

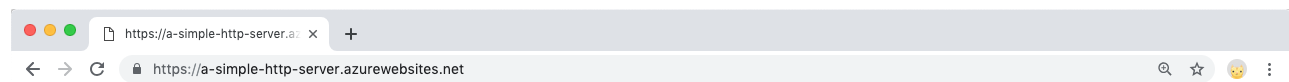
Azure Casts - 004

Hey Friends

In this video, we're going to look at how Azure actually runs Node applications, and the different ways that you can instruct it to run yours.

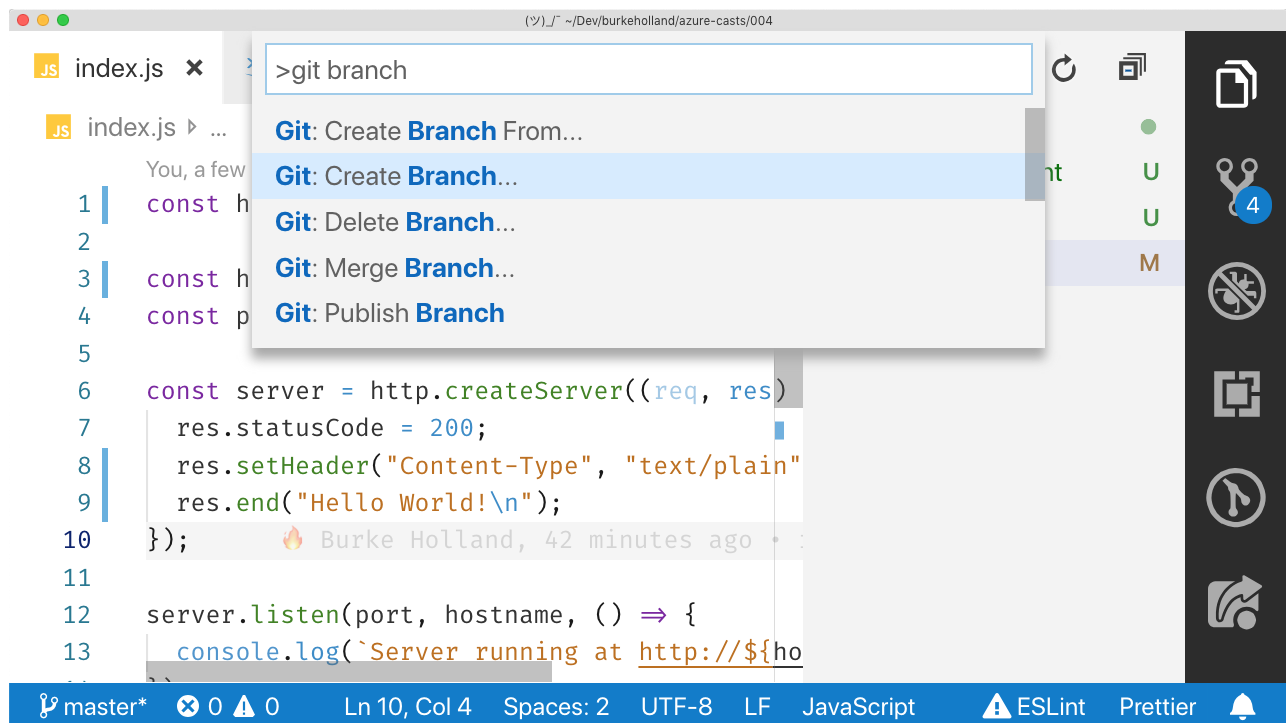
I'm a huge VS Code fan, so I use the App Service extension for VS Code when I'm working with Azure. You can get that from the VS Code extension gallery.

I've got an extremely simple Node app that is actually just the Node.js getting started guide code. I've got it deployed on Azure and it's running like a champ. You can check out episode 2 if you want to see how I did this miracle of coding simplicity.



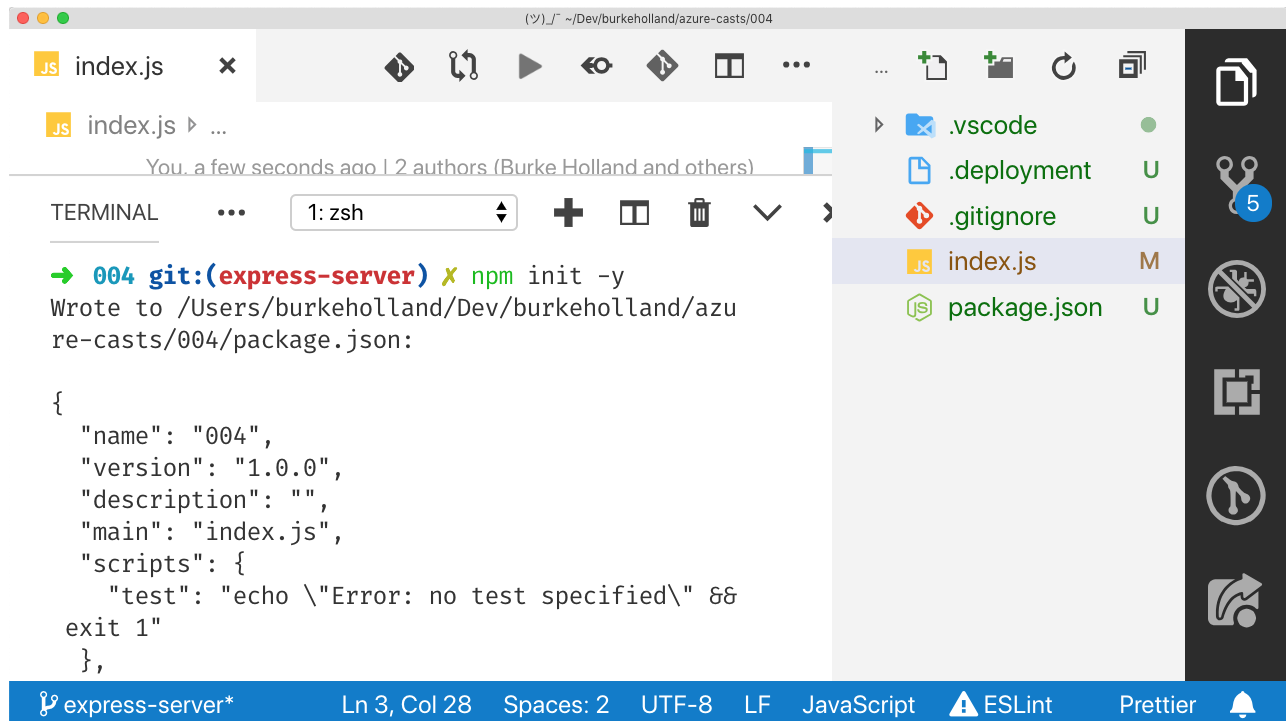
Hello World!

Let's upgrade this app a bit by adding in the Express web server package and serving up an actual HTML file. I'm going to create a new Git branch to do that since this is a pretty big change.



We can install the Express package from npm. Before we do that, let's create a package.json file. Every Node project should have one of those. We can do that with npm init. And here's a neat trick that Elijah Manor taught me - if you don't want have to answer all these questions here, just pass the -y flag

```
npm init -y
```



Now let's install Express...

```
npm install express
```

You can also just use the "i" shortcut and do...

```
npm i express
```

I'm going to change out the code in the `index.js` file to be a simple Express server that just returns an `index.html` file.

```
const express = require("express");
const app = express();
const path = require("path");
const port = process.env.PORT || 3000;

// viewed at http://localhost:3000
```

```
app.get("/", function(req, res) {
  res.sendFile(path.join(__dirname + "/index.html"));
});

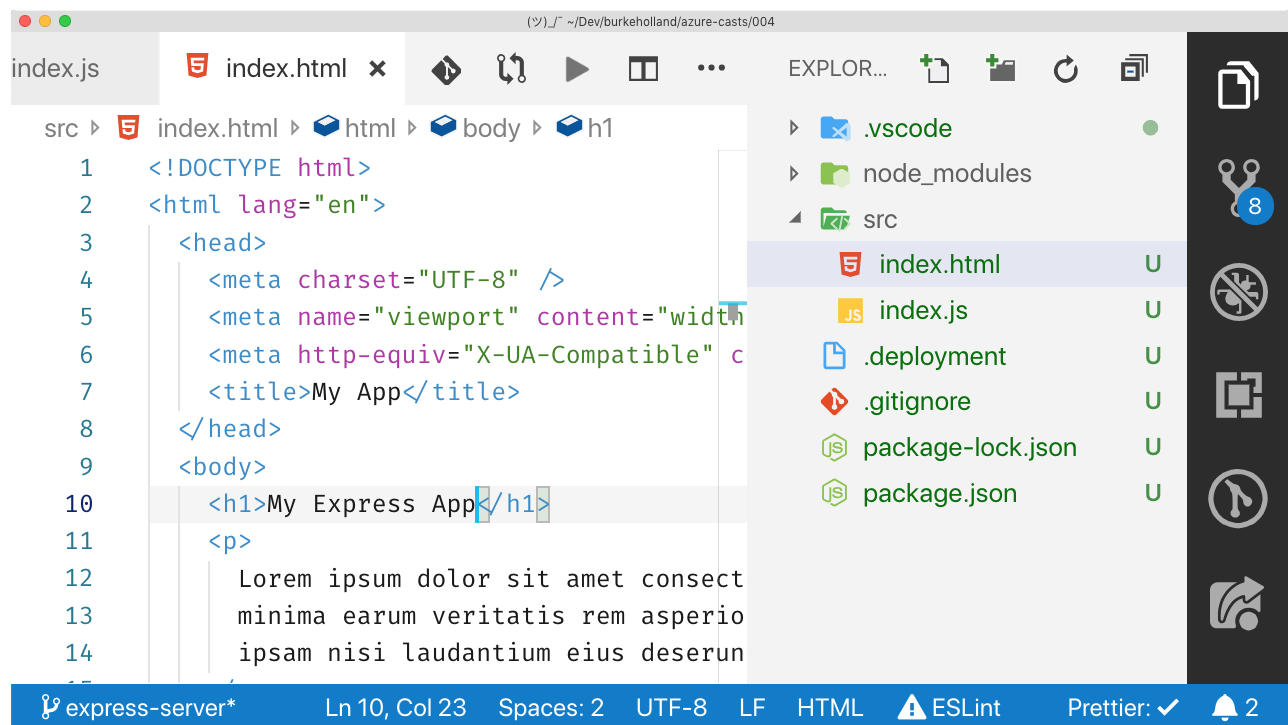
app.listen(port, () => console.log(`App listening on port ${port}!`));
```

And let's add an `index.html` file for Express to return. You can use the built-in Emmet support in VS Code to quickly scaffold out an HTML file by just typing an exclamation mark - or a bang if you will and then hitting tab. Then you can tab through all of the tab stops until you get to the body. Nifty. Emmet is a super quick way to create HTML. Let's add an H1 and return a title. And let's create a p tag with some lorem ipsum text. And emmet helps us create that lorem ipsum text too. Emmet is so neat.

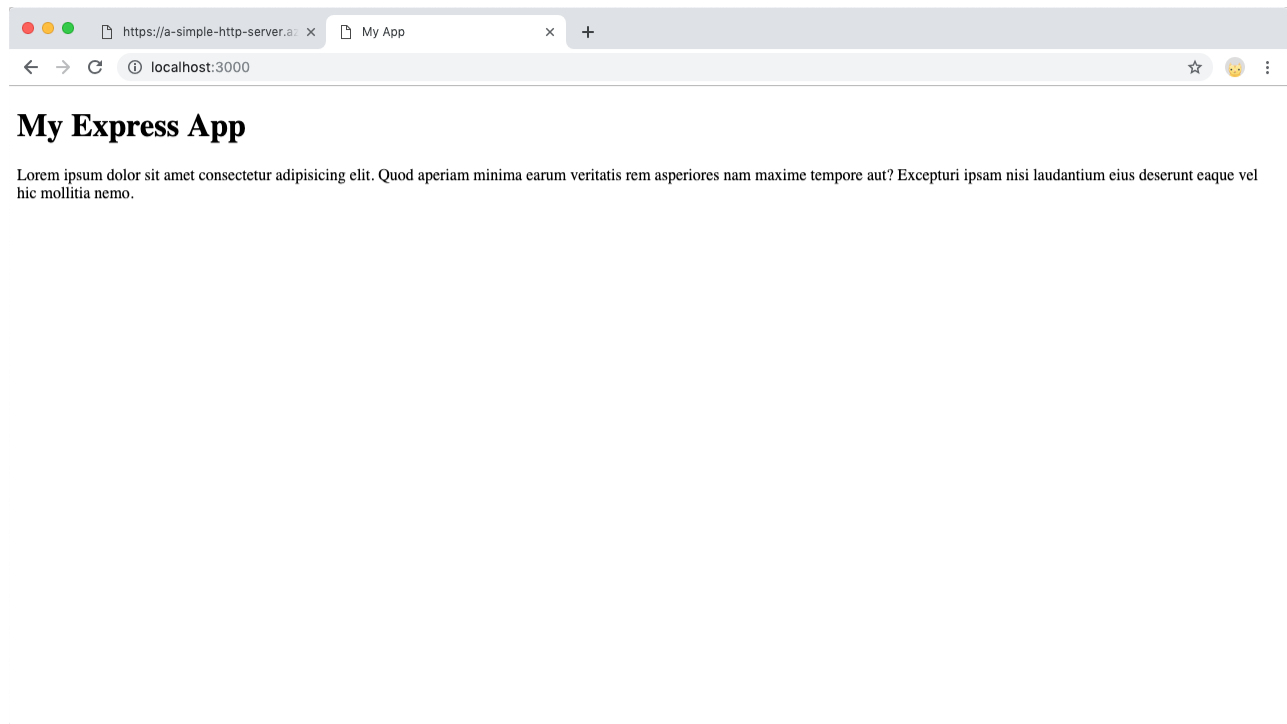
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>My App</title>
  </head>
  <body>
    <h1>My Express App</h1>
    <p>
      Lorem ipsum dolor sit amet consectetur adipisicing elit.
      Quod aperiam
      minima earum veritatis rem asperiores nam maxime tempore
      aut? Excepturi
      ipsam nisi laudantium eius deserunt eaque vel hic mollitia nemo.
```

```
</p>
</body>
</html>
```

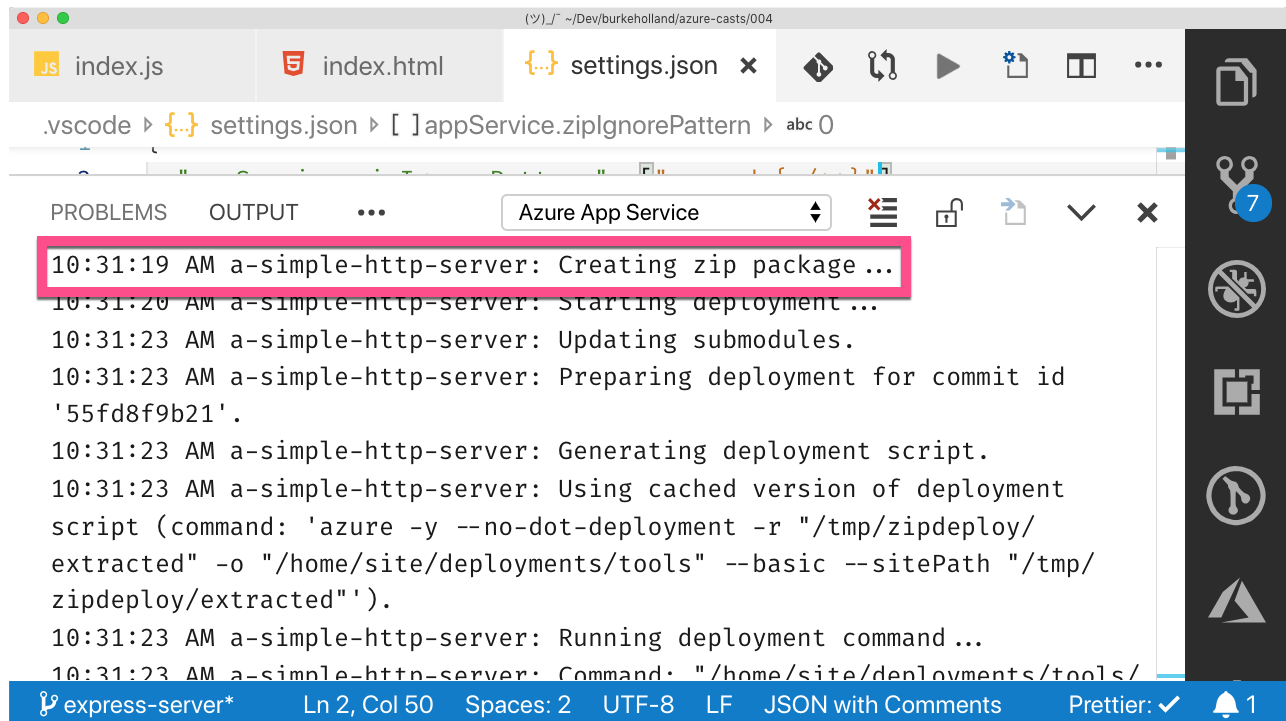
Now we're starting to get a lot of files in this project. Let's move the `index.js` and `index.html` files into an `src` folder.



Let's run it locally with `node src/index.js`. Lovely. Doesn't it feel good when things work the way you want? I love it.



Let's go ahead and deploy this. It tells us to check the output window for details, so let's do that. Open up the bottom panel in VS Code with `Cmd/Ctrl + J`. Select the "Output" tab and then select "App Service" from the dropdown. Here we can watch App Service chug along. You can see that it's creating a zip file here in the background and then uploading it.



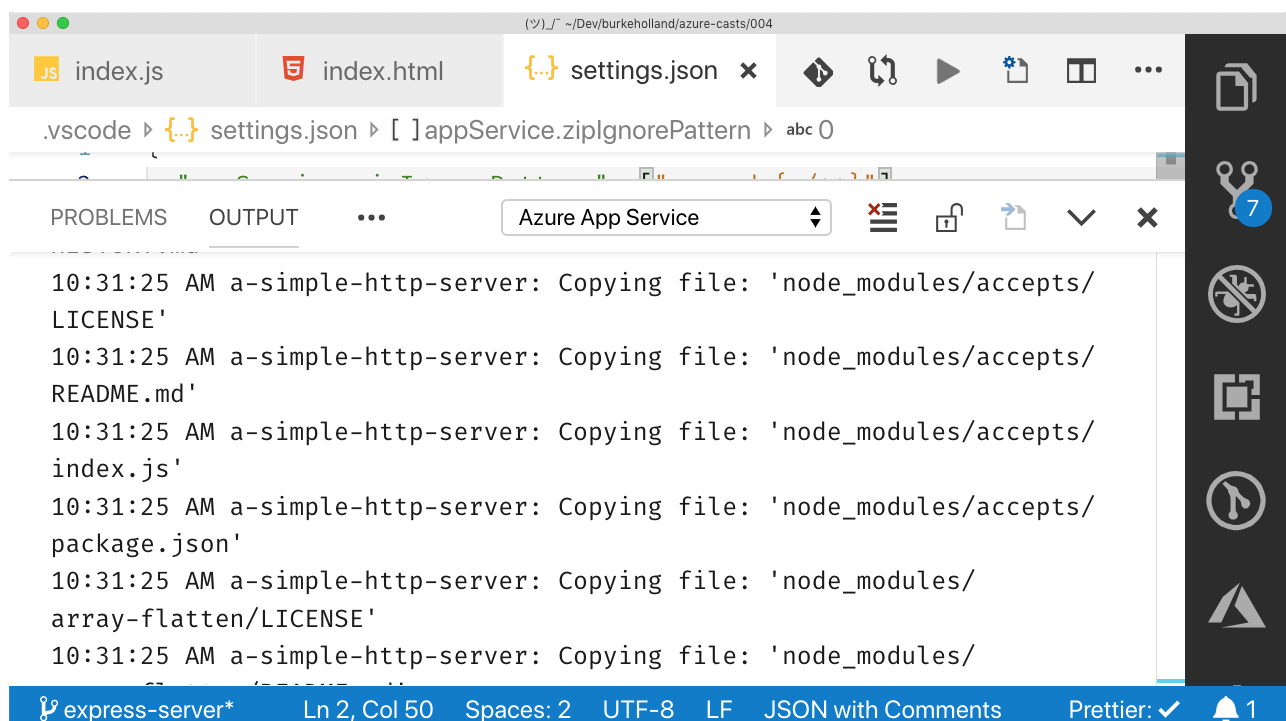
```
.vscode > settings.json > [ ] appService.zipIgnorePattern > abc 0

PROBLEMS OUTPUT ... Azure App Service

10:31:19 AM a-simple-http-server: Creating zip package ...
10:31:20 AM a-simple-http-server: Starting deployment ...
10:31:23 AM a-simple-http-server: Updating submodules.
10:31:23 AM a-simple-http-server: Preparing deployment for commit id
'55fd8f9b21'.
10:31:23 AM a-simple-http-server: Generating deployment script.
10:31:23 AM a-simple-http-server: Using cached version of deployment
script (command: 'azure -y --no-dot-deployment -r "/tmp/zipdeploy/
extracted" -o "/home/site/deployments/tools" --basic --sitePath "/tmp/
zipdeploy/extracted"').
10:31:23 AM a-simple-http-server: Running deployment command ...
10:31:23 AM a-simple-http-server: Command: "/home/site/deployments/tools/

express-server* Ln 2, Col 50 Spaces: 2 UTF-8 LF JSON with Comments Prettier: ✓ 1
```

Another thing that you can see here is that it's zipping up the entire node_modules folder and then uploading it.



```
.vscode > settings.json > [ ] appService.zipIgnorePattern > abc 0

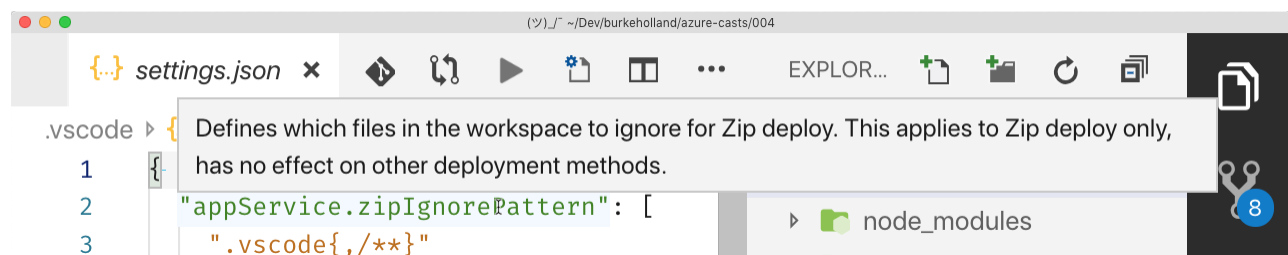
PROBLEMS OUTPUT ... Azure App Service

10:31:25 AM a-simple-http-server: Copying file: 'node_modules/accepts/
LICENSE'
10:31:25 AM a-simple-http-server: Copying file: 'node_modules/accepts/
README.md'
10:31:25 AM a-simple-http-server: Copying file: 'node_modules/accepts/
index.js'
10:31:25 AM a-simple-http-server: Copying file: 'node_modules/accepts/
package.json'
10:31:25 AM a-simple-http-server: Copying file: 'node_modules/
array-flatten/LICENSE'
10:31:25 AM a-simple-http-server: Copying file: 'node_modules/

express-server* Ln 2, Col 50 Spaces: 2 UTF-8 LF JSON with Comments Prettier: ✓ 1
```

Yikes. We don't want that. That folder will get enormous as our app gets bigger and Azure is going to run an `npm install` as part of this Oryx build command anyway. Uploading the `node_modules` folder is super inefficient. Let's instruct VS Code to please *not* do that.

Open the `settings.json` file in the `.vscode` folder. This setting here for `appService.zipIgnorePattern` looks promising. If we're not sure, we can just put our mouse over the settings and it will show us in a tooltip what this settings does.



That looks right. Let's add a pattern to ignore the `node_modules` folder. I don't know how to write ignore patterns, but I know how to copy this line here and change it to `"node_modules"`. I can copy/paste code with the best of them.

```
{
  "appService.zipIgnorePattern": [
    ".vscode{,/**}",
    "node_modules{,/**}"
  ],
  ...
}
```

If you open that `.deployment` file that was created by VS Code when we first uploaded this site, you can see a setting. This is the setting that tells AppService to do an `npm install` and pull in all of our dependencies.

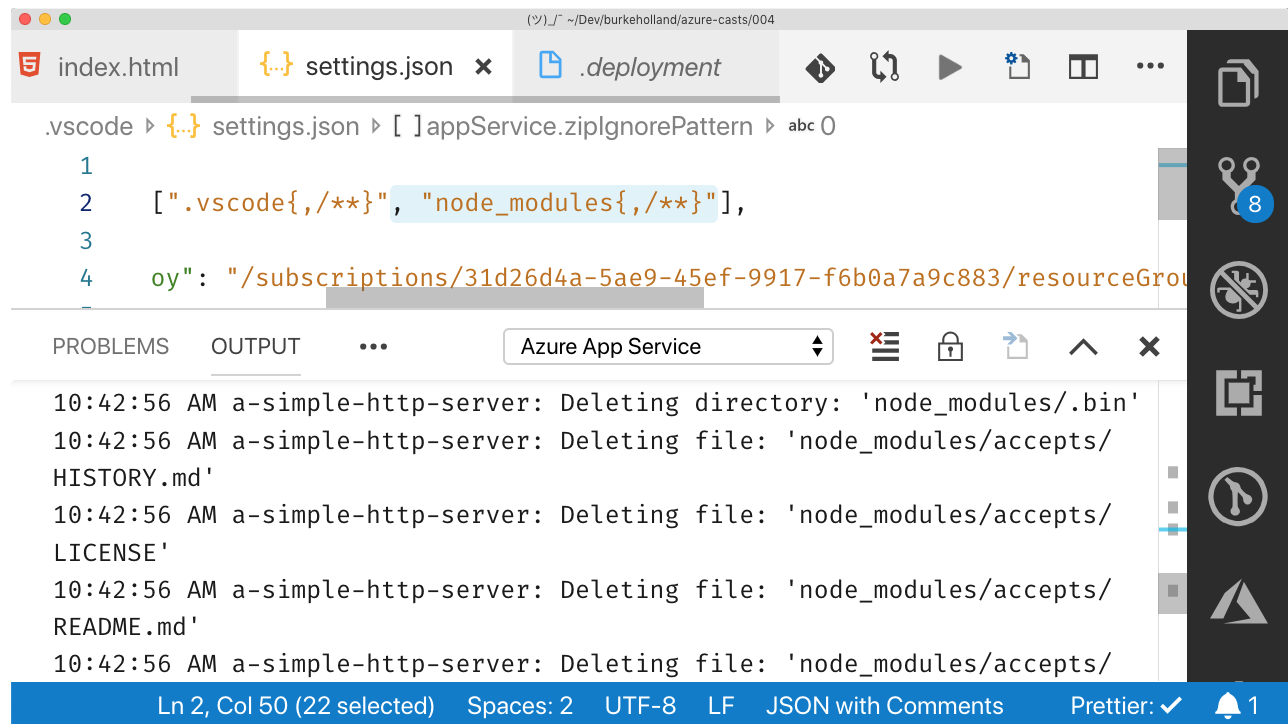
```
[config]
```



```
SCM_DO_BUILD_DURING_DEPLOYMENT=true
```

Now let's do our deployment again.

And let's open up the output panel and see what Azure is up to this time...



The screenshot shows the Visual Studio Code editor with the `.deployment` file open. The file content is as follows:

```
1  
2  [".vscode{,/**}", "node_modules{,/**}"],  
3  
4  oy": "/subscriptions/31d26d4a-5ae9-45ef-9917-f6b0a7a9c883/resourceGroup"
```

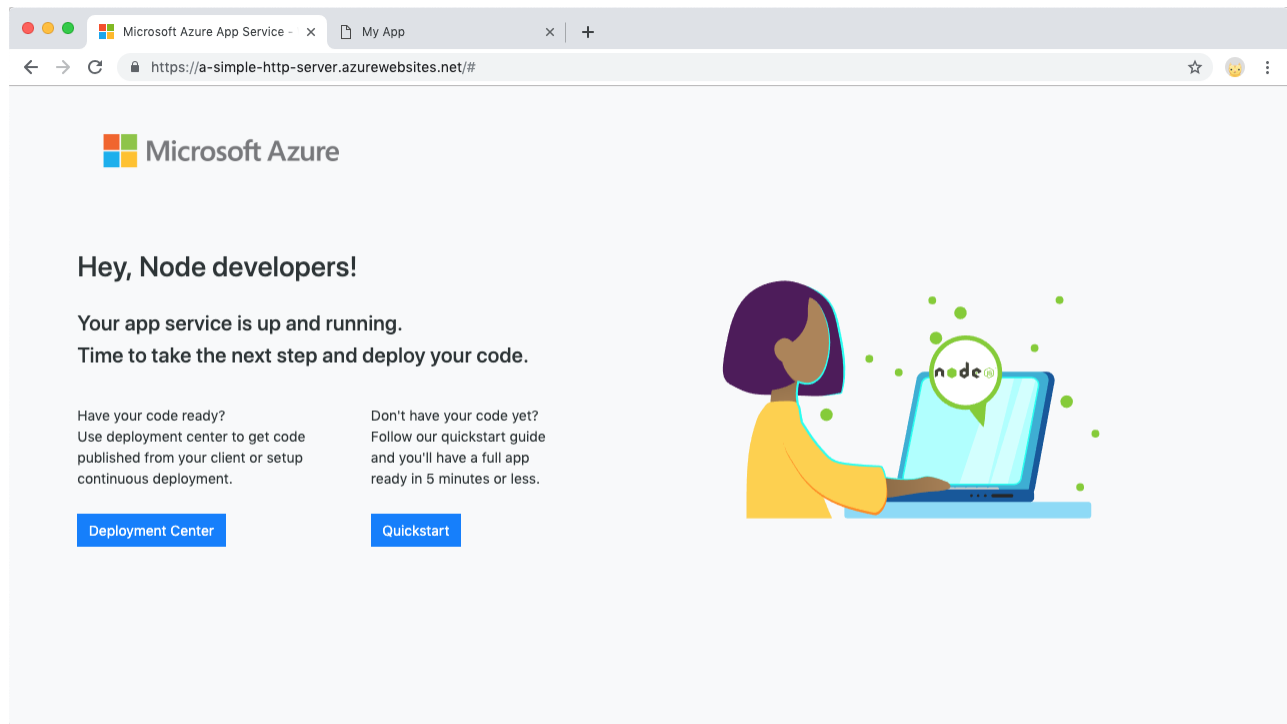
The Output panel at the bottom shows the following logs for the Azure App Service:

```
10:42:56 AM a-simple-http-server: Deleting directory: 'node_modules/.bin'  
10:42:56 AM a-simple-http-server: Deleting file: 'node_modules/accepts/  
HISTORY.md'  
10:42:56 AM a-simple-http-server: Deleting file: 'node_modules/accepts/  
LICENSE'  
10:42:56 AM a-simple-http-server: Deleting file: 'node_modules/accepts/  
README.md'  
10:42:56 AM a-simple-http-server: Deleting file: 'node_modules/accepts/
```

The status bar at the bottom indicates: Ln 2, Col 50 (22 selected) | Spaces: 2 | UTF-8 | LF | JSON with Comments | Prettier: ✓ | 1 notification.

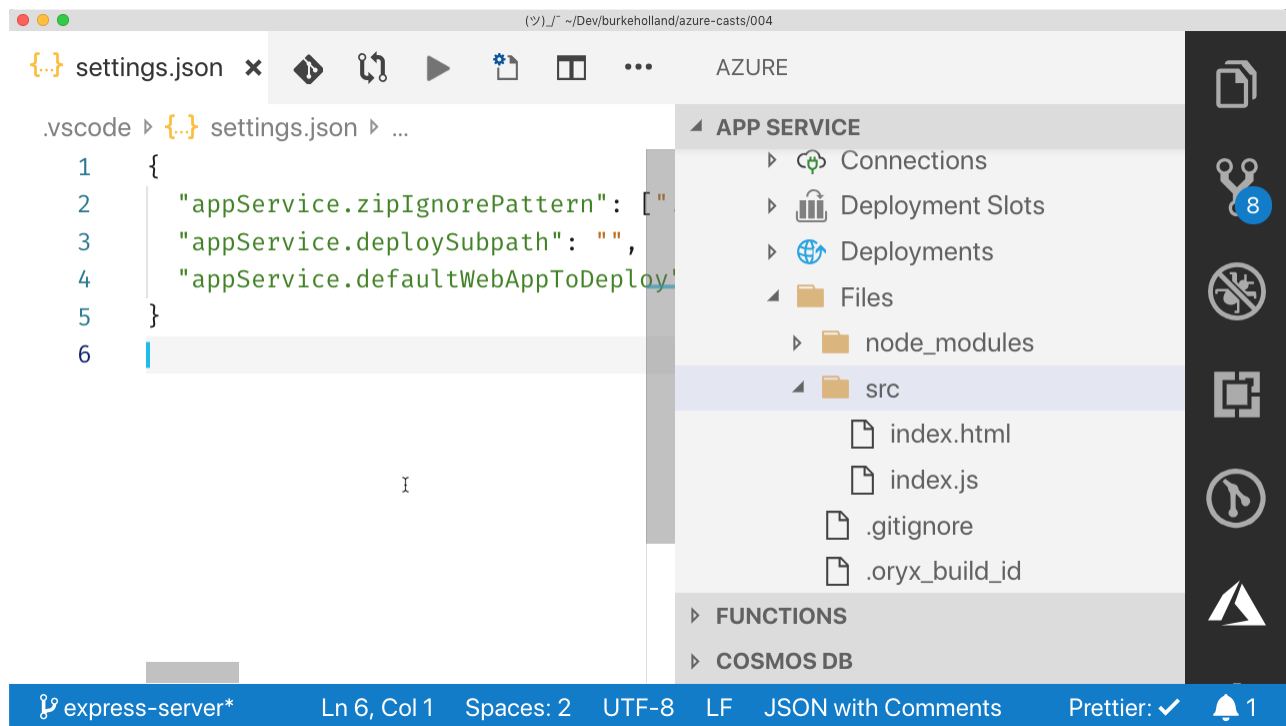
Ah - ok - so it's deleting everything and redeploying, and this time we don't get those logs about it copying over all of our node_modules because we didn't upload any. Instead it just does the npm install - you know - the way god intended.

And let's check it out on Azure to make sure it works...

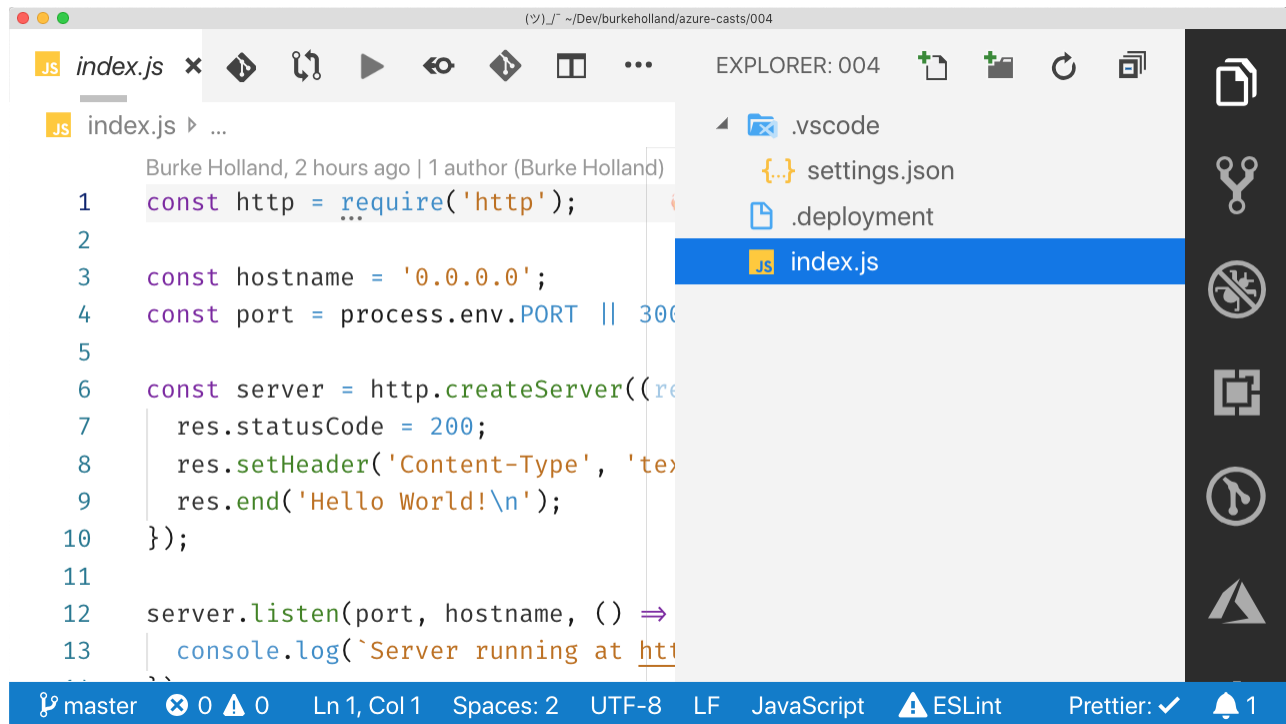


Wait. What? This is the default page you get BEFORE you deploy anything. What is going on? Did I screw up or did Azure screw up?

First, let's just make sure that our files are actually there. This page here makes it seem like they aren't, but we can know for sure by going to the AppService extension for VS Code and expanding the site and then going to files.



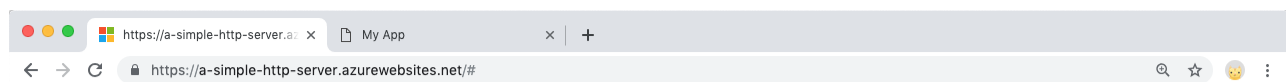
My files are definitely there. I just had this working. Let's roll back to where we started with the simple "index.js" file that was working and see what happens. I'm going to do that by just switching back to the master branch.



The screenshot shows the Visual Studio Code editor interface. The Explorer sidebar on the right lists files: .vscode, settings.json, .deployment, and index.js (which is selected). The main editor area displays the content of index.js, which is a simple HTTP server script. The status bar at the bottom indicates the current file is 'index.js', the encoding is 'UTF-8', and the language is 'JavaScript'. It also shows linting tools like ESLint and Prettier are active.

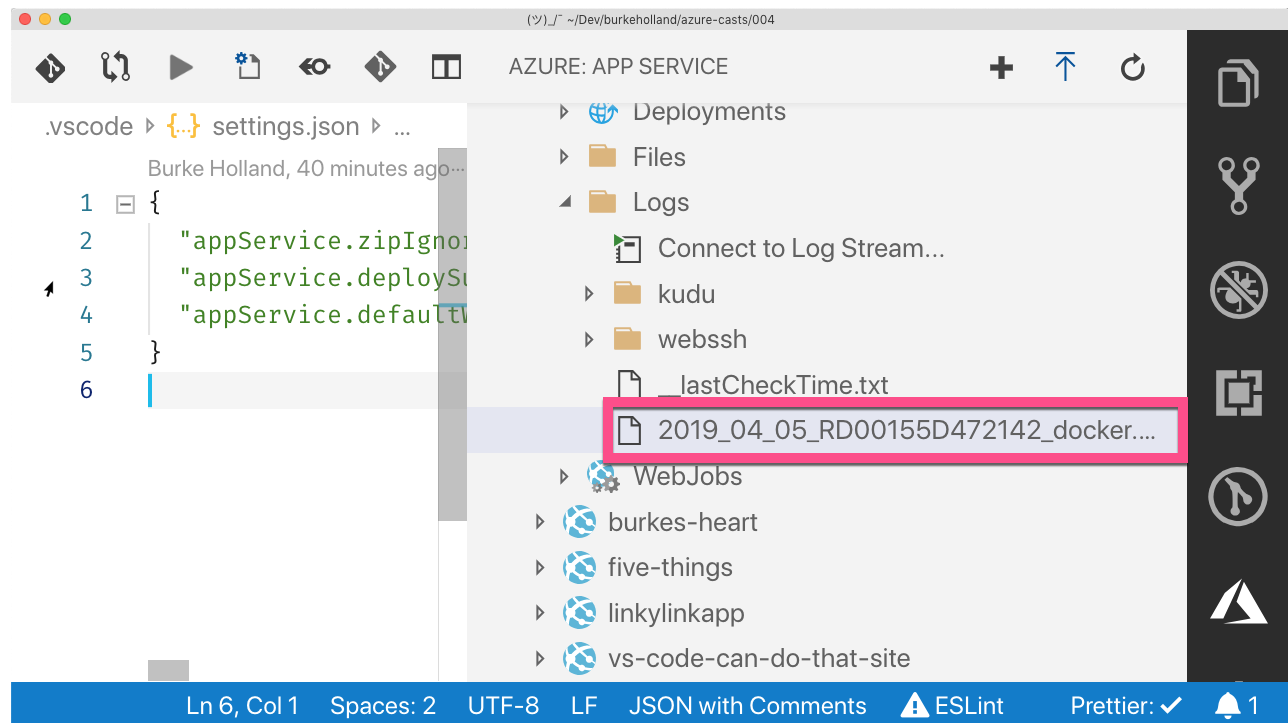
```
1 const http = require('http');
2
3 const hostname = '0.0.0.0';
4 const port = process.env.PORT || 3000;
5
6 const server = http.createServer((req, res) => {
7   res.statusCode = 200;
8   res.setHeader('Content-Type', 'text/plain');
9   res.end('Hello World!\n');
10 });
11
12 server.listen(port, hostname, () => {
13   console.log(`Server running at http://localhost:${port}`);
14 });
```

Let's deploy this again and see what we get - just to make sure we aren't going crazy here. This did work before, right?



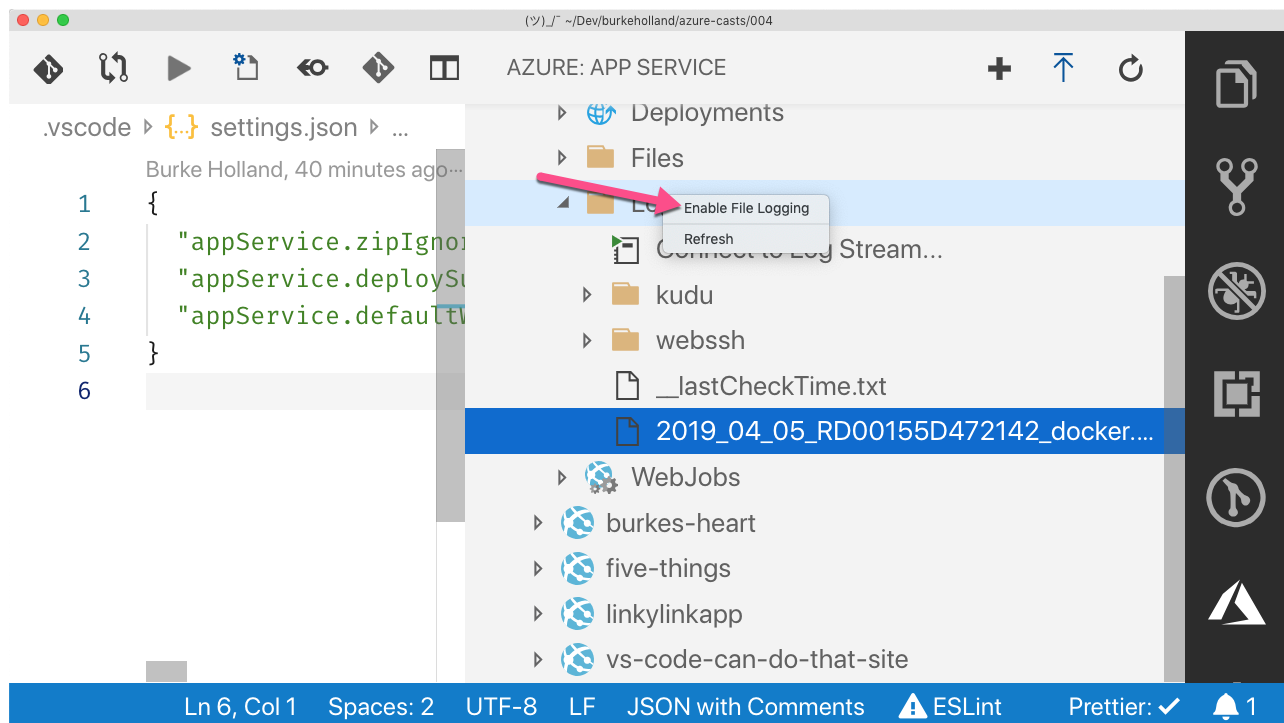
Hello World!

I'm officially confused. Let's go back to the Express project by switching branches and deploying. We know that this doesn't work, so let's go to the log files. We can get there from the AppService extension. We've got a few options in here. One of them is to connect to the Log Stream. This will stream the application logs right into VS Code. Another one is to just look at the log files themselves. There is only one right now with this date format in front of it and it says "Docker"

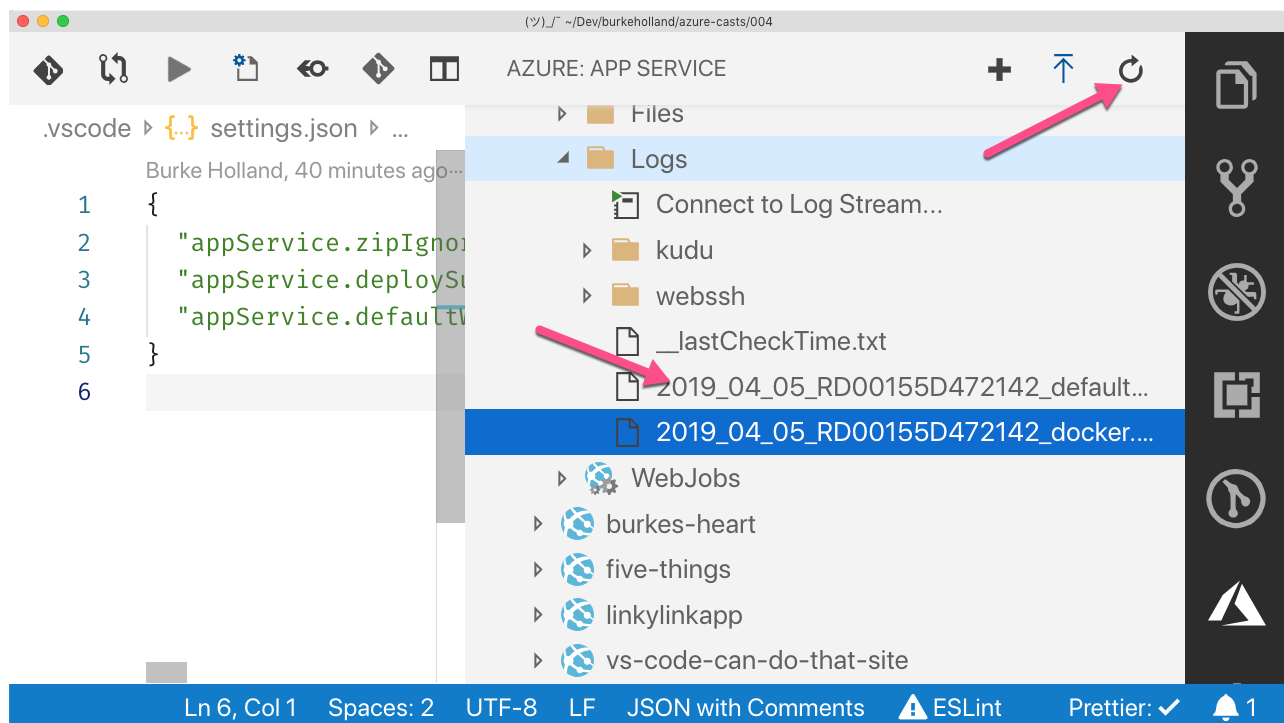


Why is there a "Docker" log? Behind the scenes AppService is running our app in a Docker Container. That's how AppService works - it's all based on containers. You can bring your own, or you can just deploy files, but they will end up in a container once they're in Azure.

We don't have any application logs yet, and that's because they aren't on by default. If we right-click the "Logs" folder, we can select "Enable File Logging" and that will restart the app and enable the file logs.




Now click the refresh icon in the extension taskbar, and we see a second file.



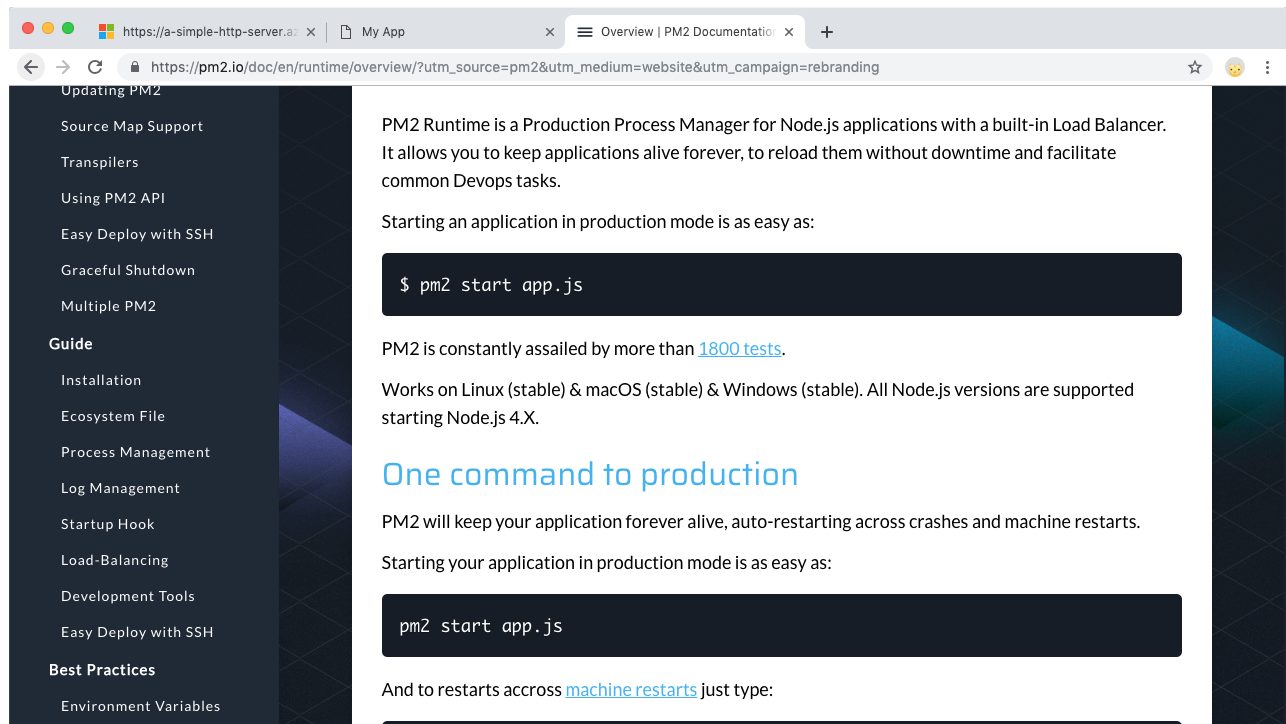
This file is the application log. Let's have a peek.

Alright. As we scroll down here we can see up at the top that Azure is reporting that it can't find a startup command or autodetected startup script. And that's it's running the static default site. So I guess that explains why we're seeing the static default site.



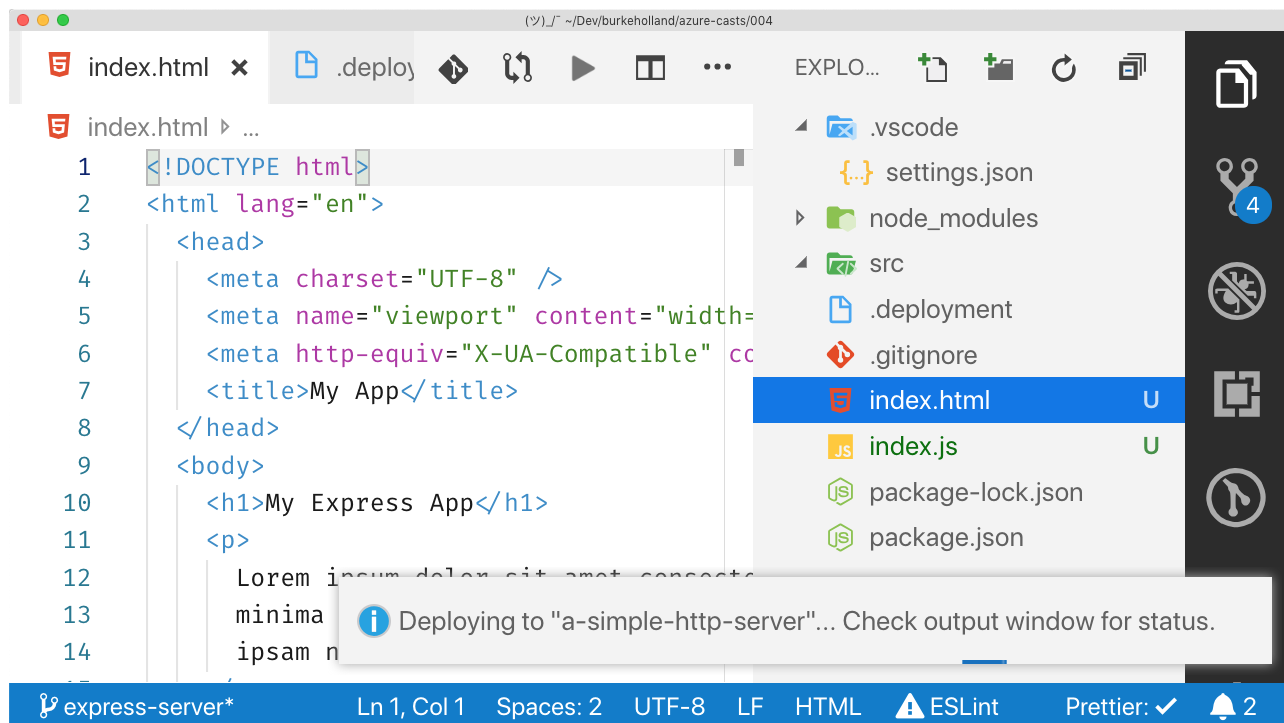
```
1 2019-04-05T16:32:26.0360054Z
2 2019-04-05T16:32:26.0387147Z
3 2019-04-05T16:32:26.0387215Z
4 2019-04-05T16:32:26.0387262Z
5 2019-04-05T16:32:26.0387307Z
6 2019-04-05T16:32:26.0387354Z
7 2019-04-05T16:32:26.0387398Z APP SERVICE ON LINUX
8 2019-04-05T16:32:26.0387441Z
9 2019-04-05T16:32:26.0387482Z Documentation: http://aka.ms/webapp-linux
10 2019-04-05T16:32:26.0387524Z NodeJS quickstart: https://aka.ms/node-qs
11 2019-04-05T16:32:26.0387566Z NodeJS Version : v10.14.1
12 2019-04-05T16:32:26.0387608Z
13 2019-04-05T16:32:26.4636905Z
14 2019-04-05T16:32:26.4856189Z No startup command or autodetected startup script found. Running default static site.
15 2019-04-05T16:32:26.4940427Z
16 2019-04-05T16:32:28.6514962Z
17 2019-04-05T16:32:28.6515314Z
18 2019-04-05T16:32:28.6515368Z
19 2019-04-05T16:32:28.6515406Z
20 2019-04-05T16:32:28.6515447Z
21 2019-04-05T16:32:28.6515490Z
22 2019-04-05T16:32:28.6515531Z
23 2019-04-05T16:32:28.6515575Z
24 2019-04-05T16:32:28.6515616Z
25 2019-04-05T16:32:28.6515654Z
26 2019-04-05T16:32:28.6515691Z
27 2019-04-05T16:32:28.6515731Z
28 2019-04-05T16:32:28.6515768Z
```

Below that we've got some ASCII art that says "PM2". Let's Google it. Below that it says it's running pm2 start to start the application. What the heck is PM2? Let's Google it.

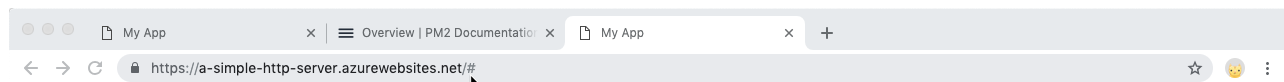


It's a Node process manager. It must be baked into our container behind the scenes.

OK, so Azure can't start our site. But it *could* start it when we just had an "index.js" file. So lets go back and move our index.js and public.html files back to the root and deploy.



And let's see if that works.



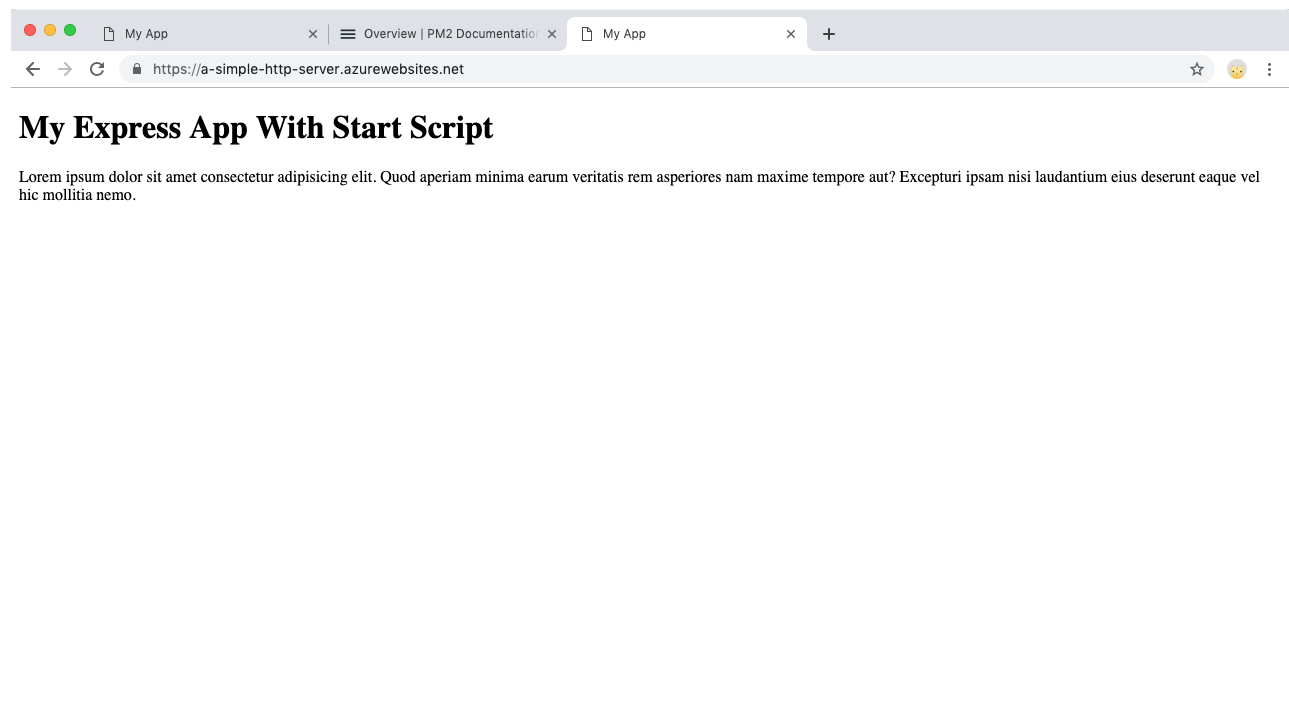
My Express App

Lorem ipsum dolor sit amet consectetur adipiscing elit. Quod aperiam minima earum veritatis rem asperiores nam maxime tempore aut? Excepturi ipsam nisi laudantium eius deserunt eaque vel hic mollitia nemo.

That works! Azure follows a convention here. If you don't tell it how to start your app, it tries to guess. It will look for "index.js", "server.js", "main.js", "app.js" or even "bin/www" which is the starting point for an Express application if you scaffold it with the Express CLI.

But we should really be telling Azure how to start this site. We can do that by adding a startup script to our `package.json` file. So let's put our files back in the "src" folder. Then we'll add a startup script. And change our index.html so we're sure our change is picked up.

OK. That works. Azure can now start our site no matter how we structure it or what we name our files.



Now it's important to note that using PM2 to start our app is kind of important here. Let me show you why.

When you are running a Node application, you run the risk of it crashing if you have an unhandled error. We can make this happen pretty easily. Let's add another route here that tries to read a file that doesn't exist into a stream.

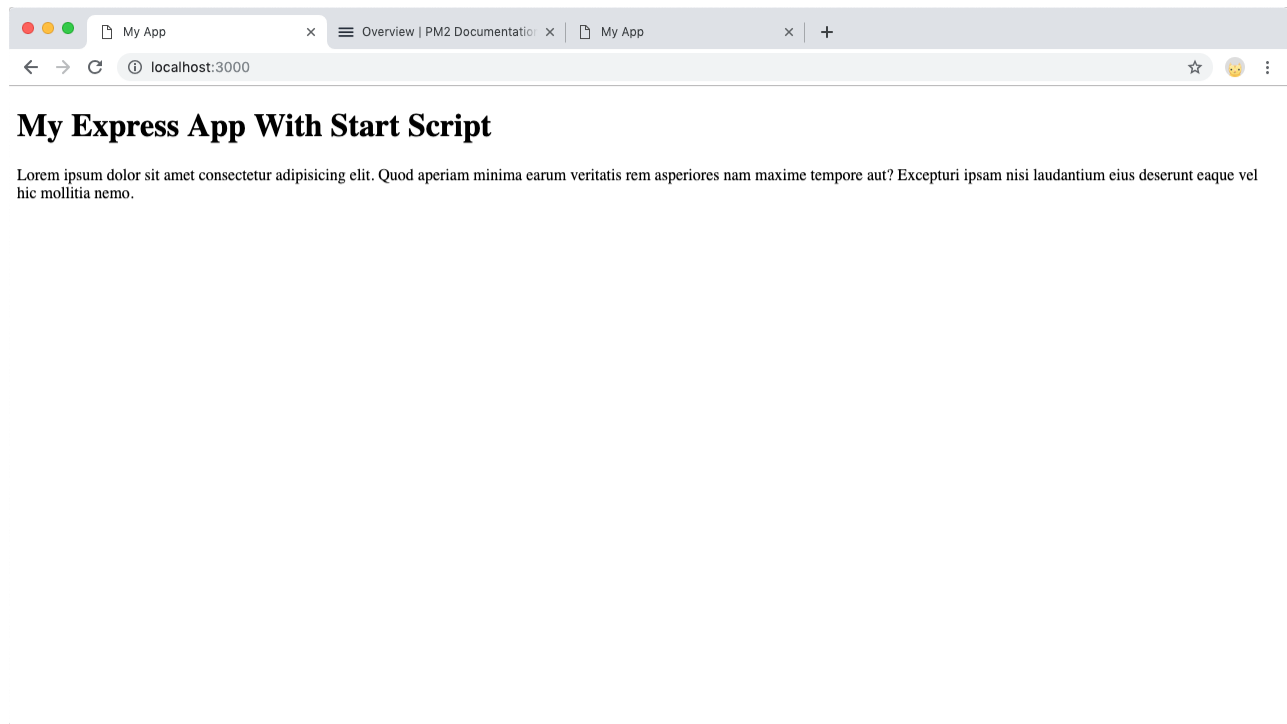
```
const express = require("express");
const app = express();
const path = require("path");
const fs = require("fs");
const port = process.env.PORT || 3000;

// viewed at http://localhost:3000
app.get("/", function(req, res) {
  res.sendFile(path.join(__dirname + "/index.html"));
});

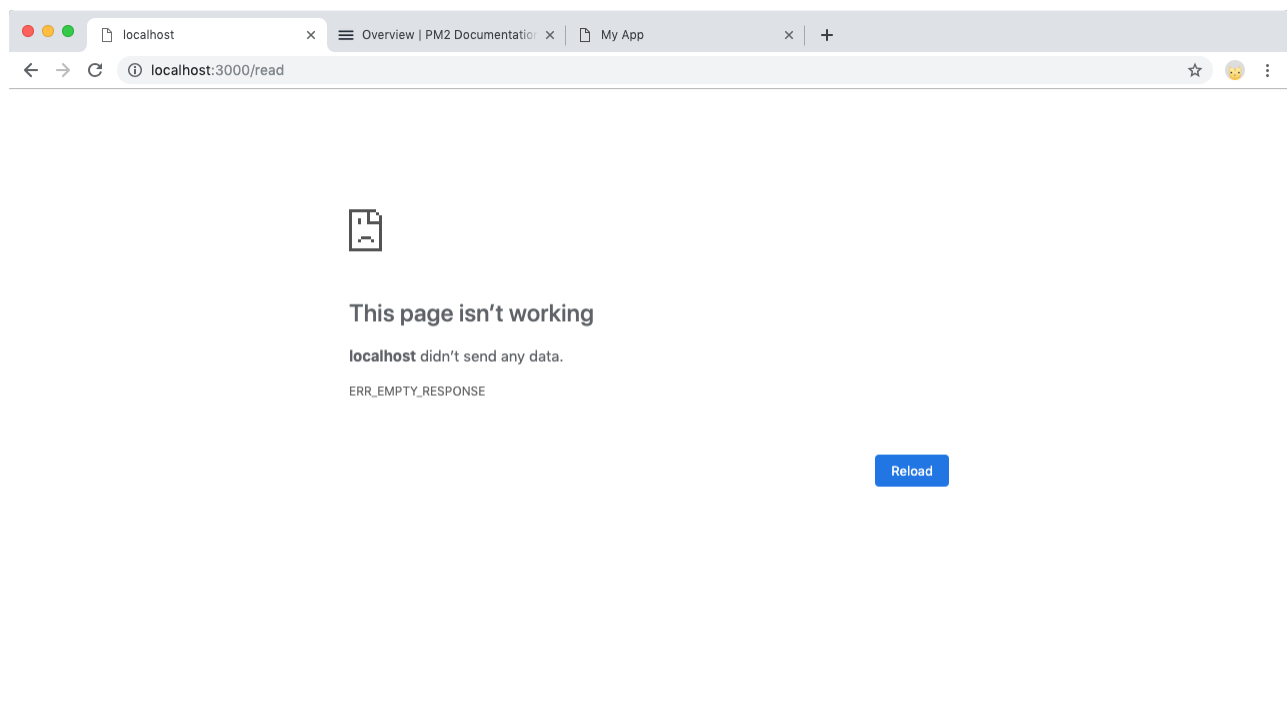
app.get("/read", function(req, res) {
  const stream = fs.createReadStream("does-not-exist.txt");
});

app.listen(port, () => console.log(`Example app listening on port ${port}!`));
```

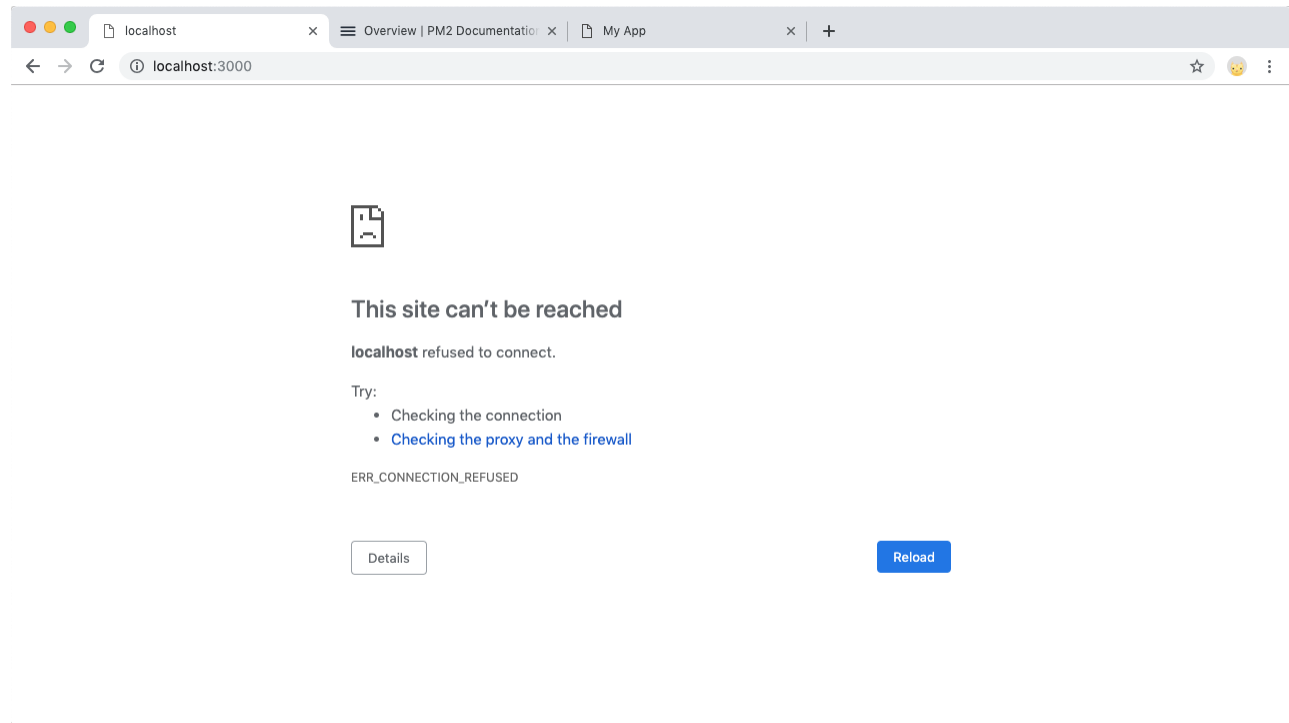
If you visit this site in the browser. Everything looks all hunky dory.



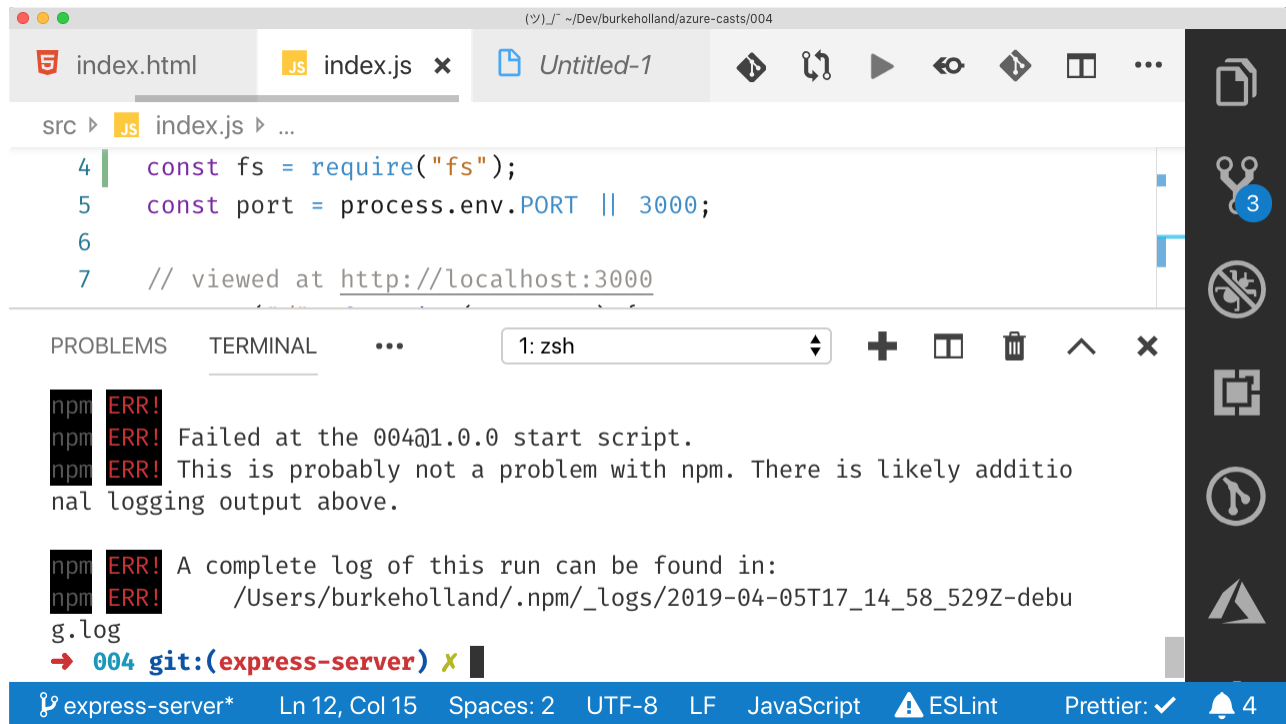
But if you go to /read and execute that bad line of code....



Uh oh. That didn't work. And here's the really bad part. If you go back to the root URL. That one isn't working anymore either.



This whole app is dead in the water. Have a look in the VS Code terminal and you'll see that Node has just crashed. This app is down for everyone. Not. Good.



The screenshot shows a VS Code editor window with the file `index.js` open. The code in the editor is:

```
4 const fs = require("fs");
5 const port = process.env.PORT || 3000;
6
7 // viewed at http://localhost:3000
```

The terminal at the bottom shows an npm error:

```
npm ERR! Failed at the 004@1.0.0 start script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.

npm ERR! A complete log of this run can be found in:
npm ERR!     /Users/burkeholland/.npm/_logs/2019-04-05T17_14_58_529Z-debug.log
→ 004 git:(express-server) x
```

The status bar at the bottom indicates the file is `express-server*`, line 12, column 15, with 2 spaces, UTF-8 encoding, LF line endings, JavaScript language, ESLint and Prettier extensions, and 4 notifications.

PM2 (and other Node process managers) will watch your Node process, and if it goes down, they restart it. Automatically.

PM2 is just installed from `npm` like any other Node package.

```
npm i -g pm2
```

PM2 works off the concept of configuration files. We could create one of those by running `pm2 init`. Then we could specify our startup point there. But we already have a startup point in our `package.json`. So instead, we can just tell PM2 to read our start script by running...

```
pm2 start npm -- start
```

Notice the space between the dashes here - that's important.

The screenshot shows a VS Code editor window with the following components:

- Editor Tabs:** index.html, index.js, and Untitled-1.
- Code Editor:** Displays the content of index.js:

```
src > JS index.js > ...  
You, 10 minutes ago | 2 authors (Burke Holland and others)  
1  const express = require("express");  
2  const app = express();  
3  const path = require("path");
```
- Terminal:** Shows PM2 logs and a table of running processes.

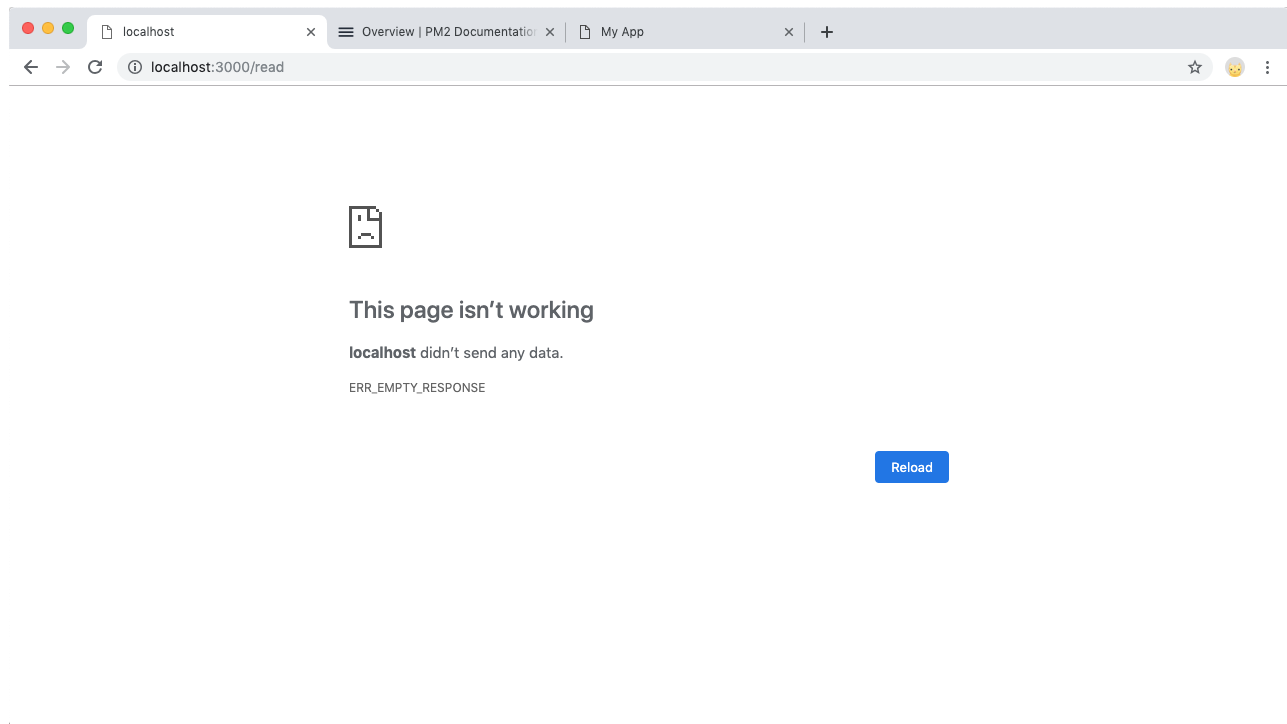
```
[PM2] Starting /usr/local/bin/npm in fork_mode (1 instance)  
[PM2] Done.
```

Name	id	mode	status	cpu	memory
npm	0	fork	online	0	0%

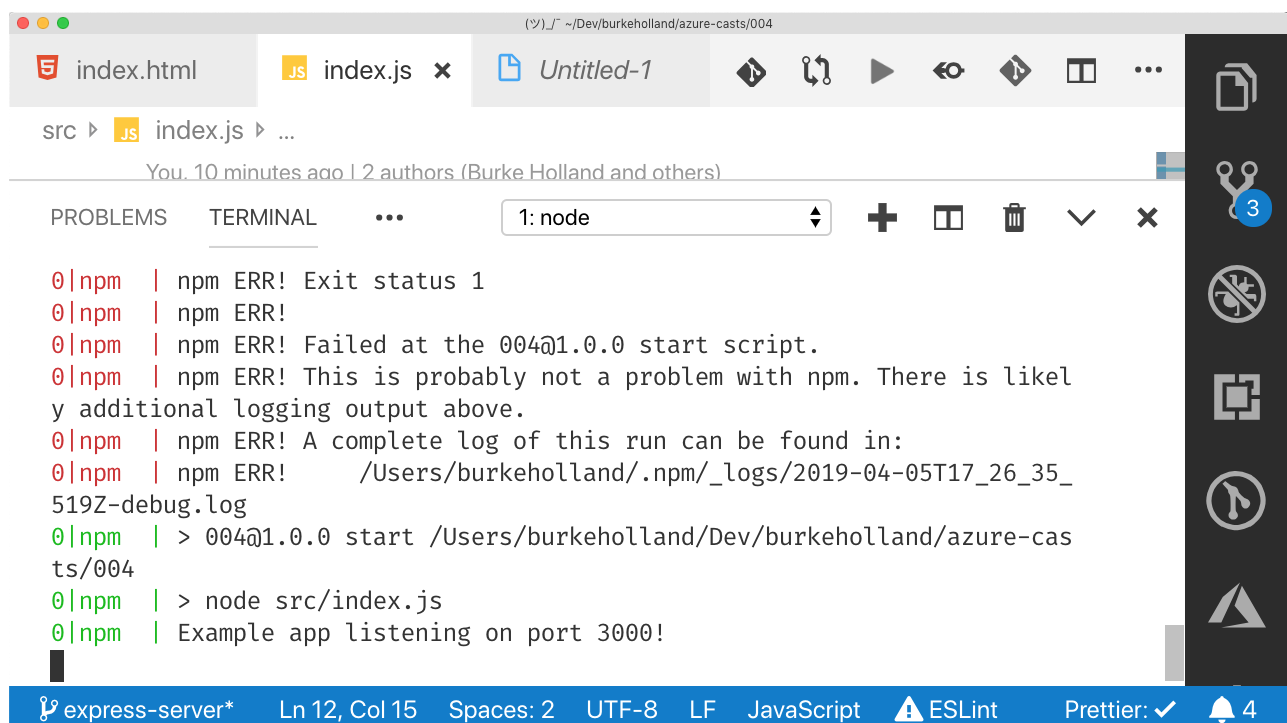
Use ``pm2 show <id/name>`` to get more details about an app
→ 004 git:(express-server) x
- Status Bar:** Shows the current file is express-server*, at line 12, column 15, with settings for Spaces: 2, UTF-8, LF, JavaScript, ESLint, and Prettier.

Now our app is running. It's name is "npm" because that's the process that PM2 executed. If we want to see the logs from our app, we can run `pm2 logs` and then pass in 0 - the ID of the process or "npm", the name of the process.

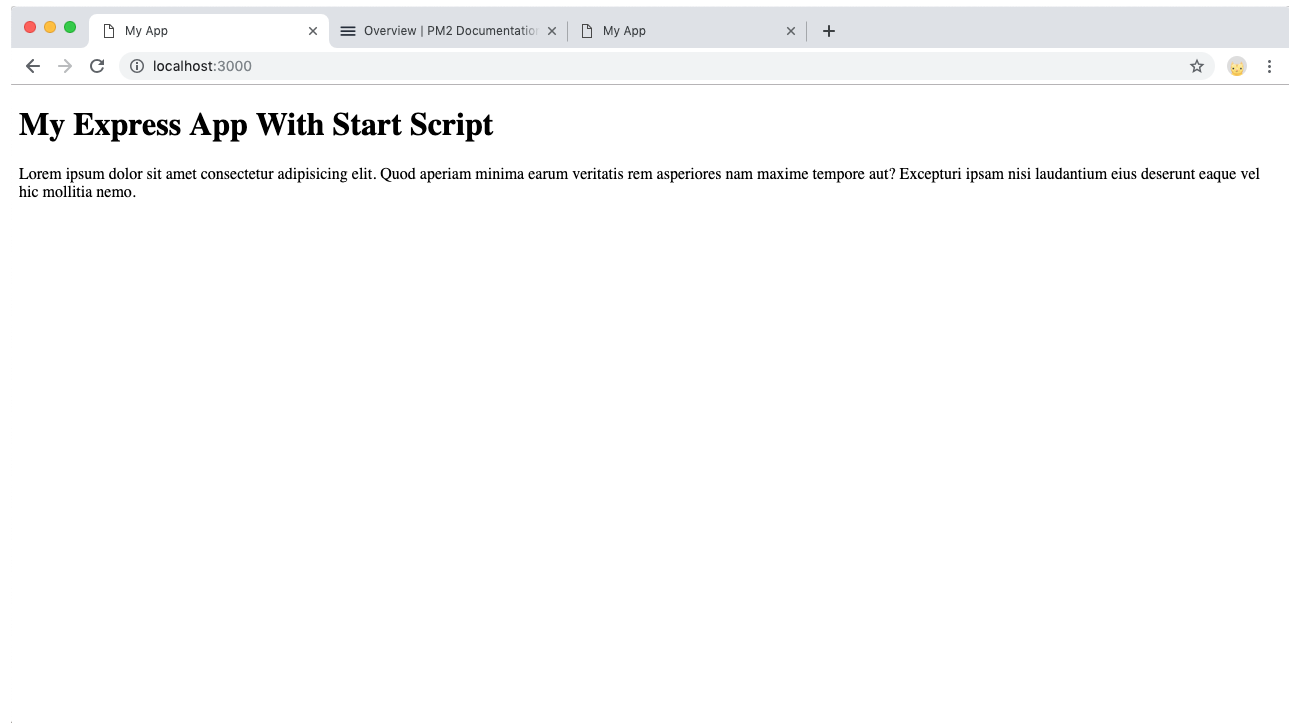
Our app is still running on port 3000. If we go to the "read" endpoint to crash it, it will crash....



BUT! But but but. This time, go back to VS Code and you can see that the app did crash and PM2 just restarted it for us.



So if we go back to the root of our app - it's still up!



Azure is using PM2 as well, so if we deploy this site to Azure - broken as it is, our app won't crash and stay down. Azure will use PM2 to make sure that it recovers. That's a very nice feature.

I hope you enjoyed this video. Please check out the other Azure Casts, and I'll see you again soon.