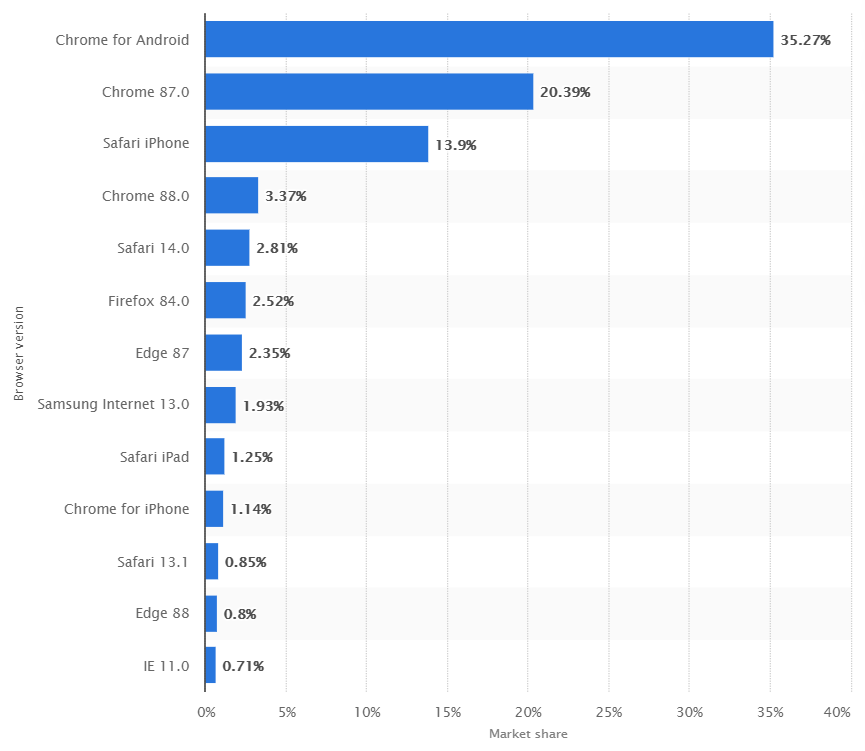
# Intro to devtools

There are many, many tools in the JS ecosystem for various kinds of needs. There are a few reasons why there are so many tools:

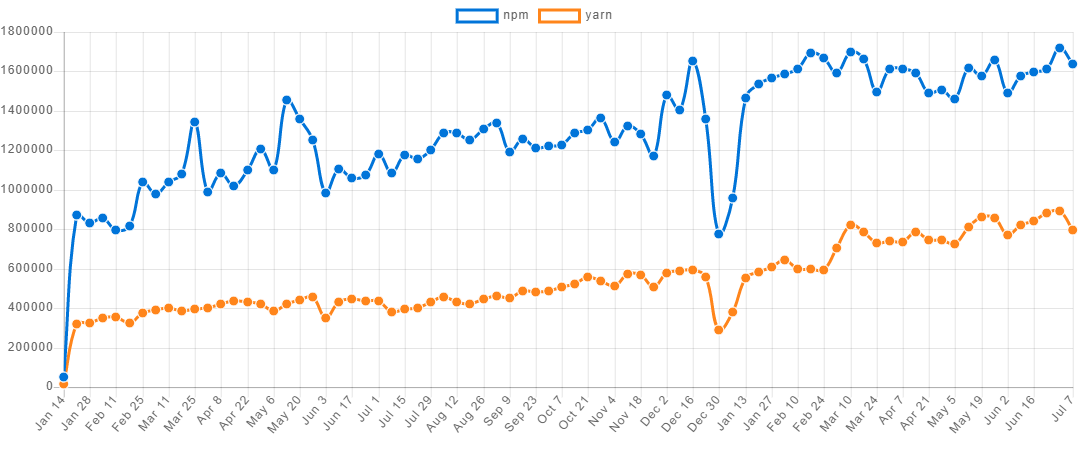
* Tools that are not supported by the browser - SASS, needs to be converted to CSS, Typescript need to be converted to supported JS version, JS features sometimes need to be translated into mechanisms that the browser supports.
* Browser needs to run code that is different than the code we actually wrote - CSS, JS need to be compressed, minified (multiple files to single file and also replacement of long variable names), obfuscated, etc.
* Javascript has a very big community and a lot of different usecases. Some people like Angular, some people like React.
* Javascript can be used in the backend and in the frontend making the ecosystem even bigger.

Some of them are:

* *webpack* - an open-source JavaScript module bundler. It is made primarily for JavaScript, but it can transform front-end assets such as HTML, CSS, and images if the corresponding loaders are included.
* *grunt* - a tool used to automatically perform frequent tasks such as minification, compilation, unit testing, and linting.
* *eslint* - static code analysis tool for identifying problematic patterns found in JavaScript code
* *babel* - free and open-source JavaScript transcompiler that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript. The need for this tool is mainly needed because browsers can not start supporting all the feature are are coming into JS right away - there is a lag period even for new browsers (you can see that in <https://caniuse.com/> ), let alone there is even bigger lag period for new browser version adoption sometimes ( you can see that: <https://gs.statcounter.com/browser-version-market-share> and [https://www.statista.com/sta...](https://www.statista.com/statistics/268299/most-popular-internet-browsers/) ). Luckily people are not using IE much, however in certain industries / working with certain clients you might need to support IE browsers or old versions of other browsers , unlikely, but possible. This is primarily true in government or restricted corporate environemnts where users are not allowed to install anything and the management of software if highly controlled.)



* *npm* - a package manager that let’s you install libraries and frameworks into your dev environment and start developing complex projects. Npm can also run tasks, like grunt. Main rival yarn:

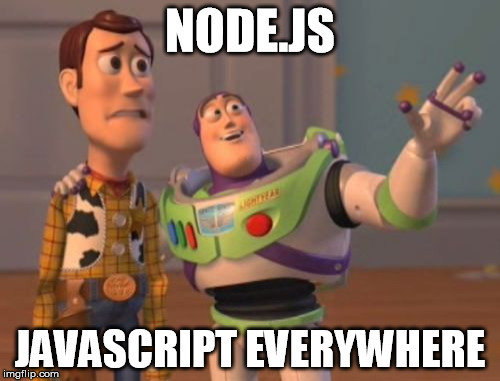


It’s not really imaginable that you will work in a JS project without using some of these tools. You will see that we will use them in this course when we start using Javascript Frameworks.

# Node

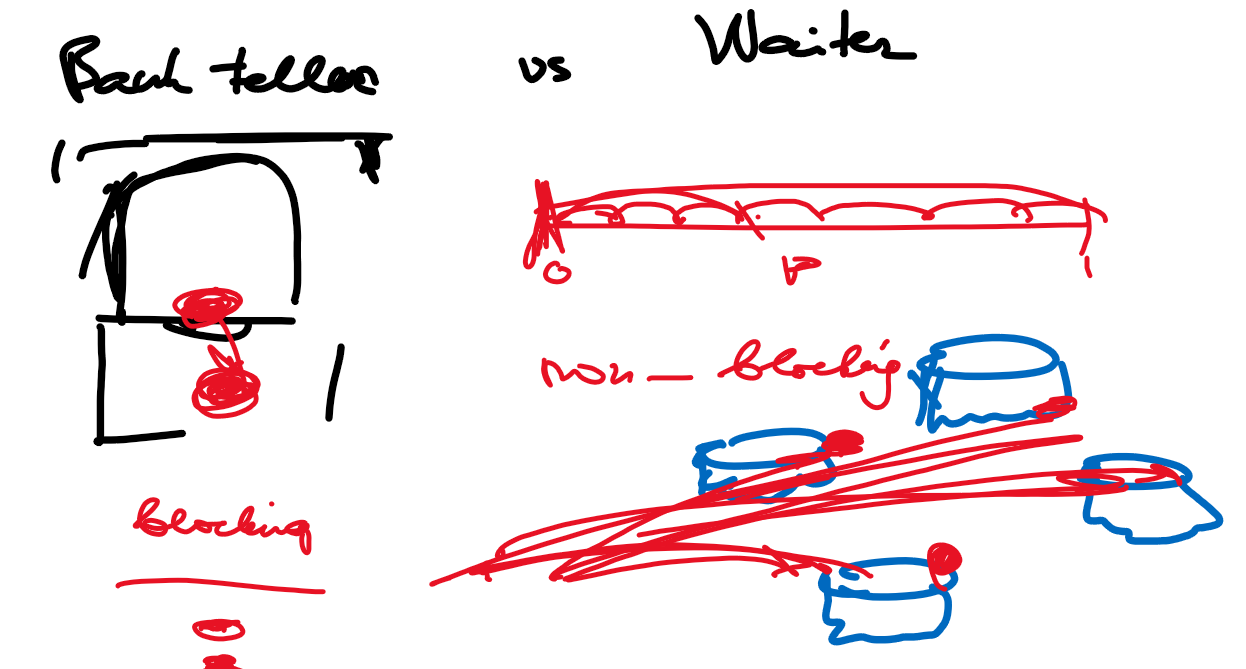
## Intro to node

Allows JavaScript to be used at server side. Designed to build scalable backend applications. It’s a runtime or an interpreter for JS. It allowed JS to be used as a backend language while before it was used as a frontend language only. This brought it into competition with PHP, Java, C# and so on. With it’s creation in 2009 it allowed JS to be run ont only the the browsers but on the server as well. Where is the revolution? Now a FS developer could write the full application - both backend and frontend - in Javascript.

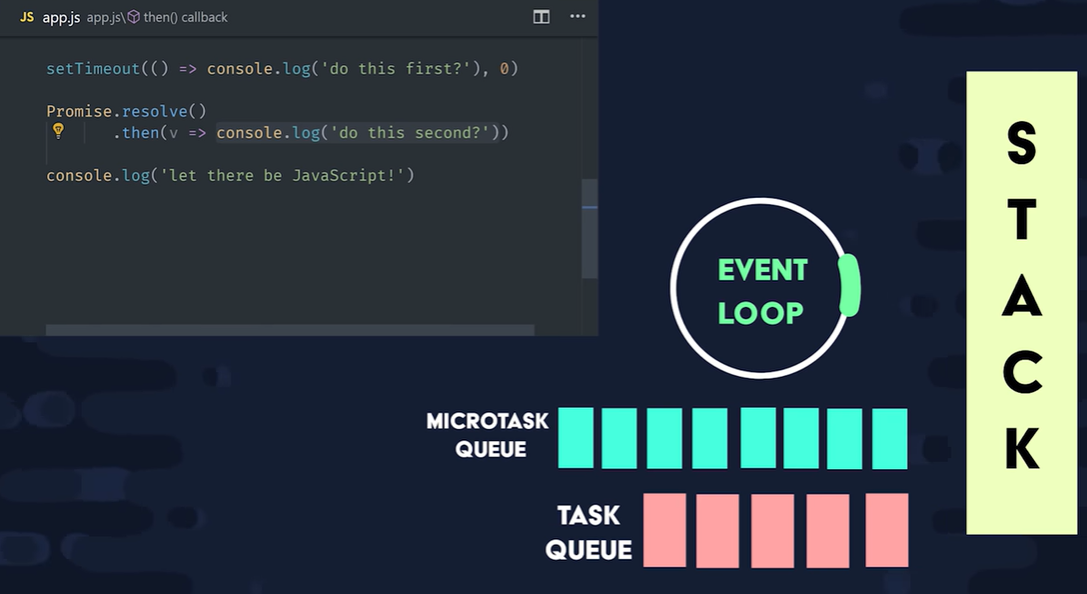


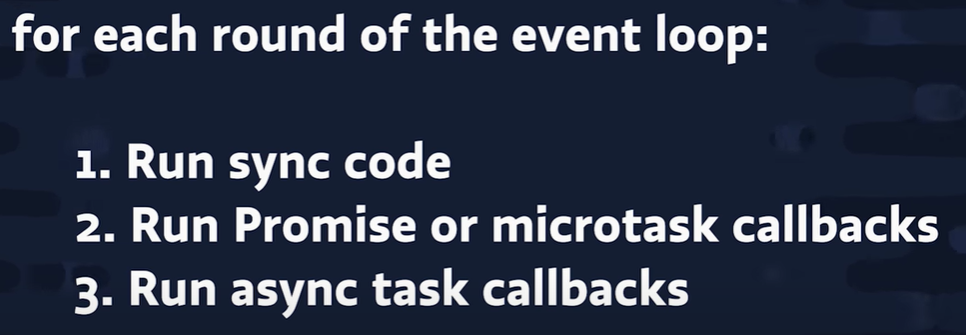
## Execution model

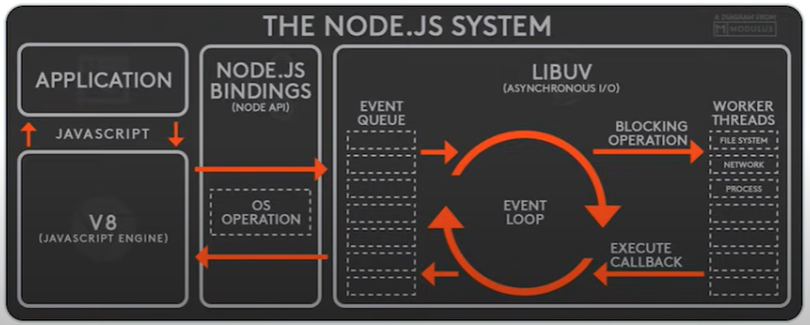
Single-threaded, non-blocking w/ asynchronous architecture that uses multiple queues - common example to explain the difference between blocking and synchronous architectures vs. non-blocking asynchronous architectures is how a bank teller vs. a waiter handles new clients as they are coming in - a teller blocks and if you want to scale you need parallel tellers. A waiter can service multiple clients all at once. This mechanism allows nodejs to be pretty performant / fast.



Works in a server-client paradigm, the same way as PHP w/ Laravel, Java w/ Spring MVC, Ruby on Rails work - the client makes a request → the sever responds with some content → the browser renders the response as a webpage.

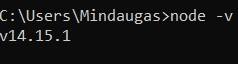






## Installation

Simple, go to <https://nodejs.org/en/> and choose the LTS version. You can check the version after installation:



## Executing simple examples

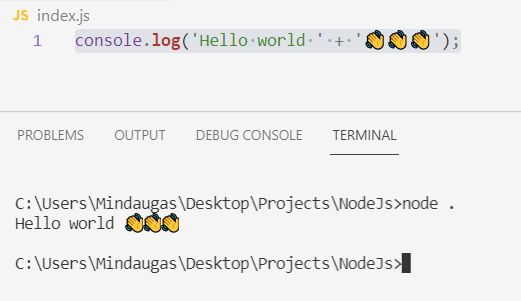
### REPL

Read-eval-print-loop is way to interact with node js from the console:

* *console.log(‘hello world’)*
* It has autocomplete
* We can define variables, perform arithmetic. *.*
* *Ctrl + C / exit.*

### Running JS files

The default start files is index.js.



Node.js treats each file as a separate module. Modules are cached after first time they are loaded. Folders can be used as modules as well as files.

*// circle.js … is left empty*

*// square.js*

const circle = **require**("./circle.js");

const { PI } = Math;

*// Export - default value*

**module**.**exports** = class **Square** {

constructor(width) {

this.width = width;

}

**area**() {

return this.width \*\* 2;

}

};

*// index.js*

const **Square** = **require**("./square.js");

let square = new **Square**(5);

console.**log**(square.**area**());

We can export functions and variables

*// helpers.js*

const **getName** = () => {

return "Jim";

};

let age = 55;

exports.**getName** = **getName**;

exports.age = age;

*// index.js*

const **Square** = **require**("./square.js");

const { **getName** , age} = **require**("./helpers.js");

let sq1 = new **Square**(5);

console.**log**("Hello world" + ' 👋👋👋 , the area is: ' + sq1.**area**() + ' name: ' + **getName**() + ' age: ' + age);

Exporting using Ecmascript modules you will need to tell than to Nodejs in the package.json file:



*// square.js*

*// const circle = require("./circle.js");*

const { PI } = Math;

*// Export - default value*

export default class **Square** {

constructor(width) {

this.width = width;

}

**area**() {

return this.width \*\* 2;

}

};

*// index.js*

*// const Square = require("./square.js");*

import Square from './square.js'

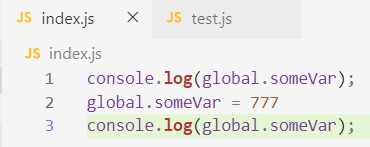
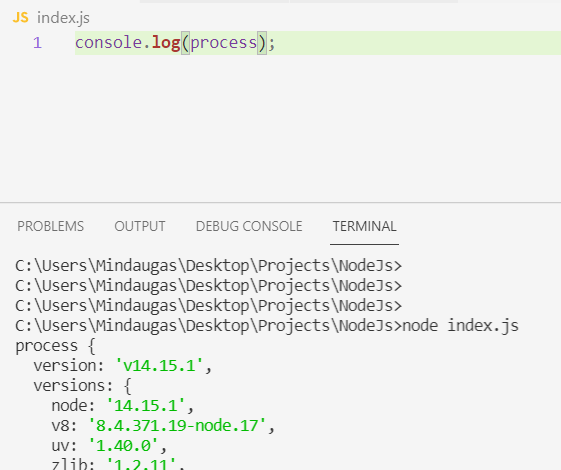
let square = new **Square**(5);

console.**log**(square.**area**());

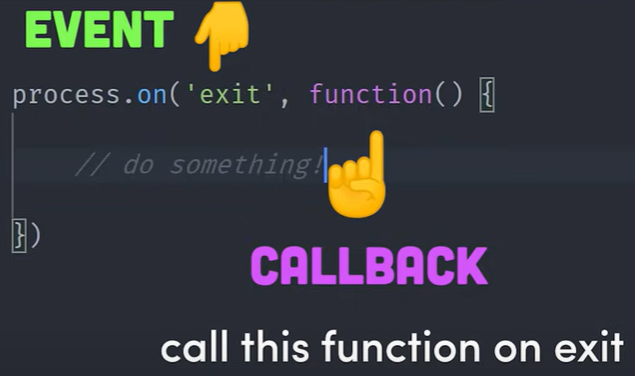
## Builtin identifiers

Globals are objects, which are available in all modules. Also, node.js provides built-in objects that are part of the JavaScript language itself, which are also globally accessible. Let’s enumerate some of them:

* \_\_dirname, \_\_filename → both not available when ecmascript modules are used: <https://nodejs.org/api/esm.html#esm_no_filename_or_dirname> and not available form the REPL.
* exports
* module
* require()
* console → printing, simple debugging.
* global → accessible through the entire runtime, can have variables attached
* process → can get the current platform name or environment variables.

## Callback and events



## File system

We can work in synchronous way, async way or async with promises way. The last one is recommended.

*// index.js*

const { **readFile**, **readFileSync**} = **require**("fs");

const txt = **readFileSync**('./hi.txt', 'utf-8');

console.**log**(txt);

*// index.js*

const { **readFile**, **readFileSync**} = **require**("fs");

**readFile**('./hi.txt', 'utf-8', (e, t) => {

console.**log**(t);

});

console.**log**('Some text');

*// index.js*

const { **readFile** } = **require**('fs').**promises**;

const txtPromise = **readFile**('./hi.txt', 'utf-8');

txtPromise.**then**((txt) => {

console.**log**(txt);

});

console.**log**('Some text');

## Creating simple backend

We know that when we are learning front-end we sometimes need a simple backend. We can create it with wordpres, json-server or … we can do it nodejs. But first we use this opportunity to talk about NPM because we use express.js on node.js. See this for more in-depth explanation: <https://zellwk.com/blog/crud-express-mongodb/>

Code:

const **express** = **require**('express');

const app = **express**();

app.**listen**(3000, () => {

console.**log**('listening on 3000')

})

app.**get**('/', (req, res) => {

console.**log**(req.headers['user-agent']);

res.**send**('<h1>Hello World</h1>');

console.**log**(res.**getHeaders**());

});

A bit more complex way to do it:

const **express** = **require**('express');

const { **readFile** } = **require**('fs');

const app = **express**();

app.**listen**(3000, () => {

console.**log**('listening on 3000')

})

*// app.get('/', (req, res) => {*

*// console.log(req.headers['user-agent']);*

*// res.send('<h1>Hello World</h1>');*

*// console.log(res.getHeaders());*

*// });*

app.**get**('/', (req, res) => {

**readFile**('views/index.html', 'utf-8', (e, t) => {

let errorString = '<h1 style="color: red">Something bad happened!</h1>';

e ? res.**status**(500).**send**(errorString) : res.**send**(t);

})

});

*// TODO :: we can actually even write data*

*// ... to the database via HTTP POST*

app.**post**('')

Html code:

<!DOCTYPE *html*>

<html *lang*="en">

<head>

<meta *charset*="UTF-8" />

<meta *http-equiv*="X-UA-Compatible" *content*="IE=edge" />

<meta *name*="viewport" *content*="width=device-width, initial-scale=1.0" />

<title>Document</title>

</head>

<body>

<h1>Hi 👋 👋 👋</h1>

<form>

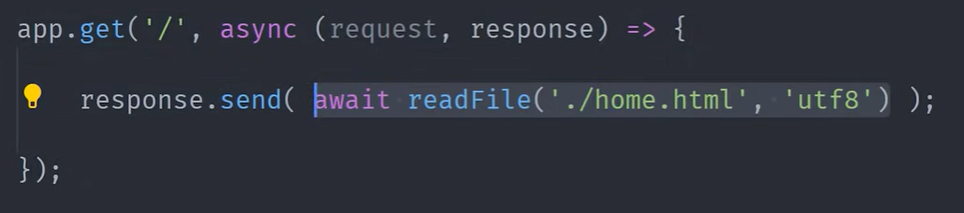
<label>Enter your name:</label>

<input *type*="text"></input>

</form>

</body>

</html>



# Npm

## Intro to Npm

Using NPM is free, publishing public packages is also free.

* the website
* the Command Line Interface (CLI)
* the registry

## Installation

Comes with Node

## Usage

Npm install express

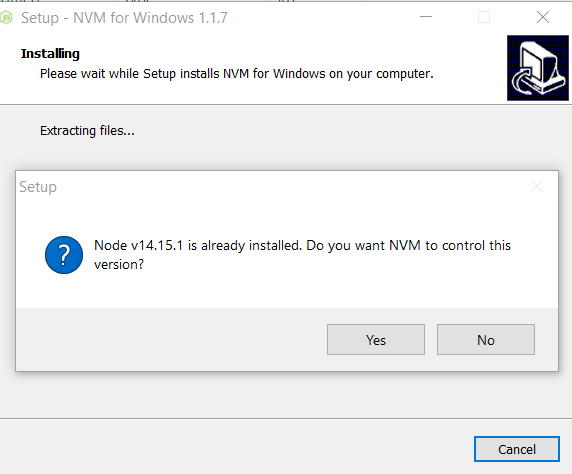
# Nvm

## Intro

NVM - node version manager. We might want to run several versions of node in one machine.

## Install

Go to: <https://github.com/coreybutler/nvm-windows#installation--upgrades>



## Usage

<https://dev.to/jlouiss/how-to-use-nvm-and-why-4e05>

# Webpack

## Credits

* Best short video on webpack thusfar: <https://www.youtube.com/watch?v=yR25JoybTxo>

## Intro to Webpack

* Most popular modules bundler. Fundamental operation - construct a dependency graph for your app (when your app depends on other libraries) and combine the app and the dependencies to be run on the browser.
* NOTE: when we develop row JS projects with webpack the following structure of the project is advised (index.html should be in the dist folder!):
  + *dist*
    - *index.html*
    - *main.js*
  + *src*
    - *script.js*
    - *… all the other js files, perhaps one per page (there are various good practices)*

## Installation

* *npm init → package.json*
* *npm install --save-dev webpack webpack-cli → install the libraries*

## Configration

* src/index.html

<!DOCTYPE *html*>

<html *lang*="en">

<head>

<meta *charset*="UTF-8" />

<meta *http-equiv*="X-UA-Compatible" *content*="IE=edge" />

<meta *name*="viewport" *content*="width=device-width, initial-scale=1.0" />

<title>Document</title>

<script *src*="../dist/main.js"></script>

</head>

<body>

Hi

</body>

</html>

* package.json

{

"name": "webpackdemo",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

**"build": "webpack",**

**"dev": "webpack serve"**

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"webpack": "^5.36.2",

"webpack-cli": "^4.7.0",

"webpack-dev-server": "^3.11.2"

},

"dependencies": {

"lodash": "^4.17.21"

}

}

* src/index.js .. we will use lodash for demo purposes.

import { camelCase } from 'lodash';

console.**log**(**camelCase**("Hello World"));

* configuration: **webpack.config.js**

const path = **require**('path');

module.exports = {

mode: 'development',

entry: './src/index.js',

output: {

path: path.**resolve**(\_\_dirname, 'destination'),

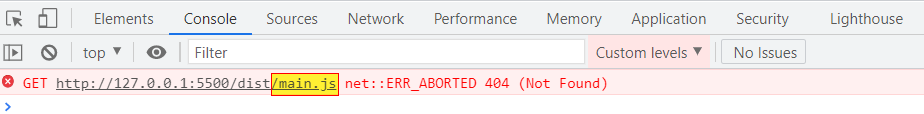
filename: 'app.js',

},

};

* after this you can run ***npm run build*** or ***npx webpack*** to invoke webpack:

Note, this was just a default config. If you see errors in the browser you need to adjust it (especially the filename and the “dist” parts!)



const path = **require**("path");

module.exports = {

mode: "development",

entry: "./script.js",

output: {

path: path.**resolve**(\_\_dirname, "dist"),

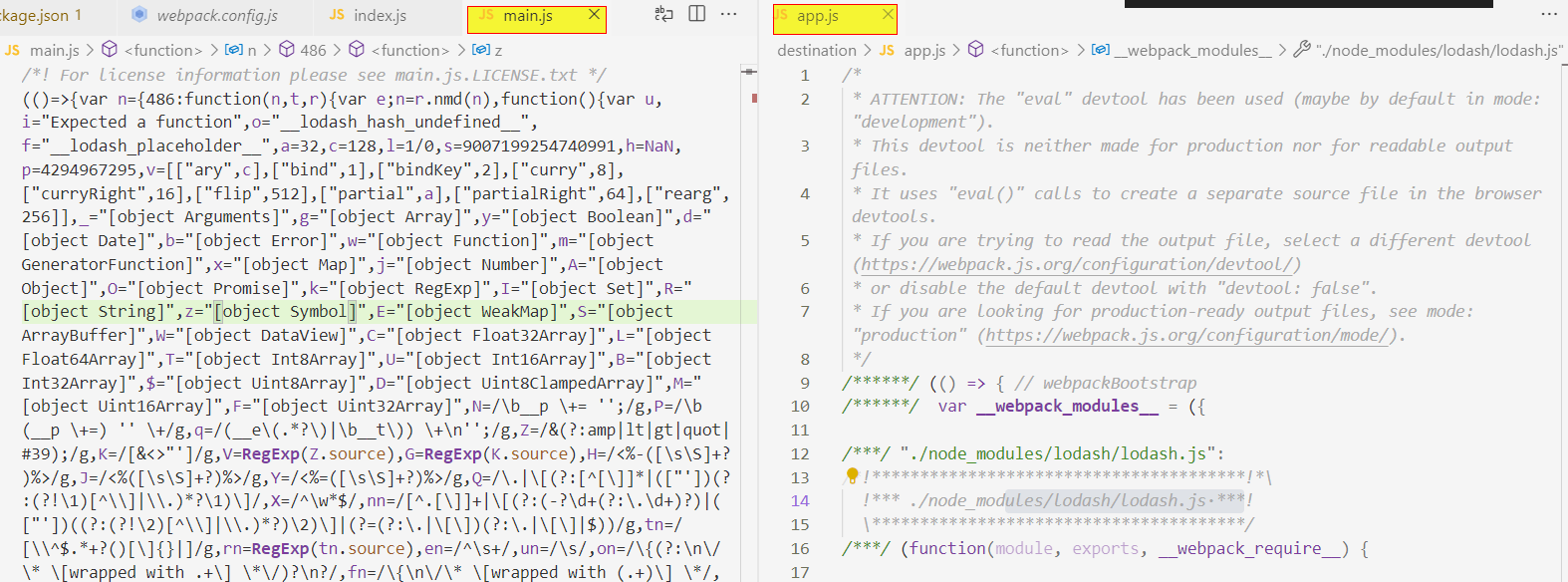
filename: "main.js", ← ERROR DUE TO THIS!!!

},

};

The configuration can be complex, for example we can build multiple modules: <https://stackoverflow.com/questions/35903246/how-to-create-multiple-output-paths-in-webpack-config>

* mode parameter controls the amount of optimizations performed, we can compare the files generated in production mode vs. development mode:



## File watching

* You will need to run ***npx webpack --watch***or configure it with package.jsonand launch it with VS Code live server.
* You can use this config (you don’t need live server for this):

const path = **require**("path");

module.exports = {

mode: "development",

entry: "./src/script.js",

output: {

path: path.**resolve**(\_\_dirname, "dist"),

filename: "main.js",

},

};

* Additionally you could add watch: true to your webpack config.

## Live server

* To install: ***npm install --save-dev webpack-dev-server***
* Run command: *npx webpack serve* or ***npx webpack serve --open*** *…* or you can configure it to use with npm.
* Config example:

const path = **require**("path");

module.exports = {

mode: "development",

entry: "./src/script.js",

output: {

path: path.**resolve**(\_\_dirname, "dist"),

filename: "main.js",

},

devServer: {

contentBase: path.**join**(\_\_dirname, "dist"),

port: 9000,

},

*// watch: true*

};

* //

{

"name": "webpackdemo",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"build": "webpack",

"dev": "webpack serve --open"

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"webpack": "^5.36.2",

"webpack-cli": "^4.7.0",

"webpack-dev-server": "^3.11.2"

},

"dependencies": {

"lodash": "^4.17.21"

}

}

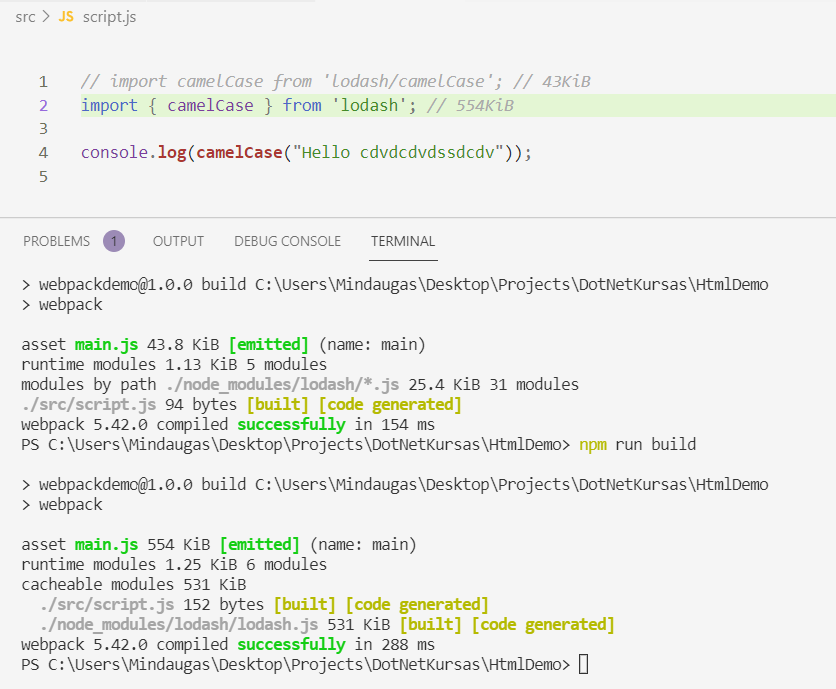
* //

import { camelCase } from 'lodash';

console.**log**(**camelCase**("Hello cdcdvdssdcdv"));

## Why use specific imports?

* Compare the file size when you include all of lodash and just specific functions that you need!
* Compare the following two imports:



## Minimize and obfuscate

## Bundle multiple js files