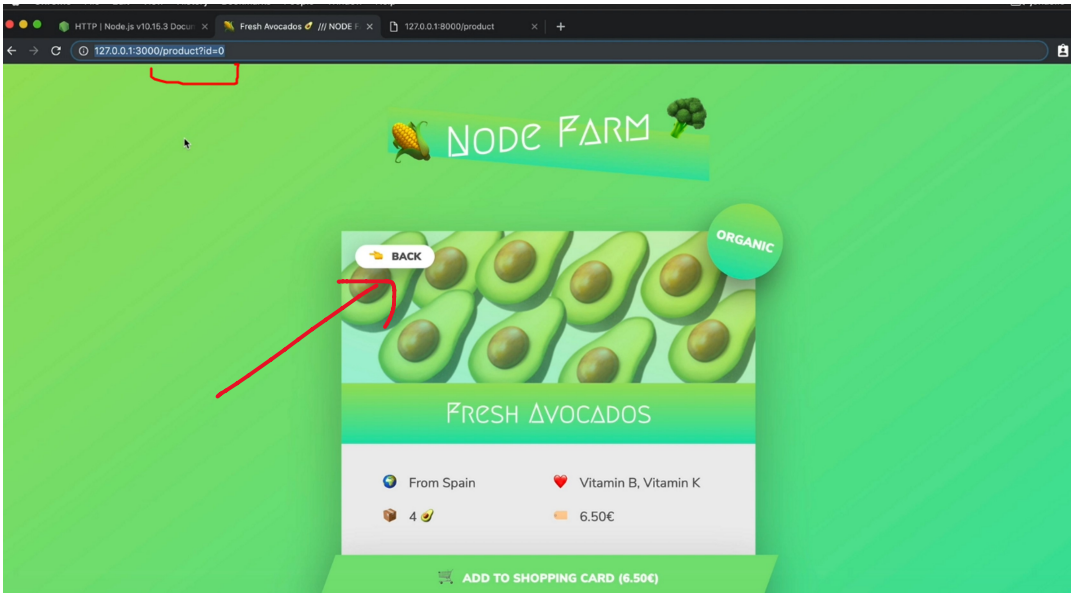
Not work! We get the exact same result.



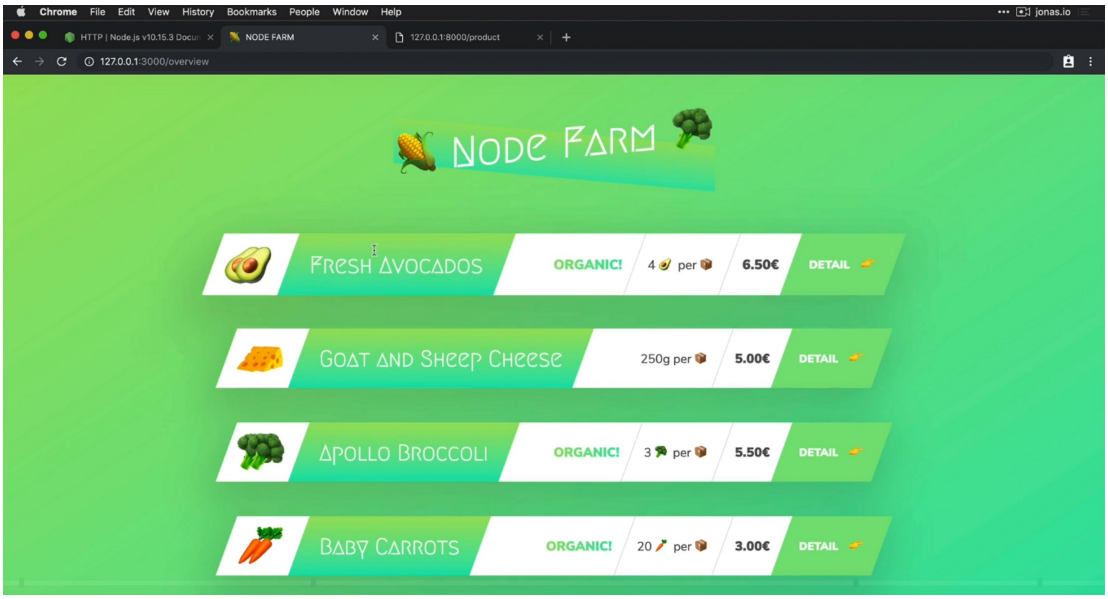
But what we want to build bro? Lets' look at our final project.



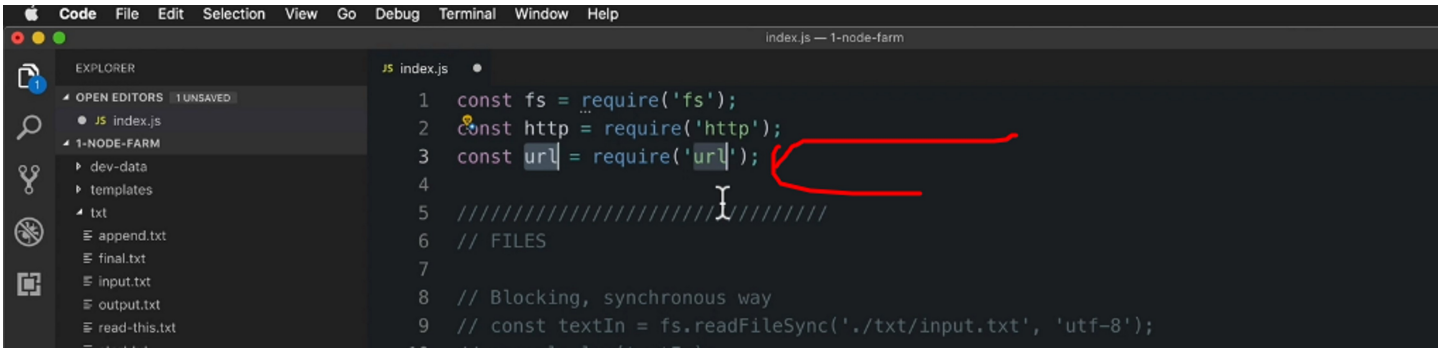


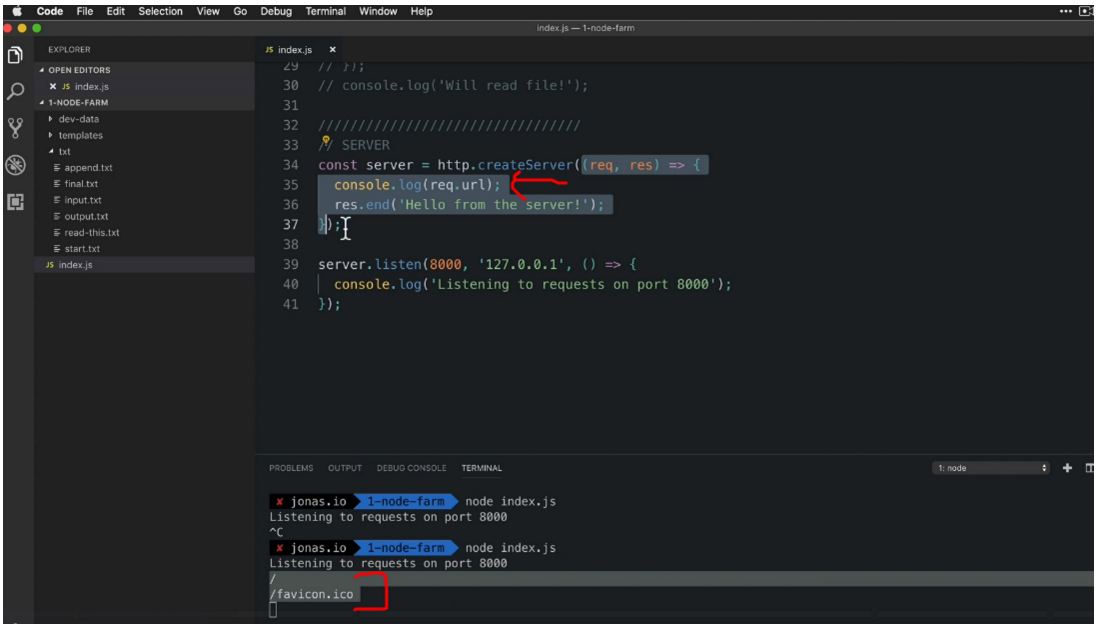


Click on 'Back' button and get back to the 'overview'



So Routing basically means implementing different actions for different URL.
Routing can actually become very complicated in a big real world application. In that case we use a tool like **Express.**

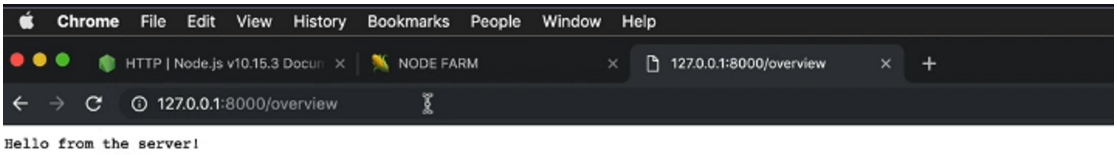


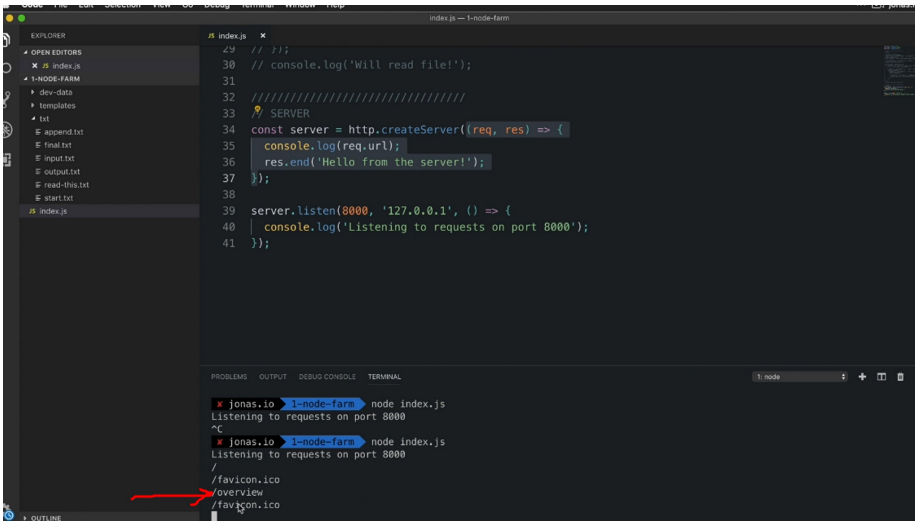


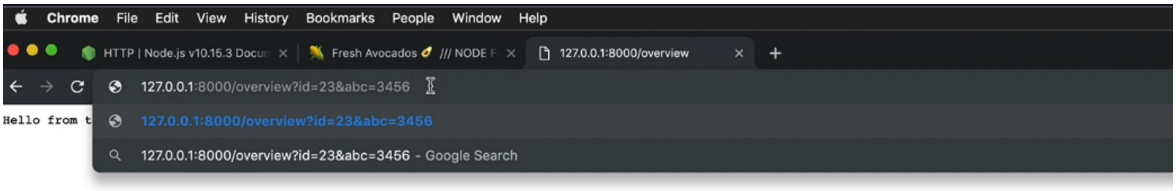
Open up the browser and hit the URL, but wait a minute. We got two response. Because

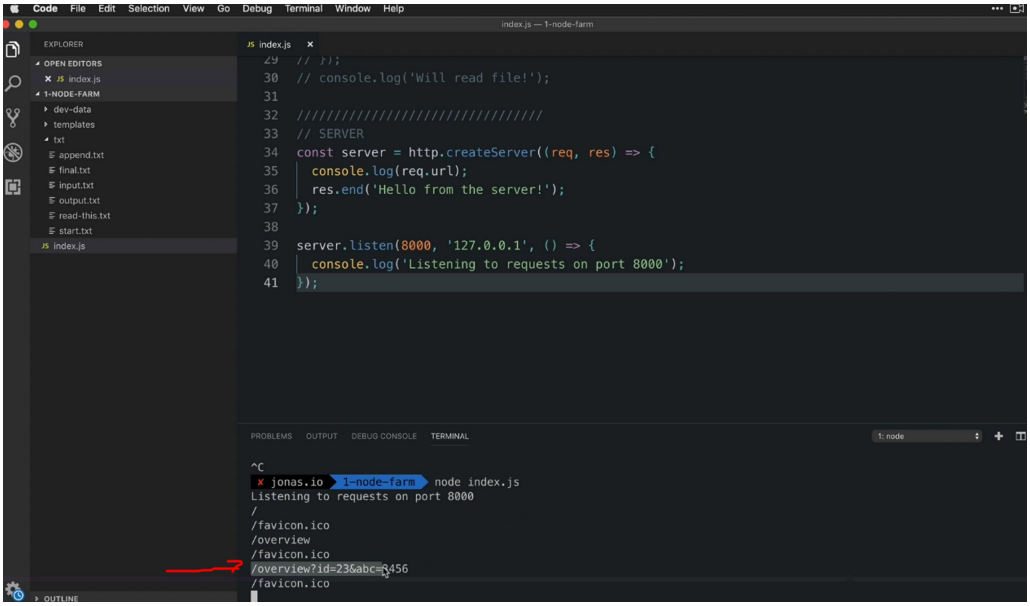
When we are using the browser, browser automatically performs a request for the website's favicon. Ignore it now.

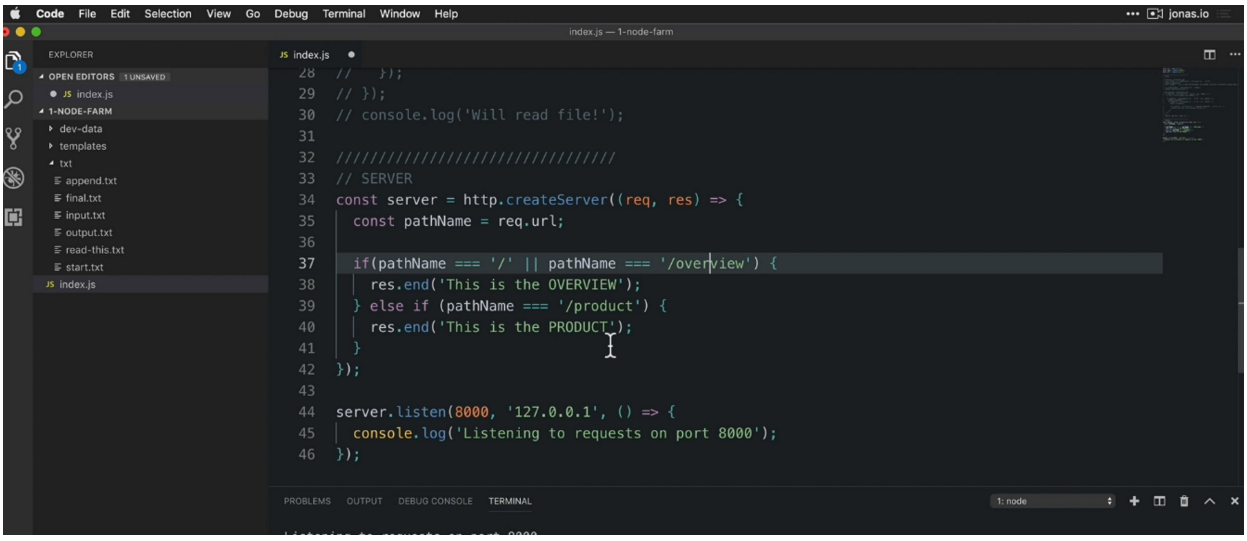
Hit /overview

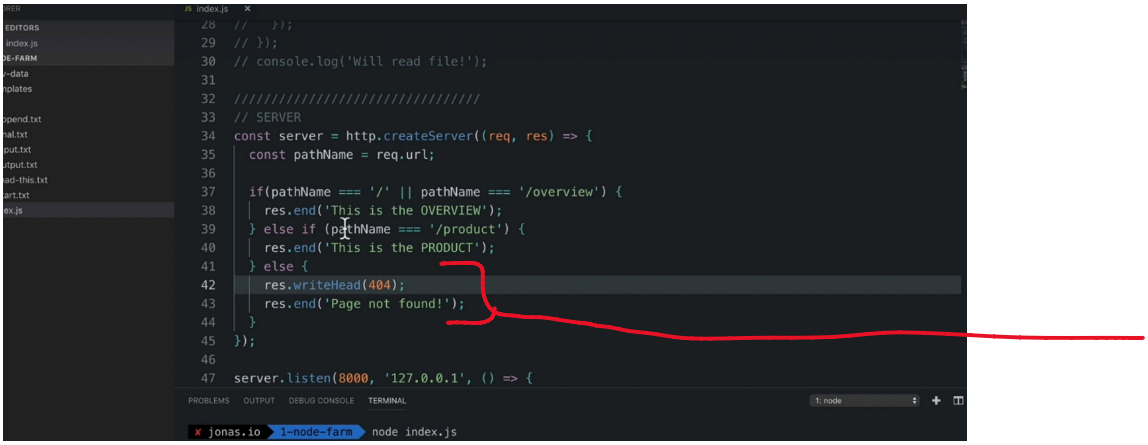


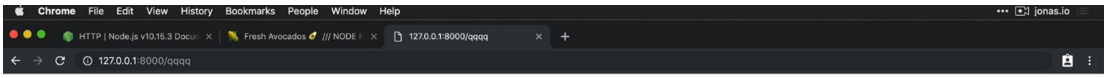


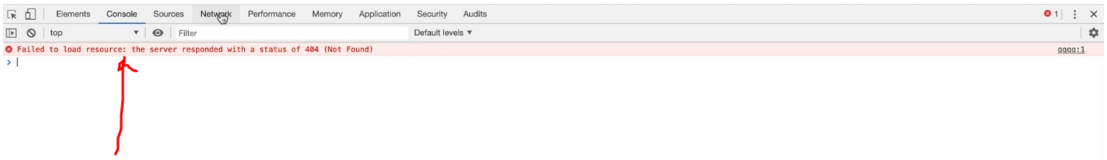

Chrome File Edit View History Bookmarks People Window Help
HTTP | Node.js v10.16.3 Docu
Fresh Avocados
/// NODE F
127.0.0.1:8000/overview
127.0.0.1:8000/overview?id=23&abc=3456
Hello from t
127.0.0.1:8000/overview?id=23&abc=3456
127.0.0.1:8000/overview?id=23&abc=3456 - Google Search


Code File Edit Selection View Go Debug Terminal Window Help
index.js — 1-node-farm
EXPLORER
OPEN EDITORS
index.js
1-NODE-FARM
dev-data
templates
txt
append.txt
final.txt
input.txt
output.txt
read-this.txt
start.txt
index.js
// console.log('Will read file!');
////////////////////////////////
// SERVER
const server = http.createServer((req, res) => {
console.log(req.url);
res.end('Hello from the server!');
});
server.listen(8000, '127.0.0.1', () => {
console.log('Listening to requests on port 8000');
});
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: node
^C
jonas.io 1-node-farm node index.js
Listening to requests on port 8000
/
/favicon.ico
/overview
/favicon.ico
/overview?id=23&abc=3456
/favicon.ico
OUTLINE

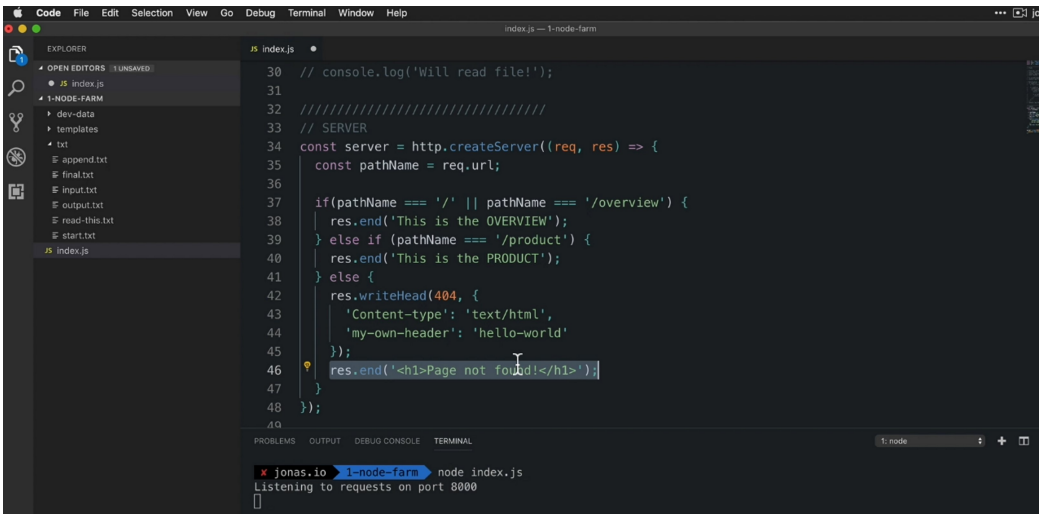

Code File Edit Selection View Go Debug Terminal Window Help
jonas.io
index.js — 1-node-farm
EXPLORER
OPEN EDITORS 1 UNSAVED
index.js
1-NODE-FARM
dev-data
templates
txt
append.txt
final.txt
input.txt
output.txt
read-this.txt
start.txt
index.js
// });
// console.log('Will read file!');
////////////////////////////////
// SERVER
const server = http.createServer((req, res) => {
const pathName = req.url;
if(pathName === '/' || pathName === '/overview') {
res.end('This is the OVERVIEW');
} else if (pathName === '/product') {
res.end('This is the PRODUCT');
}
});
server.listen(8000, '127.0.0.1', () => {
console.log('Listening to requests on port 8000');
});
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: node







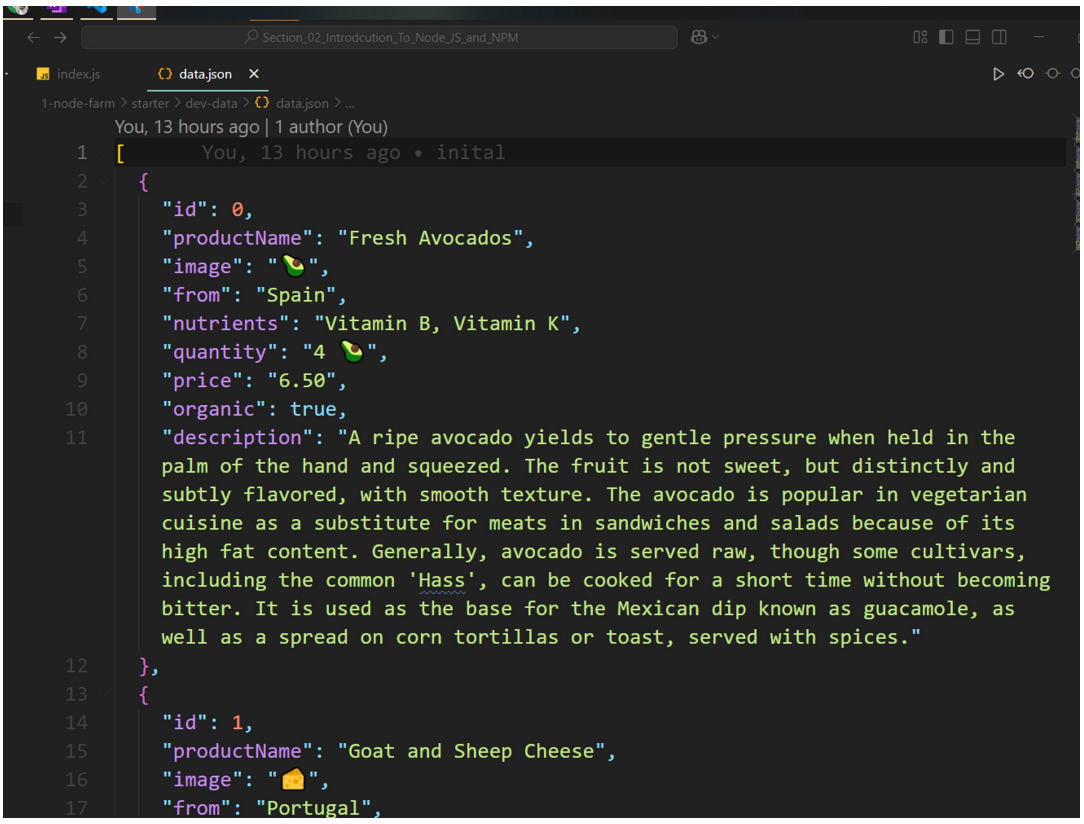
HTTP header is basically a piece of information about the response we are sending back.



Now, to start, what actually is an API?

Well, the long answer, you're gonna learn in one of the next sections,
but for now, the short answer, at least in this context of web APIs, basically,
an API is a service from which we can request some data,
so in this case, the data that the user wants to request
is data about the products that we are offering in this node farm,
so in this project.

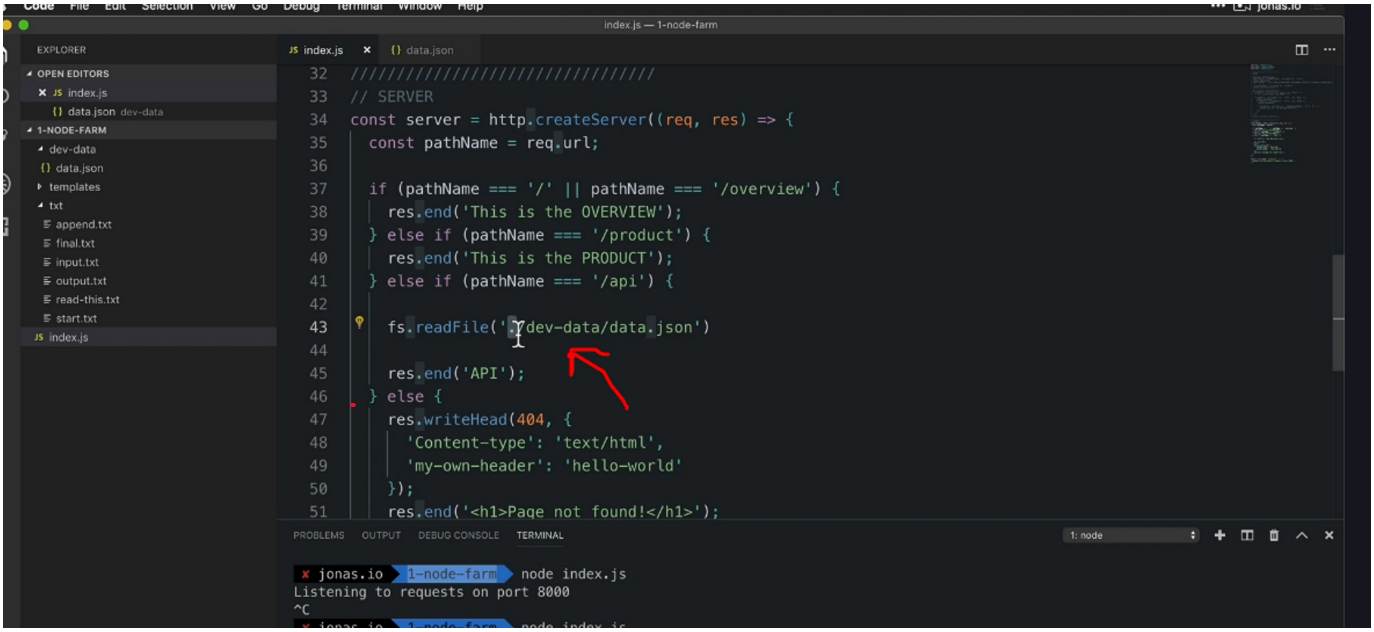
So I have here this dev-data folder,
and in there, I have a JSON file,
and JSON, in case you don't know,
is a very simple text format that looks a lot like JavaScript object.



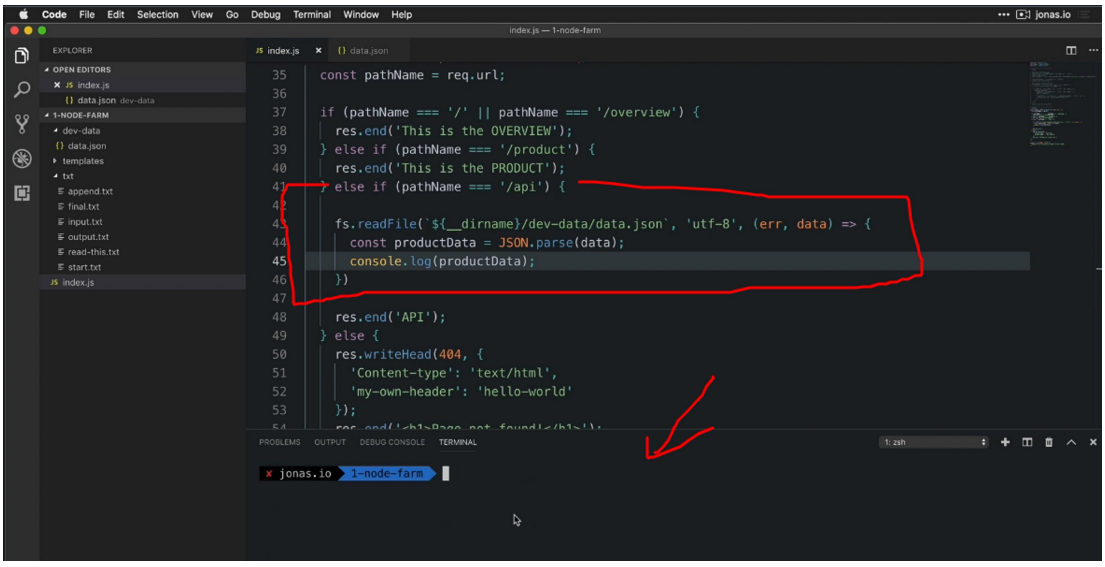
this dot here actually refers to the directory from which we run the node command in the terminal.
So, again, this dot here right now represents this current folder here
because that is where we actually run the node command,

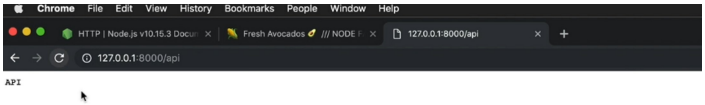
so in this 1-node-farm folder, which is our current working directory anyway in this case.
But we could have run the node command somewhere else, and then the dot would mean something else.
So, for example, we could perfectly fine go to the desktop and run node there,
and then specify the whole path to index.js.

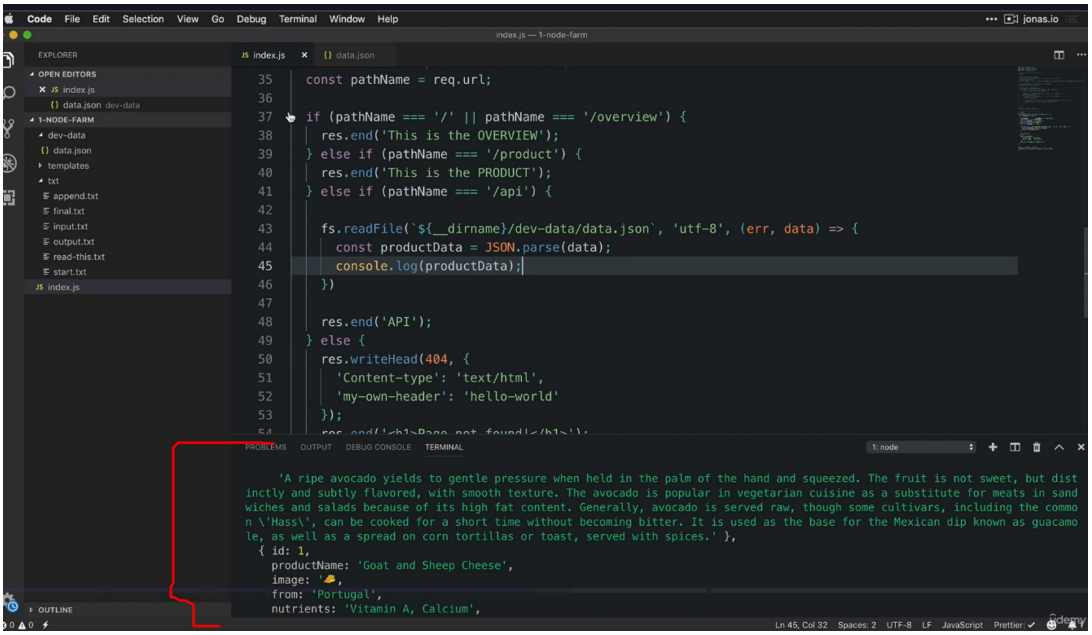
But we could do that, but then in this case, the dot would mean the desktop.
So if we started the script from the desktop,
then this here, this dot, would mean the desktop.
And so that is not always ideal, and therefore, there is a better way.

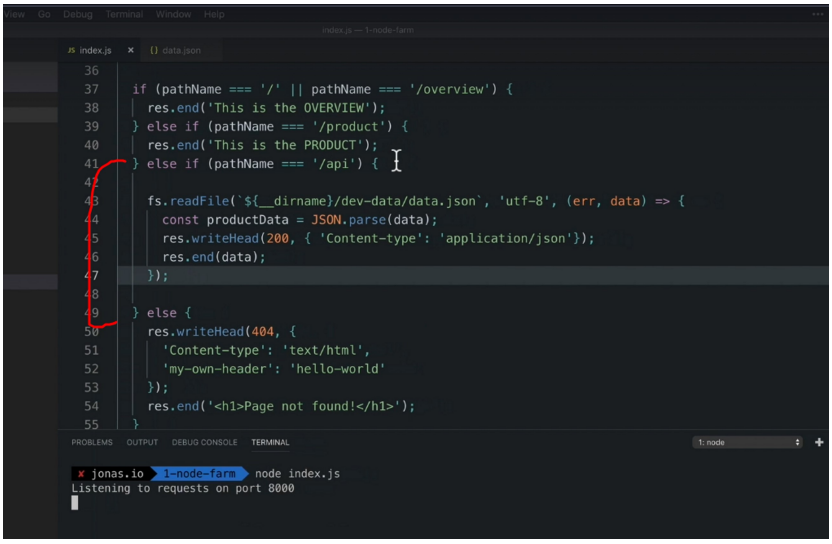


__dirName => directory name, where the current file is located.

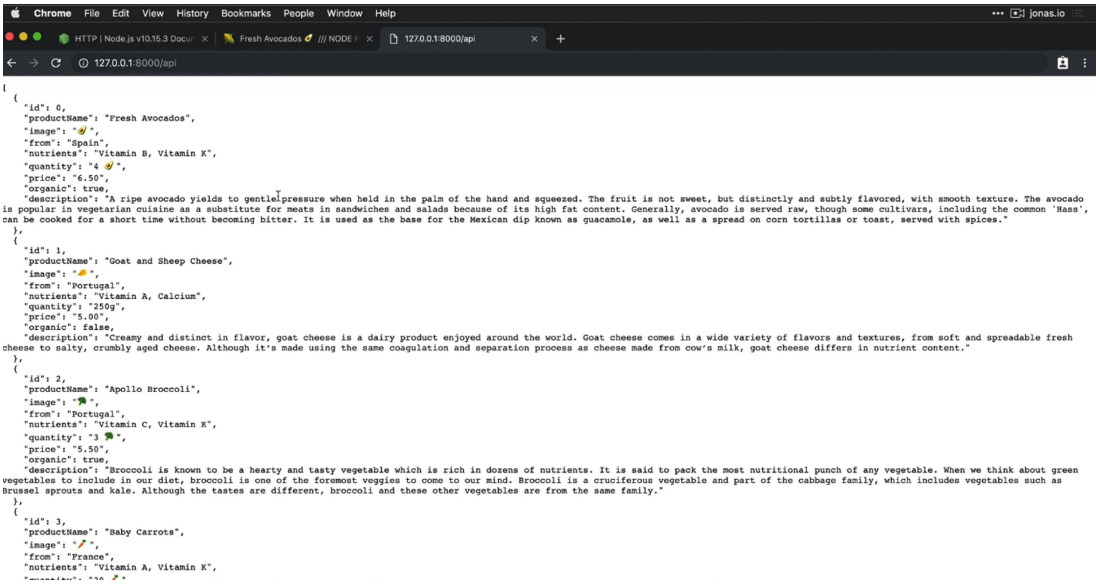


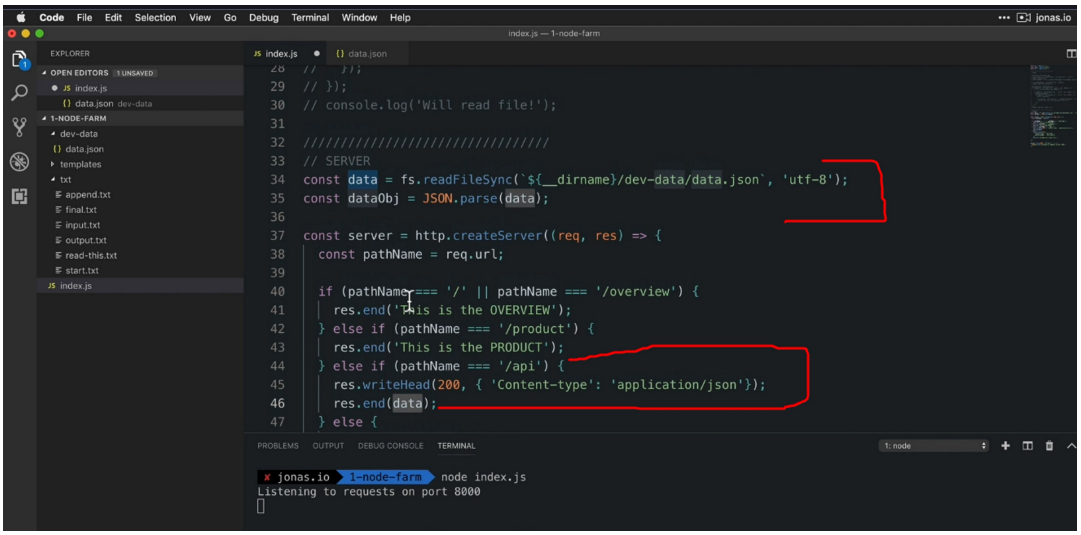




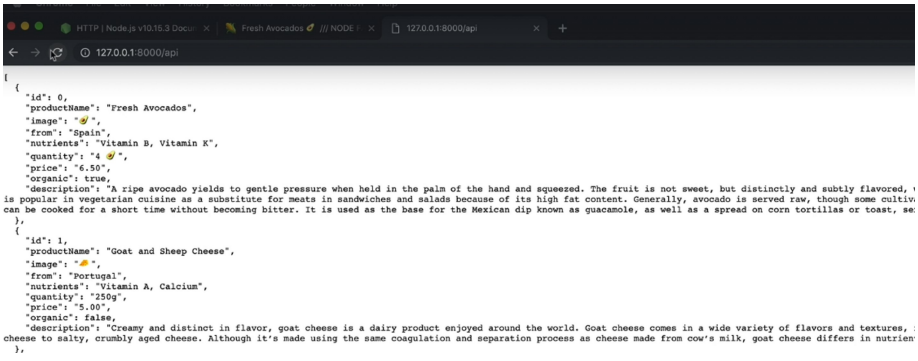


But our code is not efficient. We have to read the file every time once the api hits.

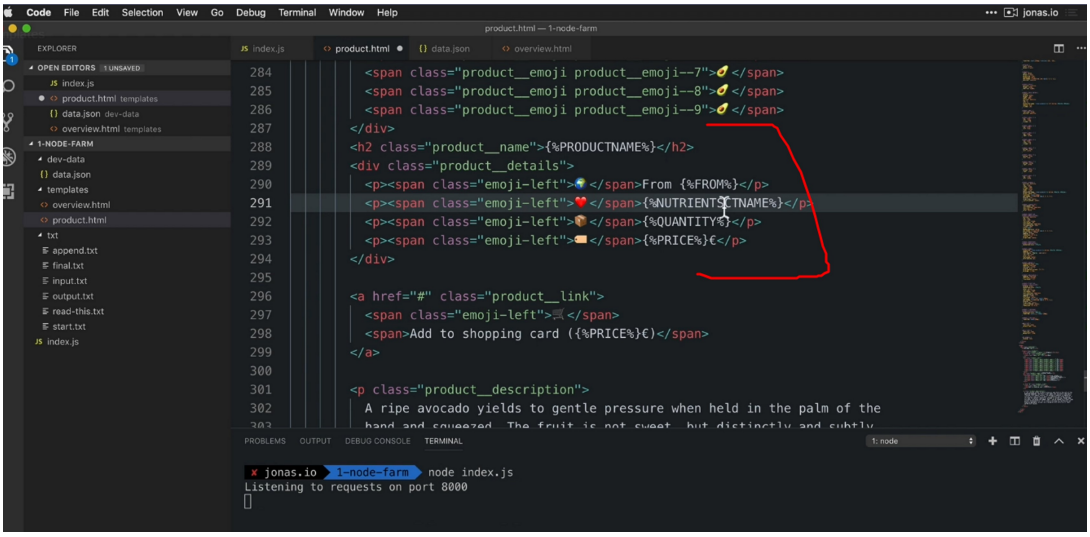


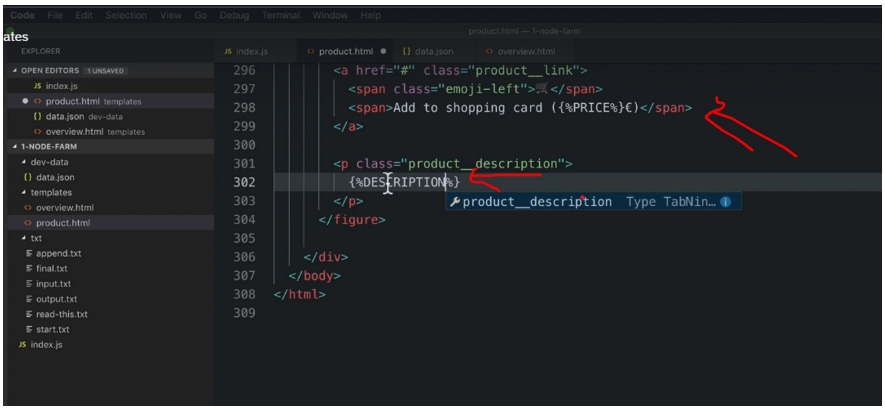


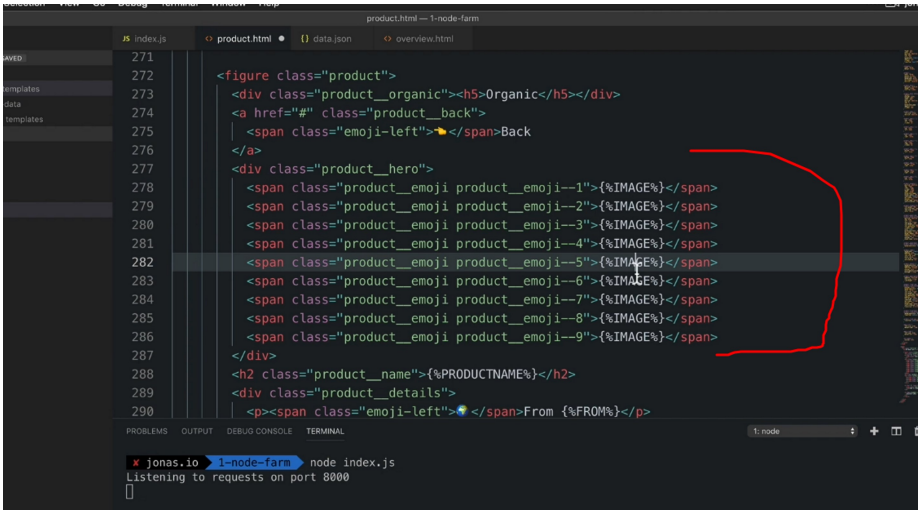
It works as before. But more efficient.



Let's start with 'product.html' file





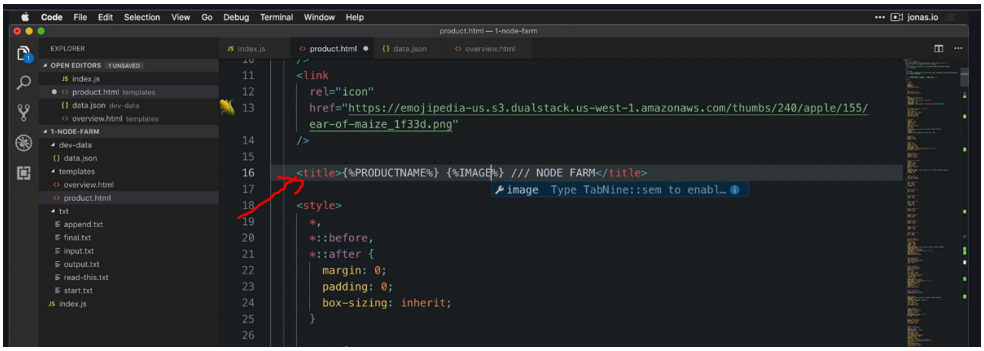

product.html — 1-node-farm
index.js
product.html
data.json
overview.html
<figure class="product">
<h5>Organic</h5>

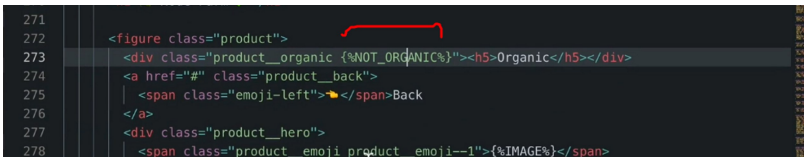
👈Back

{%IMAGE%}
{%IMAGE%}
{%IMAGE%}
{%IMAGE%}
{%IMAGE%}
{%IMAGE%}
{%IMAGE%}
{%IMAGE%}
{%IMAGE%}

<h2 class="product__name">{%PRODUCTNAME%}</h2>

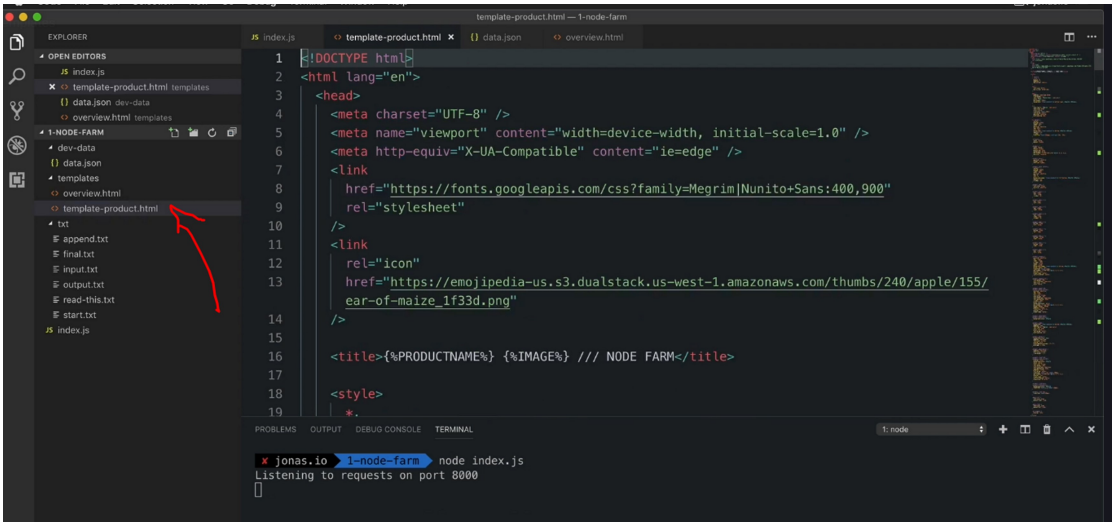
🌍From {%FROM%}
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
1: node
jonas.io
1-node-farm
node index.js
Listening to requests on port 8000

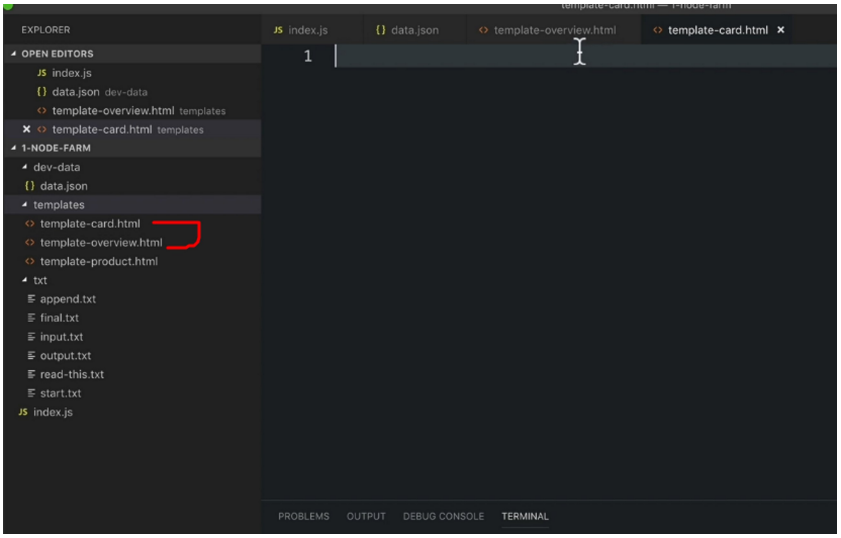

Code File Edit Selection View Go Debug Terminal Window Help
jonas.io
product.html — 1-node-farm
EXPLORER
OPEN EDITORS 1 UNSAVED
index.js
product.html templates
data.json dev-data
overview.html templates
1-NODE-FARM
dev-data
data.json
templates
overview.html
product.html
txt
append.txt
final.txt
input.txt
output.txt
read-this.txt
start.txt
index.js
<link
rel="icon"
href="https://emojipedia-us.s3.dualstack.us-west-1.amazonaws.com/thumbs/240/apple/155/ear-of-maize_1f33d.png"
/>
<title>{%PRODUCTNAME%} {%IMAGE%} /// NODE FARM</title>
image Type TabNine::sem to enabl…
<style>
*,
*::before,
*::after {
margin: 0;
padding: 0;
box-sizing: inherit;
}

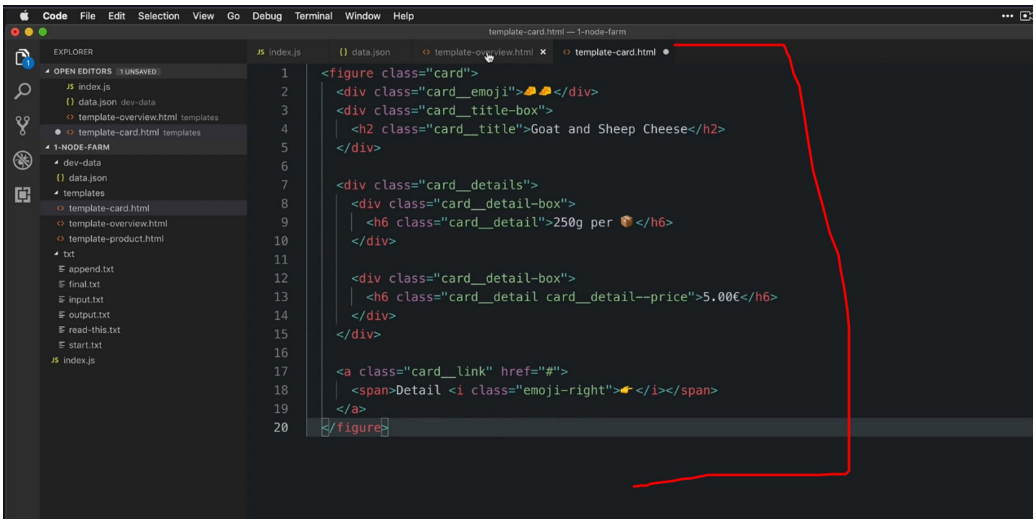

<figure class="product">
<h5>Organic</h5>

👈Back

{%IMAGE%}


template-product.html — 1-node-farm
EXPLORER
OPEN EDITORS
index.js
template-product.html templates
data.json dev-data
overview.html templates
1-NODE-FARM
dev-data
data.json
templates
overview.html
template-product.html
txt
append.txt
final.txt
input.txt
output.txt
read-this.txt
start.txt
index.js
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<meta http-equiv="X-UA-Compatible" content="ie=edge" />
<link
href="https://fonts.googleapis.com/css?family=Megrim|Nunito+Sans:400,900"
rel="stylesheet"
/>
<link
rel="icon"
href="https://emojipedia-us.s3.dualstack.us-west-1.amazonaws.com/thumbs/240/apple/155/ear-of-maize_1f33d.png"
/>
<title>{%PRODUCTNAME%} {%IMAGE%} /// NODE FARM</title>
<style>
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
1: node
jonas.io
1-node-farm
node index.js
Listening to requests on port 8000


EXPLORER
index.js
data.json
template-overview.html
template-card.html
OPEN EDITORS
index.js
data.json dev-data
template-overview.html templates
template-card.html templates
1-NODE-FARM
dev-data
data.json
templates
template-card.html
template-overview.html
template-product.html
txt
append.txt
final.txt
input.txt
output.txt
read-this.txt
start.txt
index.js
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL


Code File Edit Selection View Go Debug Terminal Window Help
template-card.html — 1-node-farm
<figure class="card">
🧀🧀

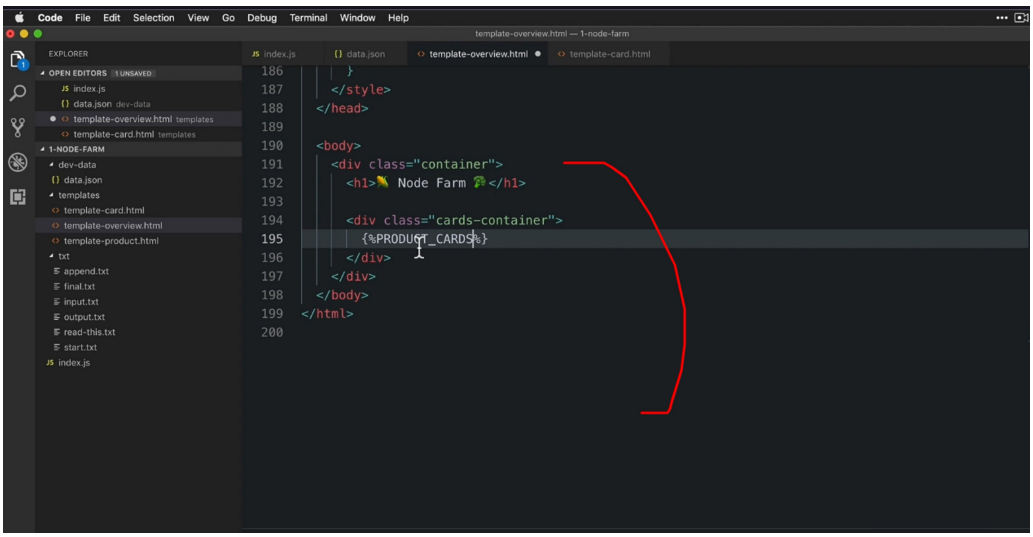
<h2 class="card__title">Goat and Sheep Cheese</h2>

<h6 class="card__detail">250g per 📦</h6>

<h6 class="card__detail card__detail--price">5.00€</h6>

Detail 👉

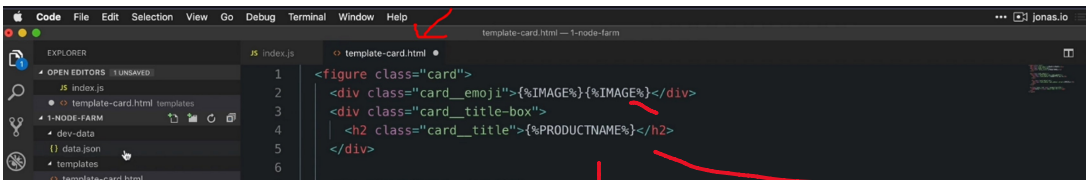
</figure>


Code File Edit Selection View Go Debug Terminal Window Help
template-overview.html — 1-node-farm
}
</style>
</head>
<body>

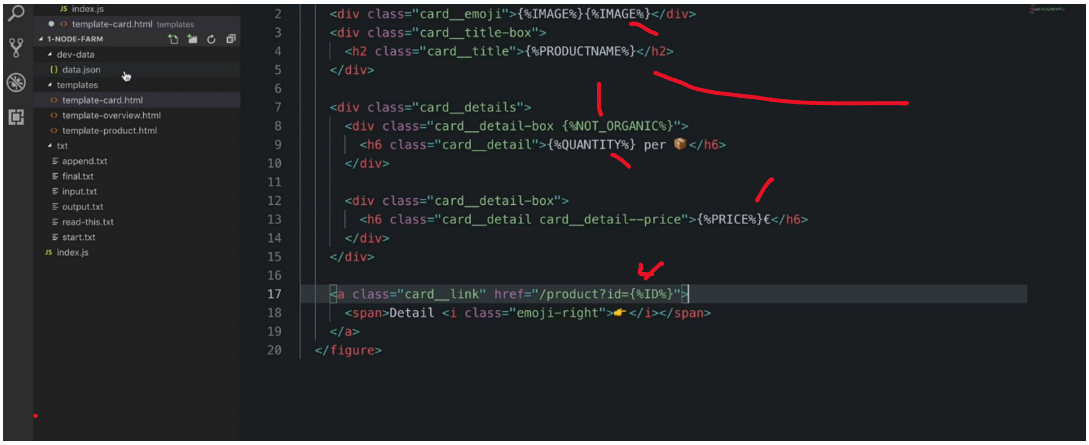
<h1>🌽 Node Farm 🌾</h1>

{%PRODUCT_CARDS%}

</body>
</html>

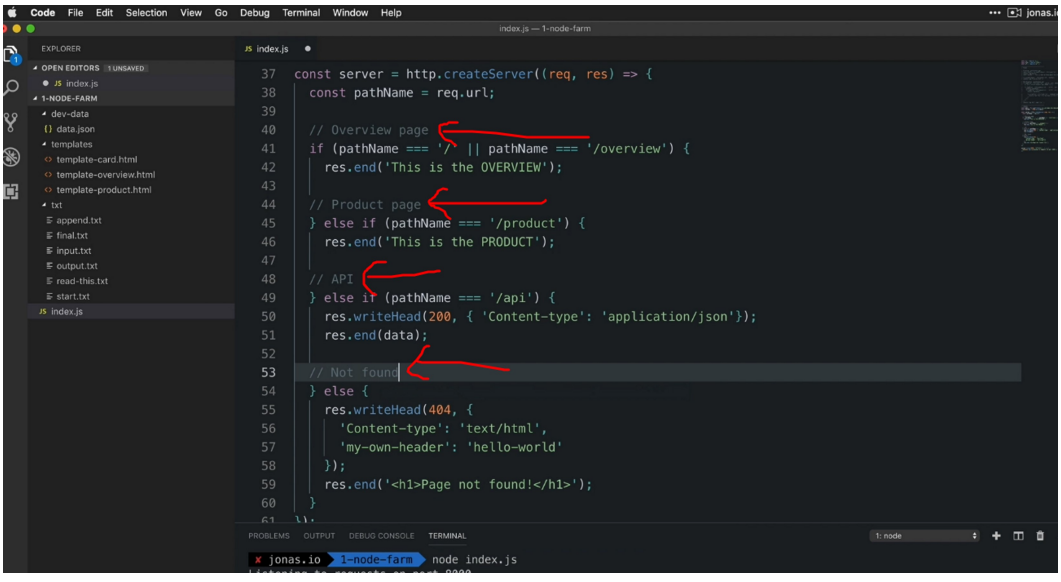

Code File Edit Selection View Go Debug Terminal Window Help
jonas.io
template-card.html — 1-node-farm
<figure class="card">
{%IMAGE%}{%IMAGE%}

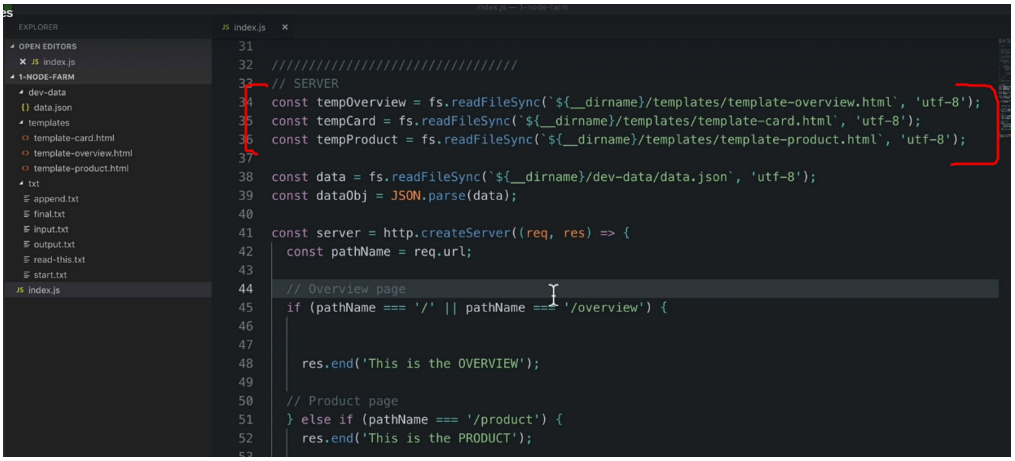
<h2 class="card__title">{%PRODUCTNAME%}</h2>



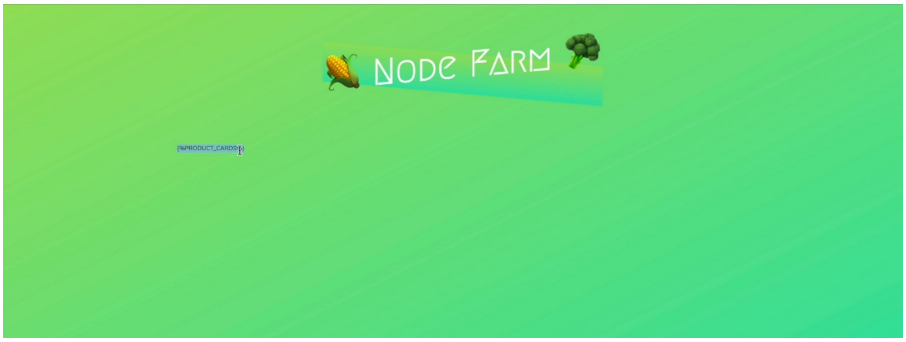
Let's replace the placeholder with the actual content.

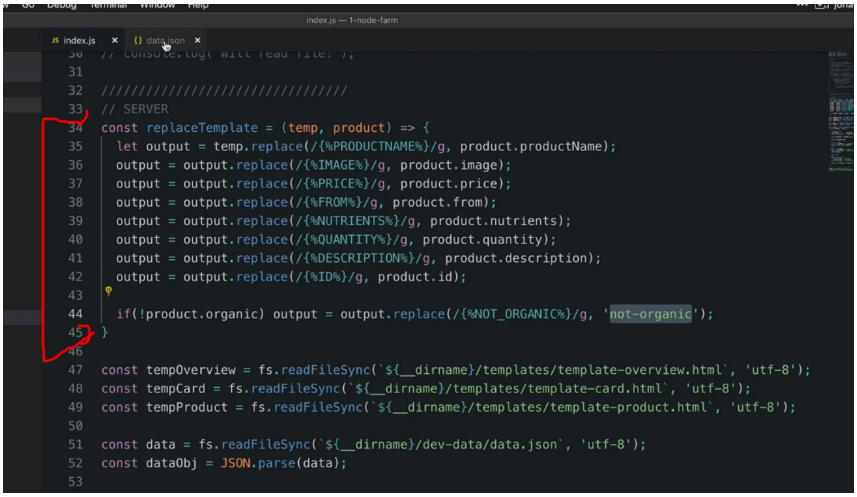
Let's add some comment



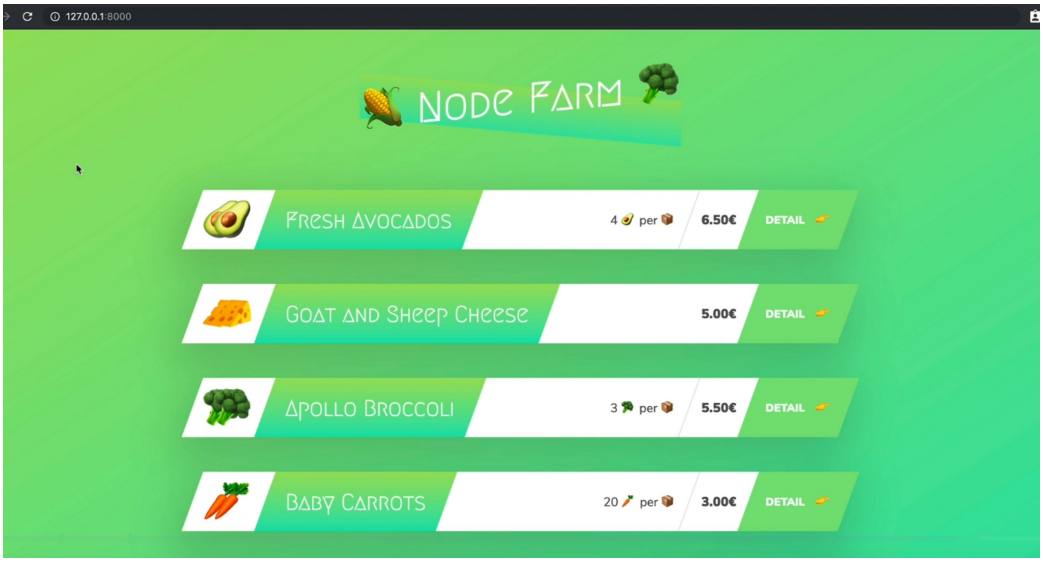


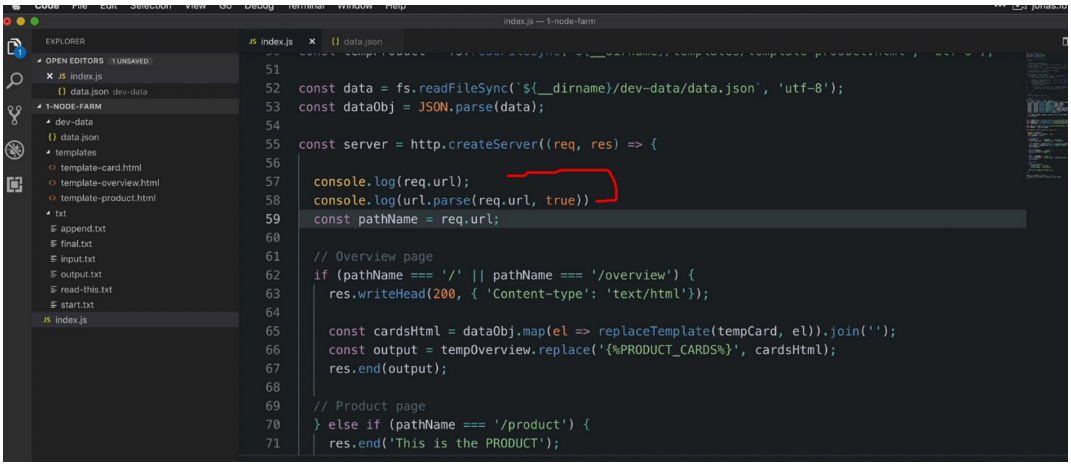
Restart the server again.





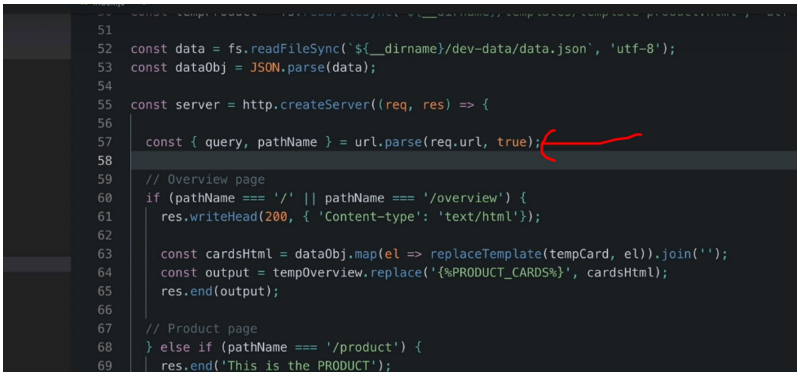
```
57
58  // Overview page
59  if (pathName === '/' || pathName === '/overview') {
60    res.writeHead(200, { 'Content-type': 'text/html'});
61
62    const cardsHtml = dataObj.map(el => replaceTemplate(tempCard, el)).join('');
63    const output = tempOverview.replace('{%PRODUCT_CARDS%}', cardsHtml);
64    res.end(output);
65
66  // Product page
67  } else if (pathName === '/product') {
68    res.end('This is the PRODUCT');
```



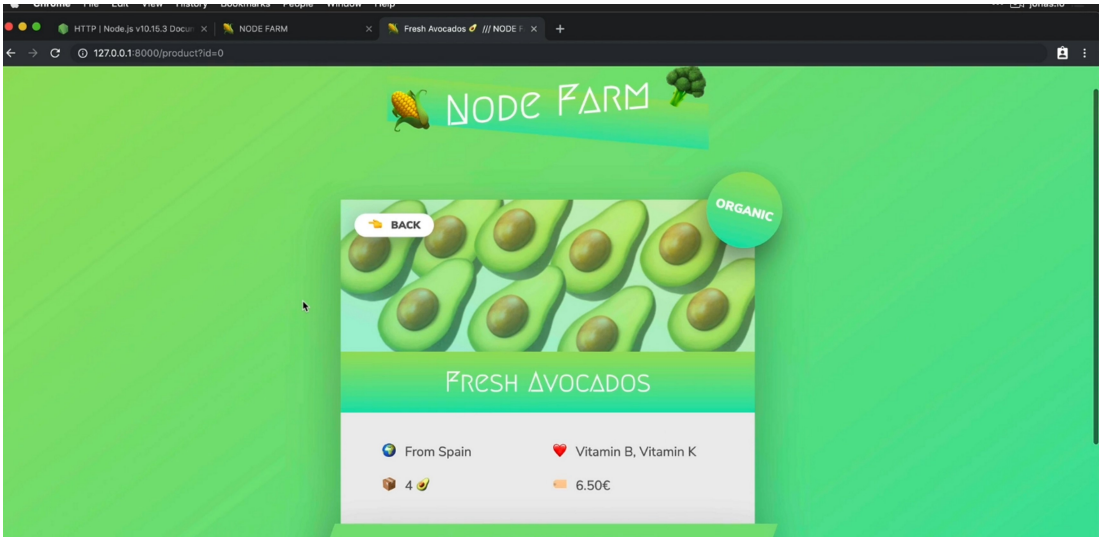

index.js — 1-node-farm
EXPLORER
OPEN EDITORS 1 UNSAVED
index.js
data.json dev-data
1-NODE-FARM
dev-data
data.json
templates
template-card.html
template-overview.html
template-product.html
txt
append.txt
final.txt
input.txt
output.txt
read-this.txt
start.txt
index.js
index.js
data.json
const data = fs.readFileSync(`${__dirname}/dev-data/data.json`, 'utf-8');
const dataObj = JSON.parse(data);
const server = http.createServer((req, res) => {
console.log(req.url);
console.log(url.parse(req.url, true))
const pathName = req.url;
// Overview page
if (pathName === '/' || pathName === '/overview') {
res.writeHead(200, { 'Content-type': 'text/html'});
const cardsHtml = dataObj.map(el => replaceTemplate(tempCard, el)).join('');
const output = tempOverview.replace('{%PRODUCT_CARDS%}', cardsHtml);
res.end(output);
// Product page
} else if (pathName === '/product') {
res.end('This is the PRODUCT');


127.0.0.1:8000/product?id=0
Page not found!


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
1: node
hostname: null,
hash: null,
search: '?id=0',
query: { id: '0' },
pathname: '/product',
path: '/product?id=0',
href: '/product?id=0' }
Ln 59, Col 28
Spaces: 2
UTF-8
LF
JavaScript
Prettier:


const data = fs.readFileSync(`${__dirname}/dev-data/data.json`, 'utf-8');
const dataObj = JSON.parse(data);
const server = http.createServer((req, res) => {
const { query, pathName } = url.parse(req.url, true);
// Overview page
if (pathName === '/' || pathName === '/overview') {
res.writeHead(200, { 'Content-type': 'text/html'});
const cardsHtml = dataObj.map(el => replaceTemplate(tempCard, el)).join('');
const output = tempOverview.replace('{%PRODUCT_CARDS%}', cardsHtml);
res.end(output);
// Product page
} else if (pathName === '/product') {
res.end('This is the PRODUCT');

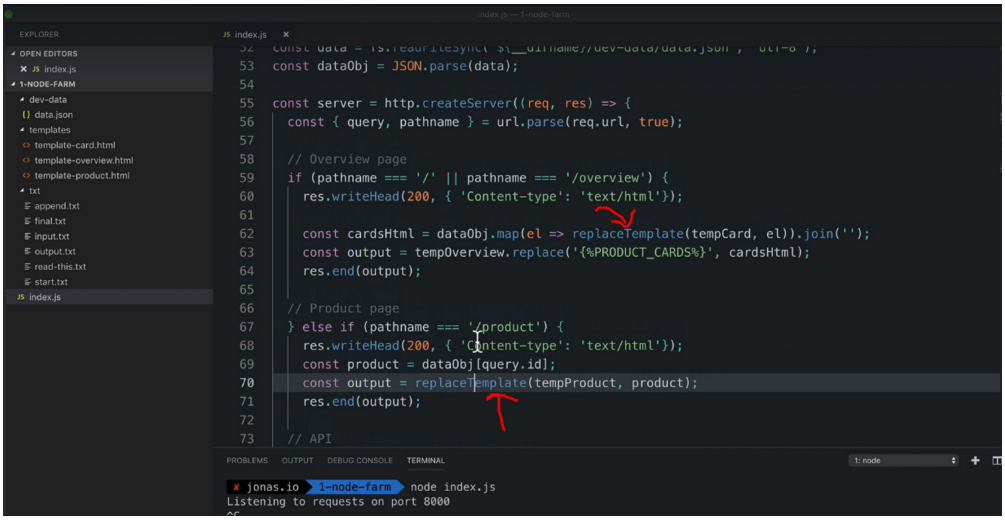

// Product page
} else if (pathname === '/product') {
res.writeHead(200, { 'Content-type': 'text/html'});
const product = dataObj[query.id];
const output = replaceTemplate(tempProduct, product);
res.end(output);



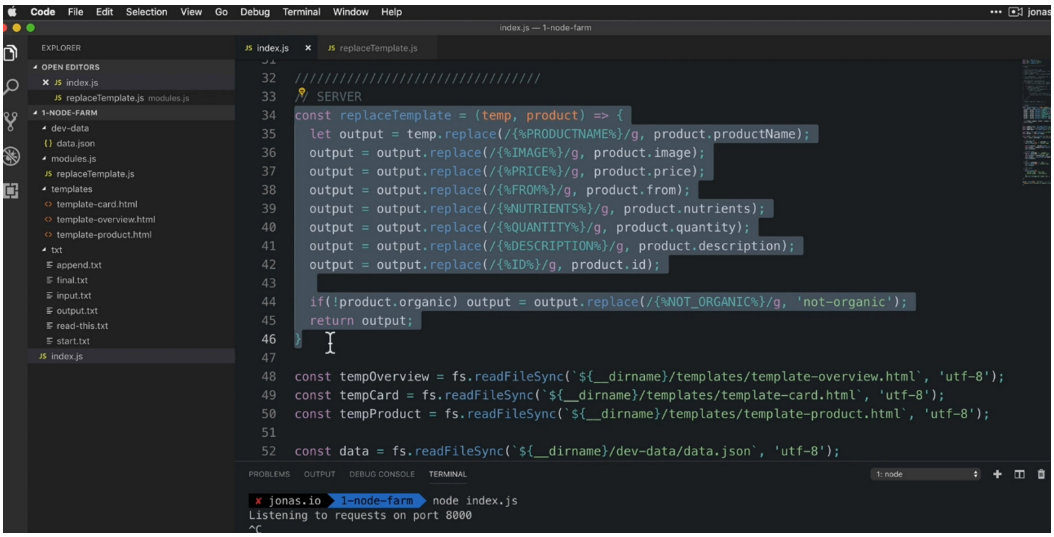
Okay, and so that is what we're gonna do in this lecture.
So let's say that we actually had
a bunch of different JavaScript files
in which we used this **replaceTemplate** function.

So right now we're just using it here in index.js.
We use it twice and that is why we have a function,
but imagine if we wanted to use this function in multiple files.
Okay, so what we can do is create a new module
and export that function from it
and then import it back here.

So, the first thing that you need to know
is that in Node.js, actually every single file is treated as a module.
And so this index.js here actually is also a module,
which in this case imports other modules—
and particularly these three.



Cut this function from 'index.js' file

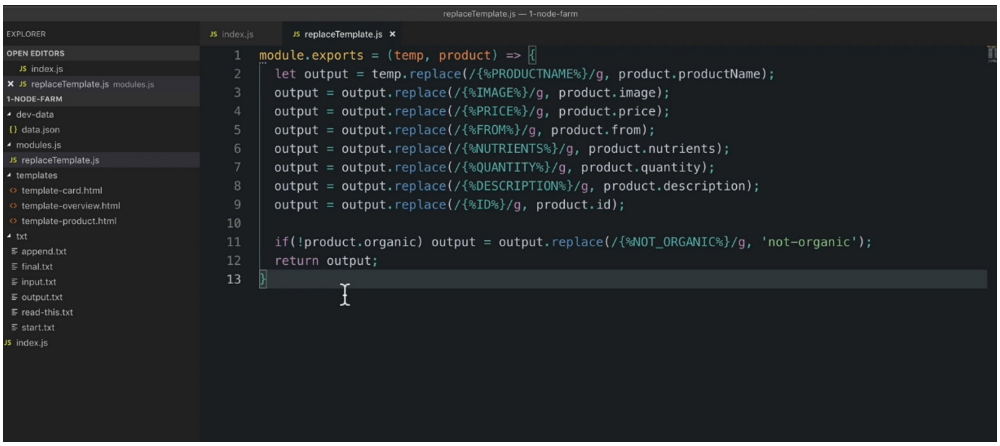


And paste it here.

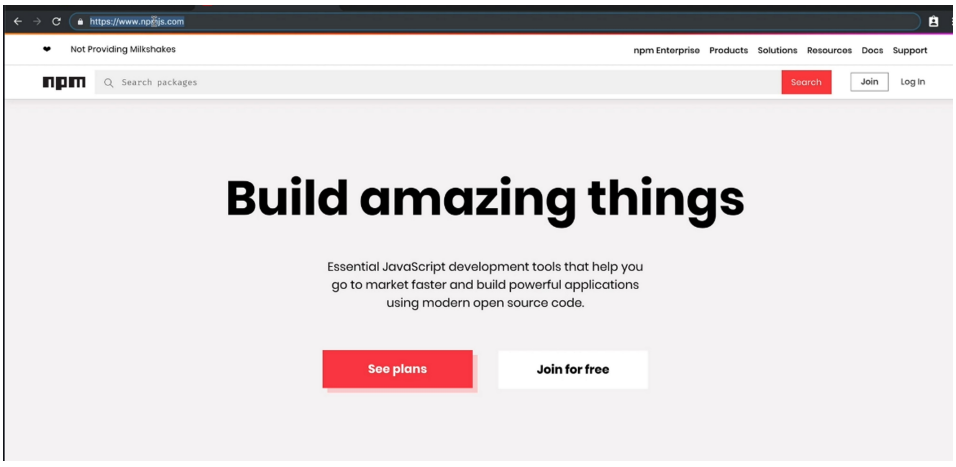
CREATE A FOLDER

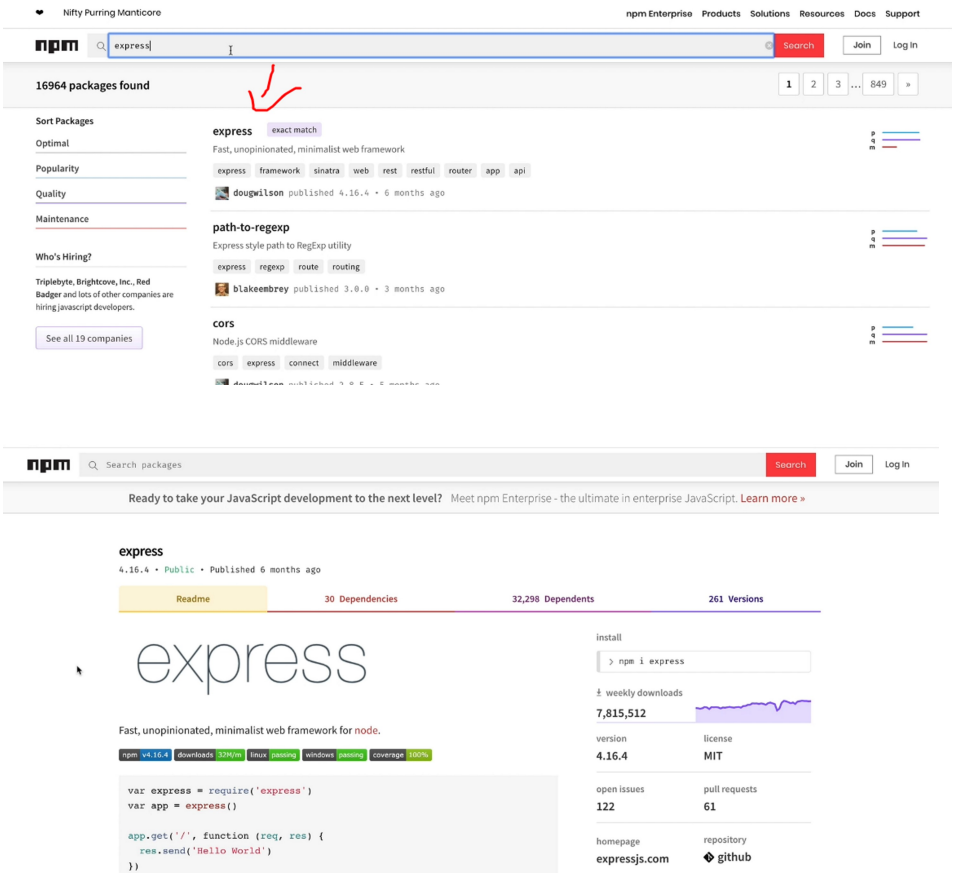
'module.js' -> replaceTemplate.js

Every node js module file has '**module**'



Now that you know the basics of Node.js,
let's talk about **npm** — the Node Package Manager.
npm is a command-line tool that comes with Node.js.
We use it to install and manage open-source packages.
These packages usually come from the **npm repository**,
which you can explore at npmjs.com.
npm is both the tool we use in the terminal
and the online registry that stores the packages.
It's the **largest software registry** in the world.
At the time of this recording,
there are around **800,000+ packages**,
and it's growing fast — soon hitting over a million.
For example, let's search for **Express**,
a Node.js framework we'll be using next.

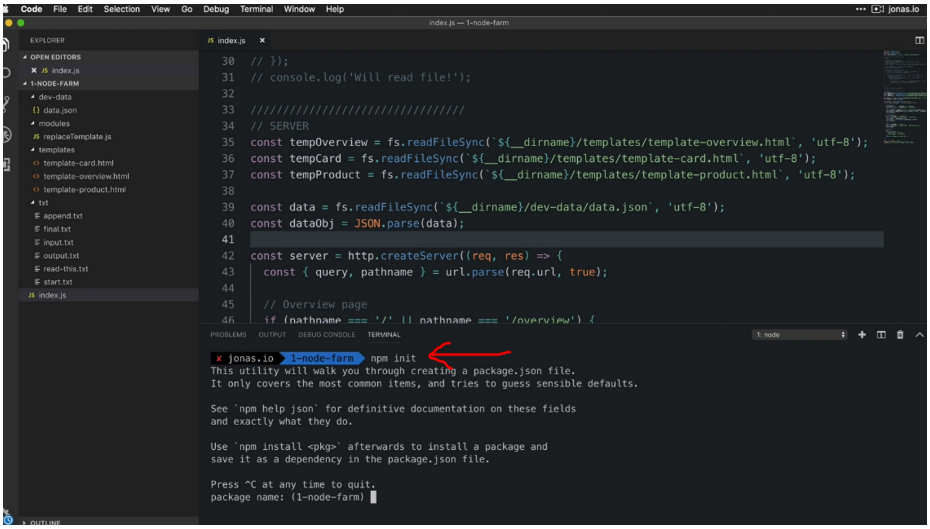




Whenever we start a new project, to start with , we use npm as the command then init.

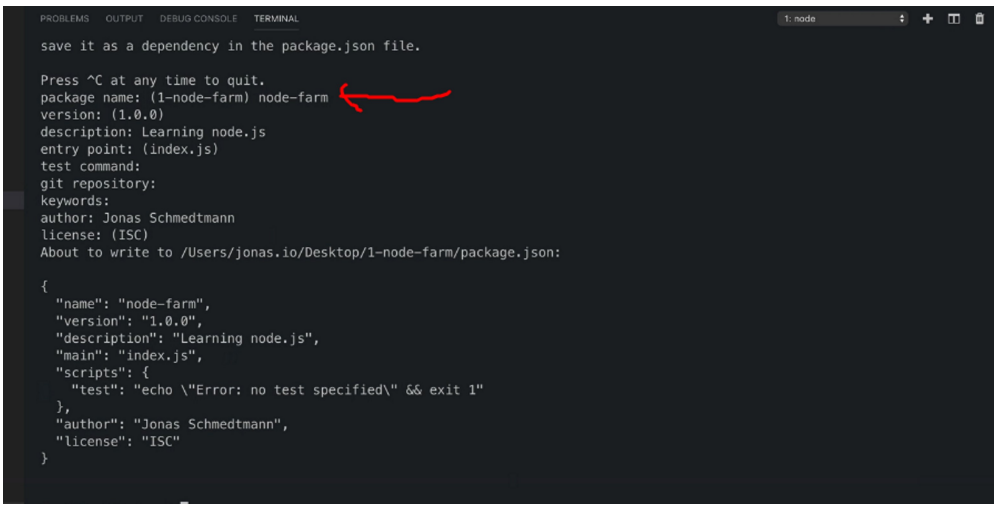
npm init

This will basically create a package.json file. Which is kind of configuration file of our project where all kinds of data about the project is stored.



It will ask me some question.

First name is package name.

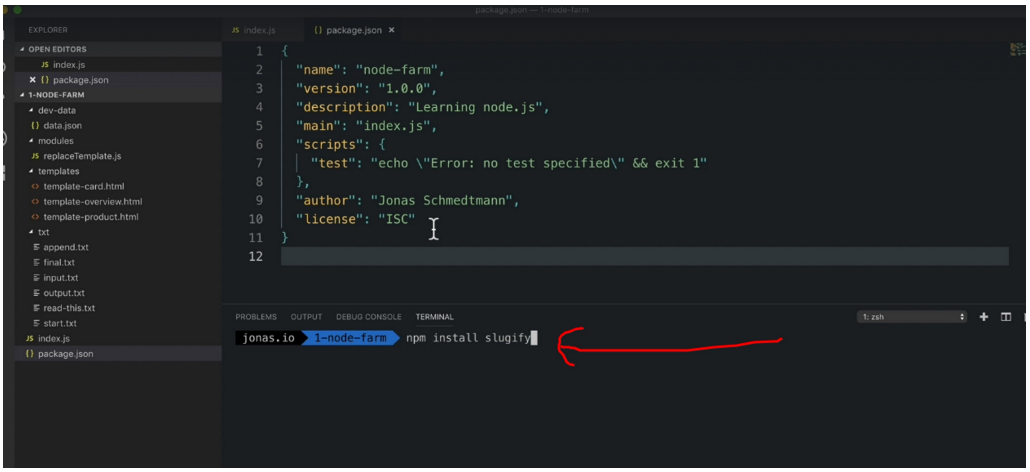


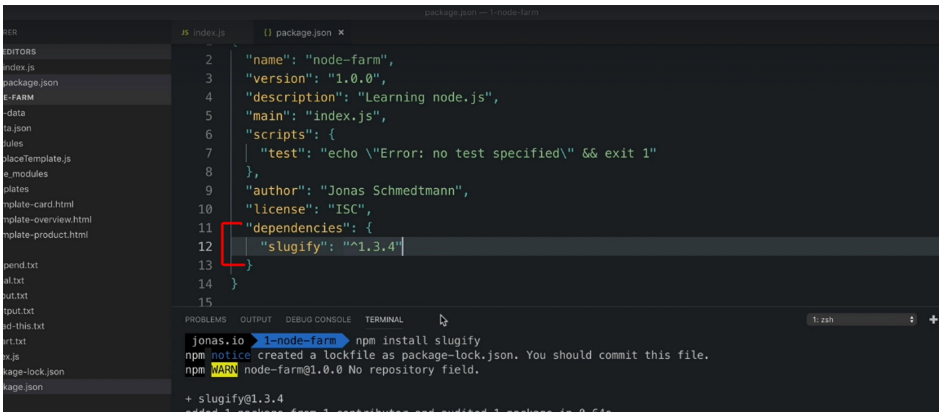
In this video, we'll look at **two types of npm packages** and **two ways to install them**.
The two types are:

Dependencies – Packages that your app needs to run (like Express).

Dev Dependencies – Packages only needed during development (like testing tools).
Let's install our first dependency: **Slugify**.

It's a small tool that turns names into URL-friendly slugs —
perfect for things like product names in our Node Farm project.





Besides these regular dependencies, we also have development dependencies.

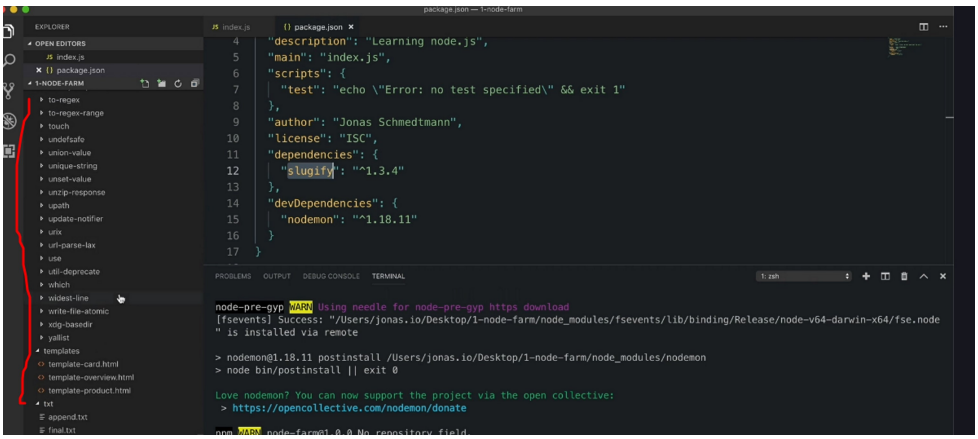
And these are usually tool for development. For example, a code bundler like WebPack, or a debugger tool or a testing library. So these are development dependencies. They are needed for production, so our codes does not really depend on them, we simply use them to develop our application.

--save-dev => defines it is a development dependencies.

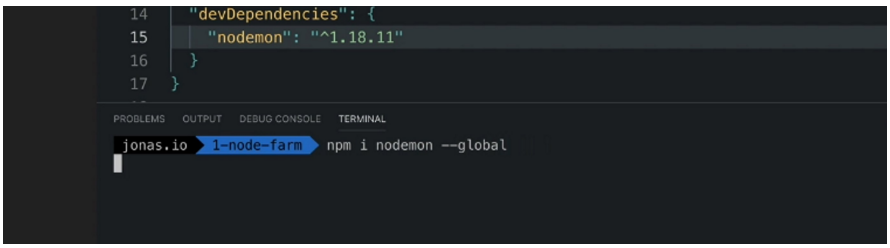
Nodemon is a very nice tool that helps us develop NodeJS application whenever we change some files in our working directory.

You might wonder why we see so many packages when we only installed **Slugify** and **nodemon**.
That's because these packages have their own dependencies.

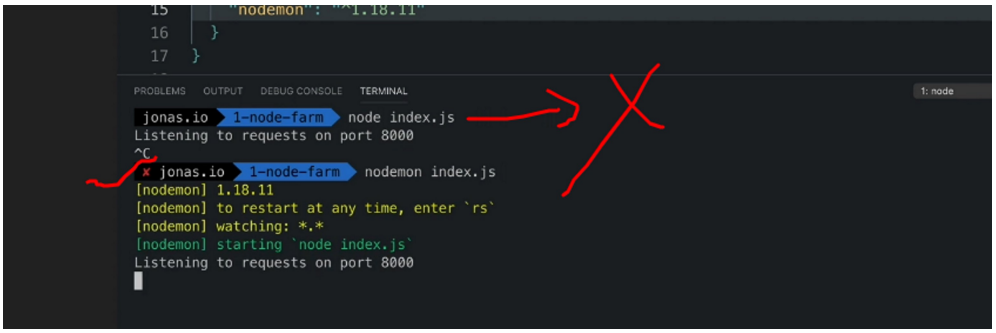
For example, if **Slugify** needs another package, npm installs it too.
That's how we end up with a long list of dependencies in our project.



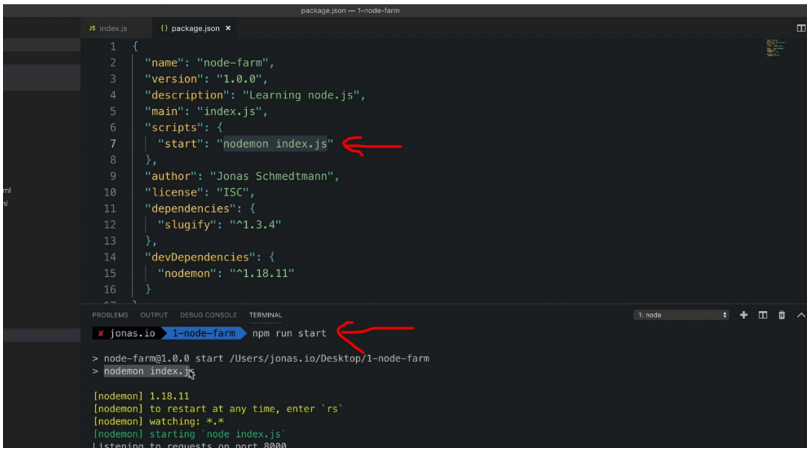
We installed the packages locally, so they only work in this project.
But with **npm**, we can also do **global installs**, which work anywhere on your machine.
Global installs are for tools that you run from the command line — like **nodemon**,
which I use in all my Node projects.



nodemon node index.js

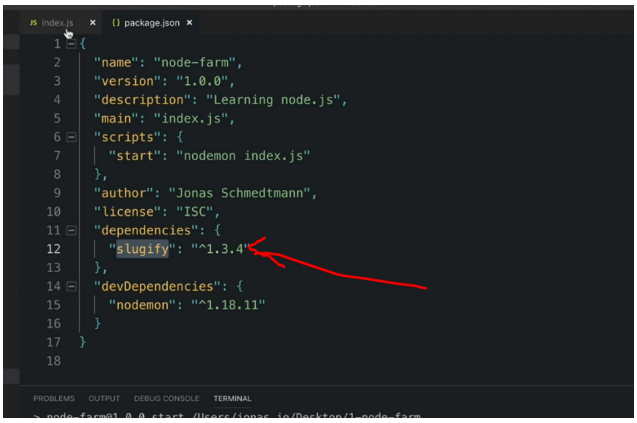


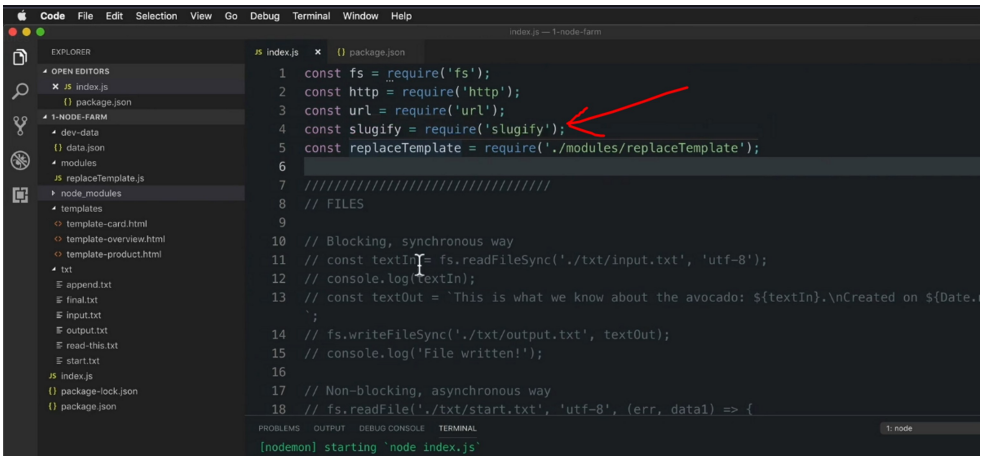
What if we don't want to install nodemon globally. Just locally ?

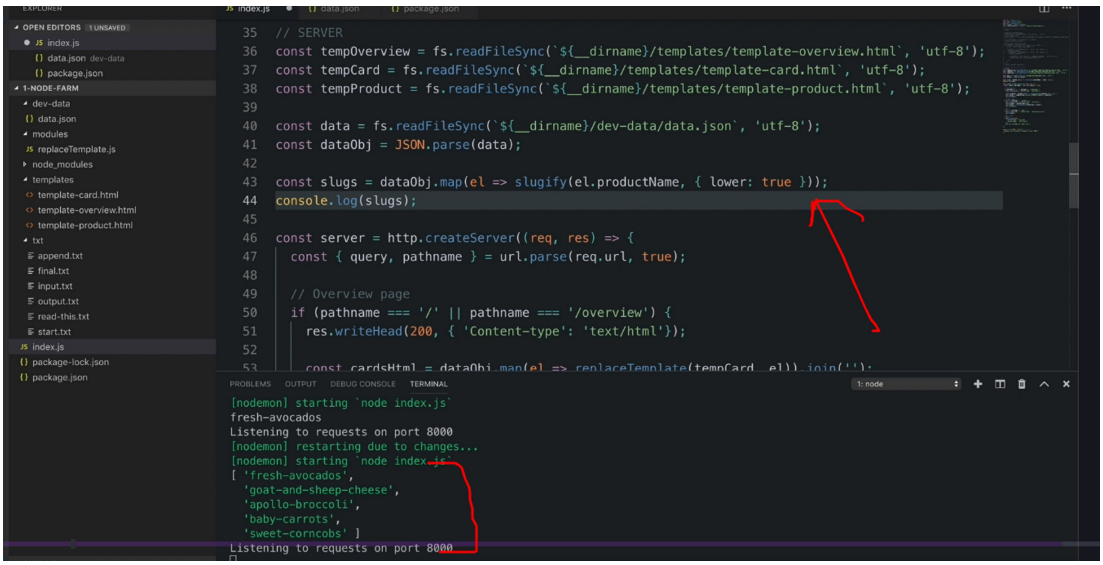

package.json — 1-node-farm
index.js
package.json
{
"name": "node-farm",
"version": "1.0.0",
"description": "Learning node.js",
"main": "index.js",
"scripts": {
"start": "nodemon index.js"
},
"author": "Jonas Schmedtmann",
"license": "ISC",
"dependencies": {
"slugify": "^1.3.4"
},
"devDependencies": {
"nodemon": "^1.18.11"
}
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
jonas.io
1-node-farm
npm run start
> node-farm@1.0.0 start /Users/jonas.io/Desktop/1-node-farm
> nodemon index.js
[nodemon] 1.18.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`

Or 'npm start'

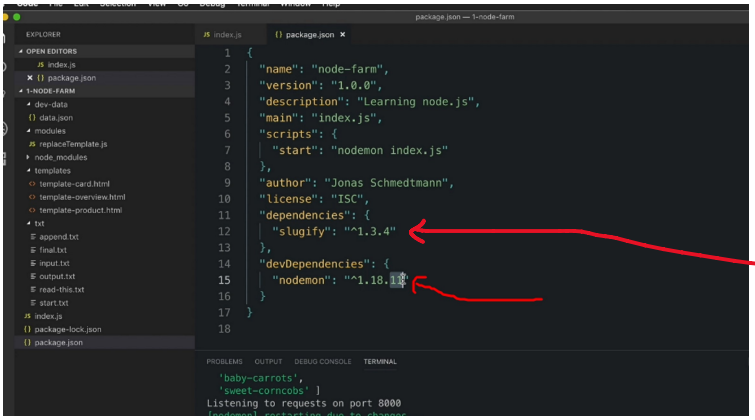

JONAS.IO
SCHMEDTMANN
NODE.JS, EXPRESS & MONGODB
THE COMPLETE BOOTCAMP
SECTION
INTRODUCTION TO NODE.JS
LECTURE
USING MODULES 3: 3RD PARTY MODULES


index.js
package.json
{
"name": "node-farm",
"version": "1.0.0",
"description": "Learning node.js",
"main": "index.js",
"scripts": {
"start": "nodemon index.js"
},
"author": "Jonas Schmedtmann",
"license": "ISC",
"dependencies": {
"slugify": "^1.3.4"
},
"devDependencies": {
"nodemon": "^1.18.11"
}
}
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL


Code File Edit Selection View Go Debug Terminal Window Help
index.js — 1-node-farm
EXPLORER
OPEN EDITORS
index.js
package.json
1-NODE-FARM
dev-data
data.json
modules
replaceTemplate.js
node_modules
templates
template-card.html
template-overview.html
template-product.html
txt
append.txt
final.txt
input.txt
output.txt
read-this.txt
start.txt
index.js
package-lock.json
package.json
index.js
package.json
1 const fs = require('fs');
2 const http = require('http');
3 const url = require('url');
4 const slugify = require('slugify');
5 const replaceTemplate = require('./modules/replaceTemplate');
6
7 ////////////////////////////////
8 // FILES
9
10 // Blocking, synchronous way
11 // const textIn = fs.readFileSync('./txt/input.txt', 'utf-8');
12 // console.log(textIn);
13 // const textOut = `This is what we know about the avocado: ${textIn}.\nCreated on ${Date.
`;
14 // fs.writeFileSync('./txt/output.txt', textOut);
15 // console.log('File written!');
16
17 // Non-blocking, asynchronous way
18 // fs.readFile('./txt/start.txt', 'utf-8', (err, data1) => {
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: node
[nodemon] starting `node index.js`


// SERVER
const tempOverview = fs.readFileSync(`${__dirname}/templates/template-overview.html`, 'utf-8');
const tempCard = fs.readFileSync(`${__dirname}/templates/template-card.html`, 'utf-8');
const tempProduct = fs.readFileSync(`${__dirname}/templates/template-product.html`, 'utf-8');
const data = fs.readFileSync(`${__dirname}/dev-data/data.json`, 'utf-8');
const dataObj = JSON.parse(data);
const slugs = dataObj.map(el => slugify(el.productName, { lower: true }));
console.log(slugs);
const server = http.createServer((req, res) => {
const { query, pathname } = url.parse(req.url, true);
// Overview page
if (pathname === '/' || pathname === '/overview') {
res.writeHead(200, { 'Content-type': 'text/html'});
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: node
[nodemon] starting `node index.js`
fresh-avocados
Listening to requests on port 8000
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
['fresh-avocados',
'goat-and-sheep-cheese',
'apollo-broccoli',
'baby-carrots',
'sweet-corncobs']
Listening to requests on port 8000

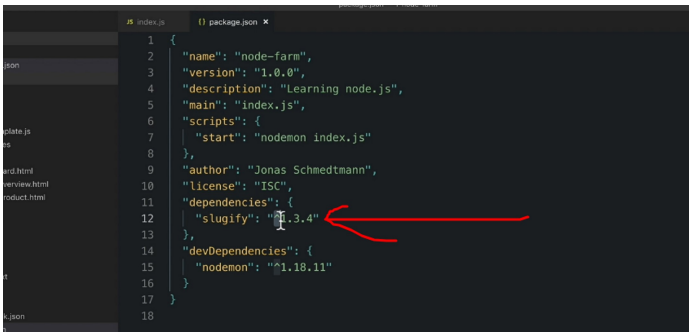

JONAS.IO
SCHMEDTMANN
NODE.JS, EXPRESS & MONGODB
THE COMPLETE BOOTCAMP
SECTION
INTRODUCTION TO NODE.JS
LECTURE
PACKAGE VERSIONING AND UPDATING



We use **semantic versioning**:

- **MAJOR.MINOR.PATCH**
- **Patch** (last number): bug fixes only (e.g. 1.18 → 1.18.1 → 1.18.2…)
- **Minor** (middle): new, backward-compatible features (e.g. 1.18 → 1.19)
- **Major** (first): big releases that can break stuff (e.g. 1.x → 2.0)

So if you see Slugify go from 1.4 to 2.0, watch out—your code might need updates. 🚀



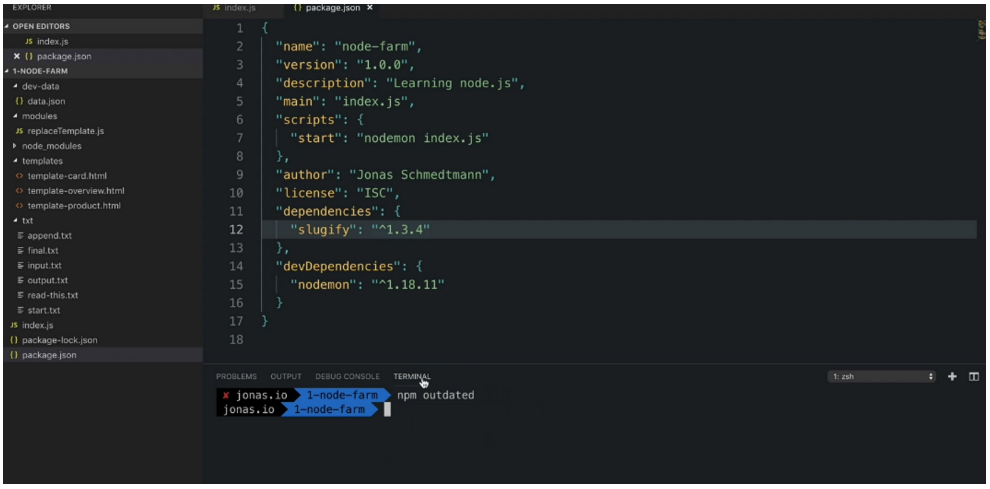
^ =>

Because of that, it is also important
to talk about updating packages.
In our package.json file,
this small symbol (^) here that comes in front
of the version number is what specifies
which updates we accept for each of the packages.
This symbol here, which npm specifies here by default
means that we accept patch and minor releases.

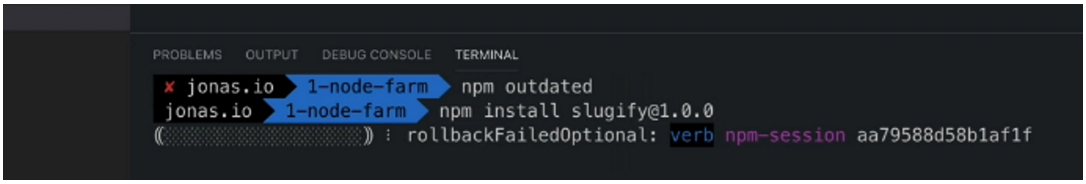
How to check if there are any outdated packages?

npm outdated

Right now they are all up to date.

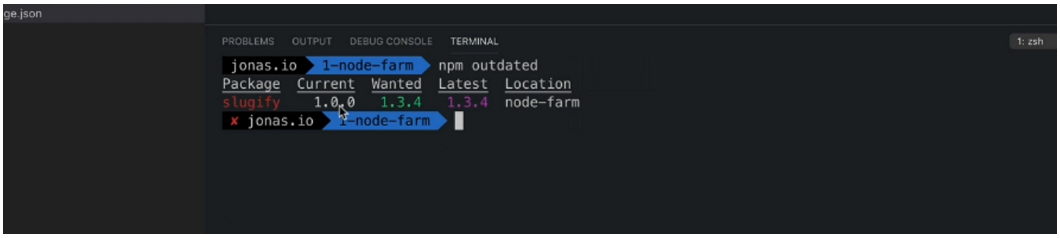


Let's now see how to install a certain package with a certain version number?



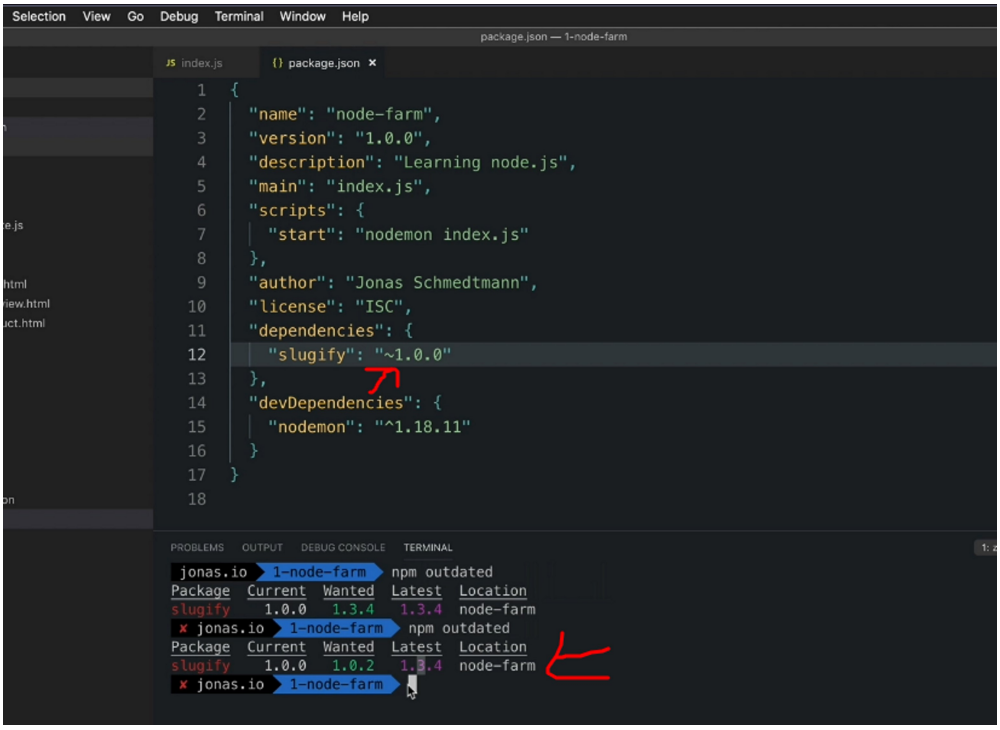
If we now run

npm outdated



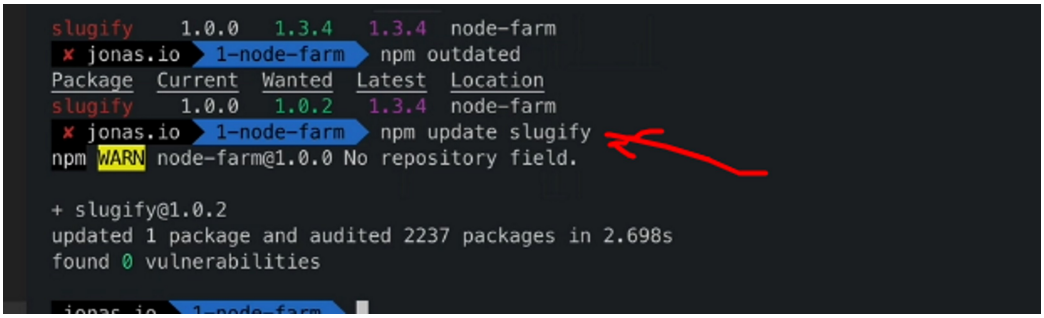
^ => we accept all the (patch+minor) releases
~ => we accept (patch) releases

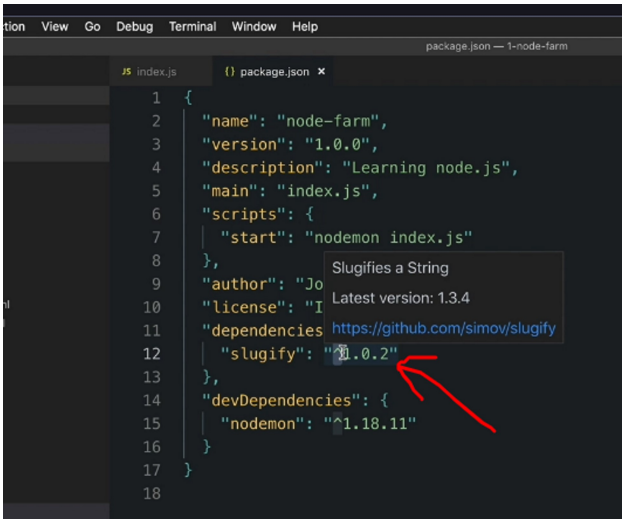
Change in the 'package.json' file and run 'npm outdated' again.



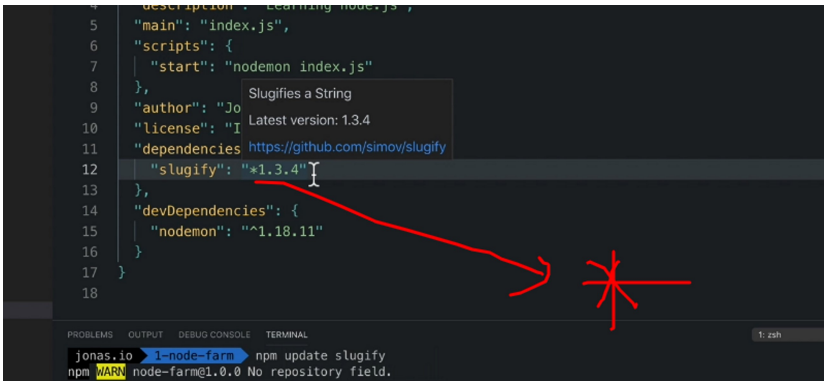
See the 'wanted' column. It's 1.0.2 version that is accepted.

Run 'npm update slugify' command.





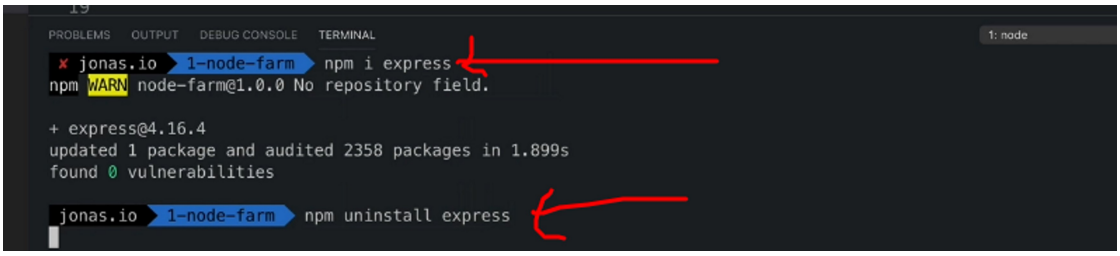
If there is "*" and we update package that means it will be updated to the latest version.



Safest option is to use ~

We can also delete packages.

Lets now install 'express' so we can delete



NEVER SHARE 'NODE_MODULES' FOLDER TO GITHUB, OR ANY WHERE.

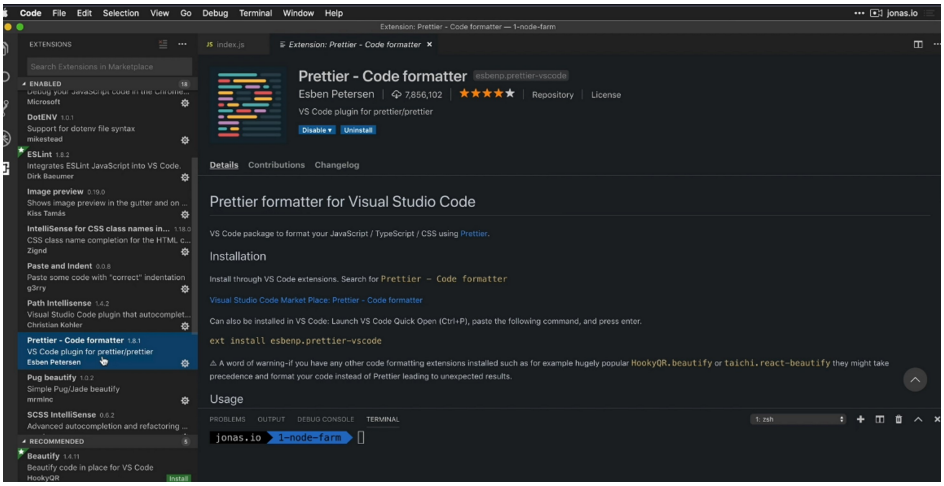
You can get it by pressing 'npm i'

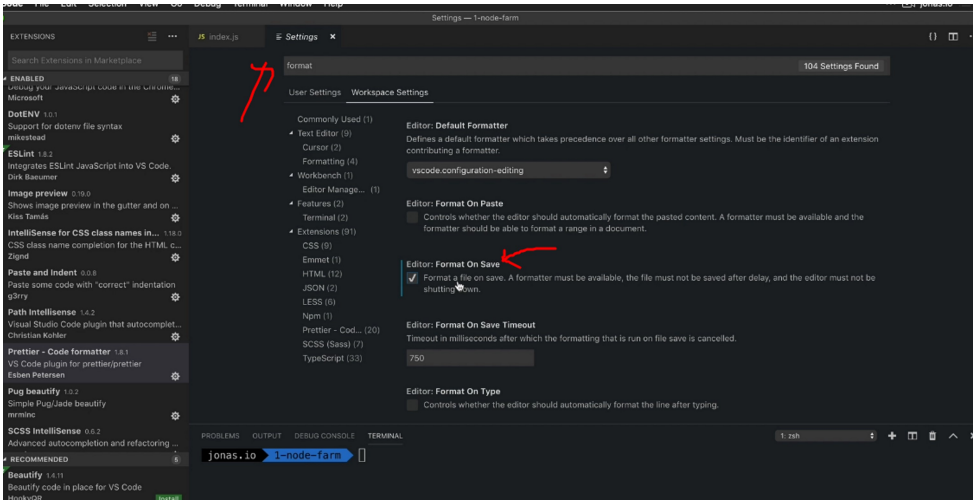
Basically it reads our 'package.json' file and install corresponding dependencies.

Now one important piece of this puzzle
is the package-lock.json file.
If we open that up, we get a list of
all the versions of all the packages
that we're actually using.

That includes the dependencies of our dependencies.
Let's go to Slugify, for example.
Slugify, and so here we see that
we're using version 1.3.4.

That is very important because
if you share your code, you want the other developer
or even yourself to be using
the exact same package versions.

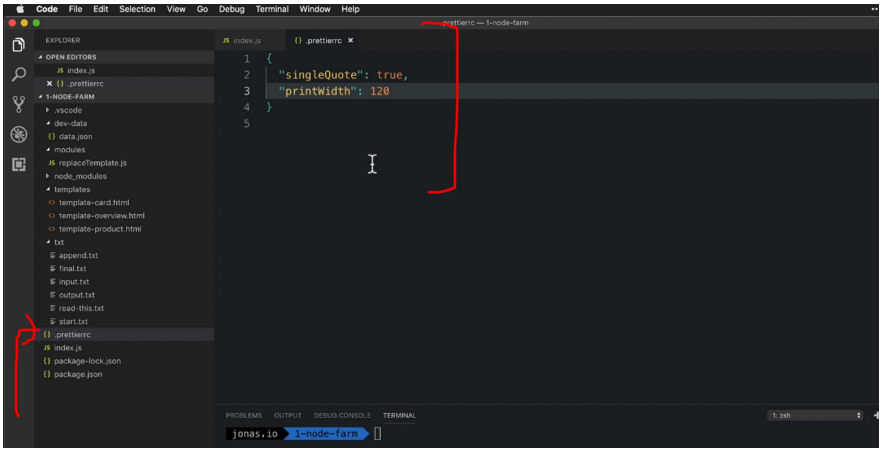




Now we can configure prettier now. Let's define a configuration file.

We also can do in the vs code. But we prefer to create a separate file.

(Helpful for working in a team)


Code File Edit Selection View Go Debug Terminal Window Help
.prettierrc — 1-node-farm
EXPLORER
OPEN EDITORS
index.js
.prettierrc
1-NODE-FARM
.vscode
dev-data
data.json
modules
replaceTemplate.js
node_modules
templates
template-card.html
template-overview.html
template-product.html
txt
append.txt
final.txt
input.txt
output.txt
read-this.txt
start.txt
.prettierrc
index.js
package-lock.json
package.json
{
"singleQuote": true,
"printWidth": 120
}
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: zsh
jonas.io 1-node-farm


JONAS.IO
SCHMEDTMANN
NODE.JS, EXPRESS & MONGODB
THE COMPLETE BOOTCAMP
SECTION
INTRODUCTION TO NODE.JS
LECTURE
RECAP AND WHAT'S NEXT