Nodejs

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

A Node.js app is run in a single process, without creating a new thread for every request. When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back. This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can also enable specific experimental features by running Node.js with flags

Node.js frameworks and tools

Node.js is a low-level platform. In order to make things easy and exciting for developers, thousands of libraries were built upon Node.js by the community.

Many of those established over time as popular options. Here is a non-comprehensive list of the ones worth learning:

- **AdonisJs**: A full-stack framework highly focused on developer ergonomics, stability, and confidence. Adonis is one of the fastest Node.js web frameworks.
- **Express**: It provides one of the most simple yet powerful ways to create a web server. Its minimalist approach, unopinionated, focused on the core features of a server, is key to its success.
- **Fastify**: A web framework highly focused on providing the best developer experience with the least overhead and a powerful plugin architecture. Fastify is one of the fastest Node.js web frameworks.
- **Gatsby**: A React-based, GraphQL powered, static site generator with a very rich ecosystem of plugins and starters.
- **hapi**: A rich framework for building applications and services that enables developers to focus on writing reusable application logic instead of spending time building infrastructure.

- **koa**: It is built by the same team behind Express, aims to be even simpler and smaller, building on top of years of knowledge. The new project born out of the need to create incompatible changes without disrupting the existing community.
- **Loopback.io**: Makes it easy to build modern applications that require complex integrations.
- **Meteor**: An incredibly powerful full-stack framework, powering you with an isomorphic approach to building apps with JavaScript, sharing code on the client and the server. Once an off-the-shelf tool that provided everything, now integrates with frontend libs React, Vue, and Angular. Can be used to create mobile apps as well.
- **Micro**: It provides a very lightweight server to create asynchronous HTTP microservices.
- **NestJS**: A TypeScript based progressive Node.js framework for building enterprise-grade efficient, reliable and scalable server-side applications.
- **Next.js**: A framework to render server-side rendered React applications.
- **Nx**: A toolkit for full-stack monorepo development using NestJS, Express, React, Angular, and more! Nx helps scale your development from one team building one application to many teams collaborating on multiple applications!
- **Sapper**: Sapper is a framework for building web applications of all sizes, with a beautiful development experience and flexible filesystem-based routing. Offers SSR and more!
- **Socket.io**: A real-time communication engine to build network applications.
- **Strapi**: Strapi is a flexible, open-source Headless CMS that gives developers the freedom to choose their favorite tools and frameworks while also allowing editors to easily manage and distribute their content. By making the admin panel and API extensible through a plugin system, Strapi enables the world's largest companies to accelerate content delivery while building beautiful digital experiences.

Installation

Official packages for all the major platforms are available at https://nodejs.org/en/download/.

Other package managers for Linux and Windows are listed in https://nodejs.org/en/download/package-manager/

Difference between Node.js and Browser

Both the browser and Node.js use JavaScript as their programming language.

Building apps that run in the browser is a completely different thing than building a Node.js application.

In the browser, most of the time what you are doing is interacting with the DOM, or other Web Platform APIs like Cookies. Those do not exist in Node.js, of course. You don't have the `document`, `window` and all the other objects that are provided by the browser.

And in the browser, we don't have all the nice APIs that Node.js provides through its modules, like the filesystem access functionality.

Since JavaScript moves so fast, but browsers can be a bit slow and users a bit slow to upgrade, sometimes on the web, you are stuck with using older JavaScript / ECMAScript releases.

You can use Babel to transform your code to be ES5-compatible before shipping it to the browser, but in Node.js, you won't need that.

V8 JavaScript engine

V8 is the name of the JavaScript engine that powers Google Chrome. It's the thing that takes our JavaScript and executes it while browsing with Chrome.

V8 provides the runtime environment in which JavaScript executes. The DOM, and the other Web Platform APIs are provided by the browser.

The cool thing is that the JavaScript engine is independent of the browser in which it's hosted. This key feature enabled the rise of Node.js. V8 was chosen to be the engine that powered Node.js back in 2009, and as the popularity of Node.js exploded, V8 became the engine that now powers an incredible amount of server-side code written in JavaScript

Other browsers have their own JavaScript engine:

- Firefox has **SpiderMonkey**
- Safari has **JavaScriptCore** (also called Nitro)
- Edge was originally based on **Chakra** but has more recently been rebuilt using Chromium and the V8 engine.
-

JavaScript is compiled or interpreted?

JavaScript is generally considered an interpreted language, but modern JavaScript engines no longer just interpret JavaScript, they compile it.

This has been happening since 2009, when the SpiderMonkey JavaScript compiler was added to Firefox 3.5, and everyone followed this idea.

JavaScript is internally compiled by V8 with **just-in-time** (JIT) **compilation** to speed up the execution.

How to run Node.js scripts from command line

The usual way to run a Node.js program is to run the node globally available command (once you install Node.js) and pass the name of the file you want to execute.

Syntax:

```
node filename.js
```

REPL in Node.js

REPL also known as Read Evaluate Print Loop is a programming language environment(Basically a console window) that takes single expression as user input and returns the result back to the console after execution.

Exporting modules

Module is a set of functions that we want to include in our application.

Node.js has a built-in module system.

A Node.js file can import functionality exposed by other Node.js files.

When you want to import something you use

```
const var-name=require("path of module");
```

When you assign an object or a function as a new `exports` property, that is the thing that's being exposed, and as such, it can be imported in other parts of your app, or in other apps as well. This can be done in two ways

Way-1:

Test.js

```
var test={a:10,b:20}

module.exports=test;
```

The first is to assign an object to `module.exports`, which is an object provided out of the box by the module system, and this will make your file export *just that object*.

While importing,

```
const testObj = require("./test");
```

way-2:

The second way is to add the exported object as a property of `exports`. This way allows you to export multiple objects, functions or data

Test.js

```
exports.test={a:10,b:20}
```

while importing

```
const testObj = require("./test").test;
```

So, the difference between above two approaches is, the first exposes the object it points to. The latter exposes *the properties* of the object it points to.

NPM(Node Package Manager)

"**npm**" is the standard package manager for Node.js.

In January 2017 over 350000 packages were reported being listed in the npm registry, making it the biggest single language code repository on Earth, and you can be sure there is a package for (almost!) everything.

It started as a way to download and manage dependencies of Node.js packages, but it has since become a tool used also in frontend JavaScript.

Installing all packages(dependencies) :

If a project has a `package.json` file, by running

```
npm install
```

it will install everything the project needs, in the `node_modules` folder, creating it if it's not existing already.

Installing a single package:

One can install a specific package by running

```
npm install <package-name>
```

Often you'll see more flags added to this command:

- `--save` installs and adds the entry to the `package.json` file *dependencies*
- `--save-dev` installs and adds the entry to the `package.json` file *devDependencies*

The difference is mainly that devDependencies are usually development tools, like a testing library, while `dependencies` are bundled with the app in production.

Local install vs global install

When you install a package using `npm` you can perform 2 types of installation:

- a local install
- a global install

local install:

By default, when you type an `npm install` command, like:

```
npm install package-name
```

the package is installed in the current file tree, under the `node_modules` subfolder.

As this happens, `npm` also adds the `lodash` entry in the `dependencies` property of the `package.json` file present in the current folder.

Global install:

A global installation is performed using the `-g` flag:

```
npm install -g lodash
```

When this happens, npm won't install the package under the local folder, but instead, it will use a global location.

Where, exactly?

The `npm root -g` command will tell you where that exact location is on your machine.

Package.json file

The `package.json` file is kind of a manifest for your project. It can do a lot of things, completely unrelated. It's a central repository of configuration for tools, for example. It's also where `npm` and `yarn` store the names and versions for all the installed packages.

there are *lots* of things going on here:

`version` indicates the current version

- `name` sets the application/package name
- `description` is a brief description of the app/package
- `main` set the entry point for the application
- `private` if set to `true` prevents the app/package to be accidentally published on `npm`
- `scripts` defines a set of node scripts you can run
- `dependencies` sets a list of `npm` packages installed as dependencies
- `devDependencies` sets a list of `npm` packages installed as development dependencies
- `engines` sets which versions of Node.js this package/app works on
- `browserslist` is used to tell which browsers (and their versions) you want to support

All those properties are used by either `npm` or other tools that we can use.

Uninstall npm packages

To uninstall a package you have previously installed **locally** (using `npm install <package-name>` in the `node_modules` folder, run

```
npm uninstall <package-name>
```

from the project root folder (the folder that contains the node_modules folder).

Using the `-S` flag, or `--save`, this operation will also remove the reference in the `package.json` file.

If the package was a development dependency, listed in the devDependencies of the `package.json` file, you must use the `-D` / `--save-dev` flag to remove it from the file:

```
npm uninstall -S <package-name>

npm uninstall -D <package-name>
```

If the package is installed **globally**, you need to add the `-g` / `--global` flag:

```
npm uninstall -g <package-name>
```

Node.js event loop

The **Event Loop** is one of the most important aspects to understand about Node.js

It explains how Node.js can be asynchronous and have non-blocking I/O, and so it explains, the thing that made it this successful.

The Node.js JavaScript code runs on a single thread. There is just one thing happening at a time.

Any JavaScript code that takes too long to return back control to the event loop will block the execution of any JavaScript code in the page, even block the UI thread, and the user cannot click around, scroll the page, and so on.

To deal with blocking operations, JavaScript is based so much on callbacks, and more recently on promises and async/await.

To understand event loop, first let us understand call stack.

Call stack:

The call stack is a LIFO queue (Last In, First Out).

The event loop continuously checks the **call stack** to see if there's any function that needs to run.

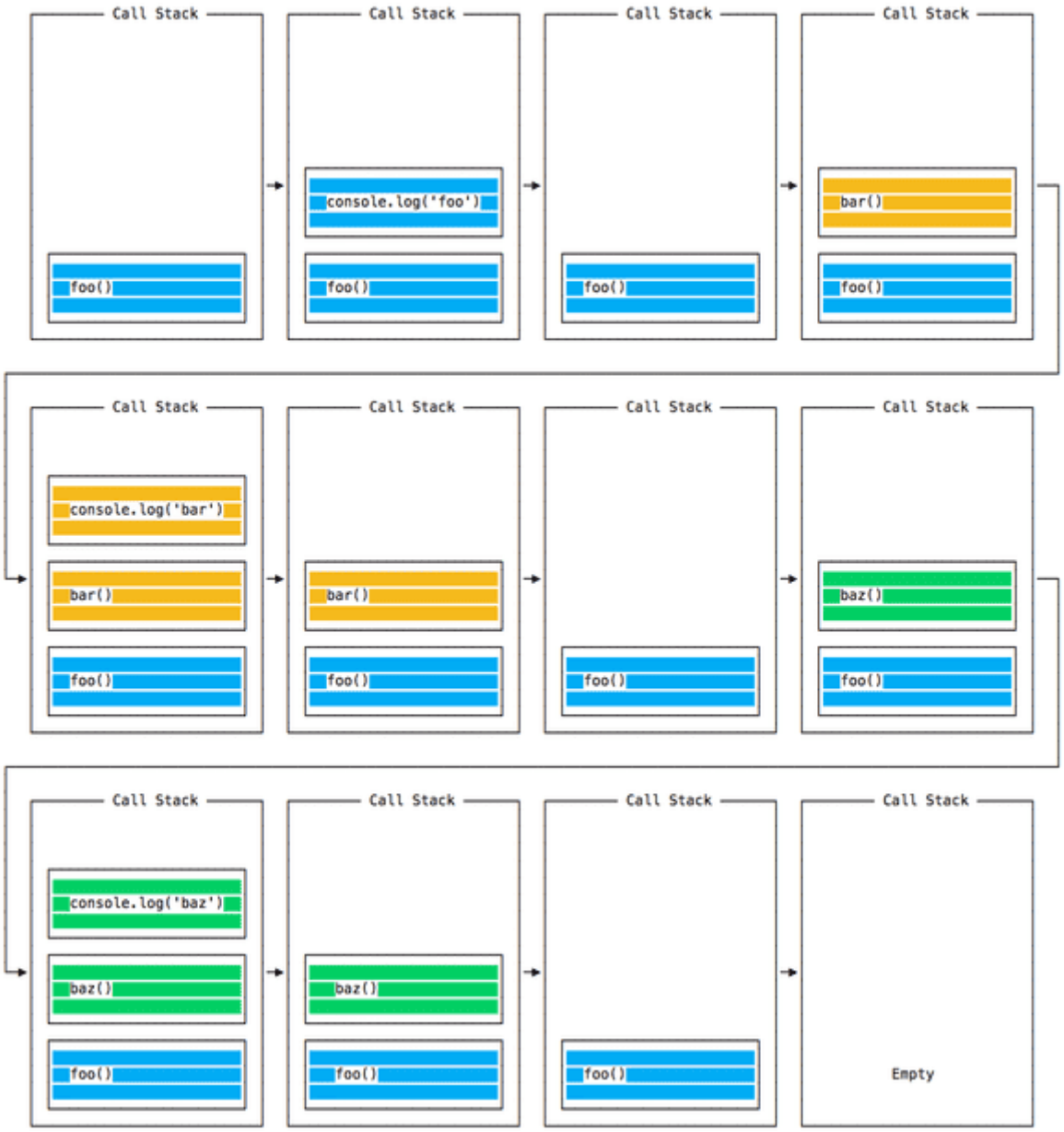
While doing so, it adds any function call it finds to the call stack and executes each one in order.

Let us understand call stack with simple example.

```
const bar = () => console.log('bar')

const baz = () => console.log('baz')

const foo = () => {

    console.log('foo')
```

```
    bar()
  baz()
  }

//calling foo()
foo()
```

At this point the call stack looks like this:


Call Stack
foo()
Call Stack
console.log('foo')
foo()
Call Stack
foo()
Call Stack
bar()
foo()
Call Stack
console.log('bar')
bar()
foo()
Call Stack
bar()
foo()
Call Stack
foo()
Call Stack
baz()
foo()
Call Stack
console.log('baz')
baz()
foo()
Call Stack
baz()
foo()
Call Stack
foo()
Call Stack
Empty

With this knowledge, let us understand how it deals with blocking requests.

```
const bar = () => console.log('bar')

const baz = () => console.log('baz')

const foo = () => {

        console.log('foo')

        setTimeout(bar, 0)

         baz()

        }


foo()
```

Here, calling bar() function is a blocking request. It will be handled by “Message queue” like below.

When setTimeout() is called, the Browser or Node.js starts the timer. Once the timer expires, in this case immediately as we put 0 as the timeout, the callback function is put in the **Message Queue**.

The Message Queue is also where user-initiated events like click or keyboard events, or fetch responses are queued before your code has the opportunity to react to them. Or also DOM events like `onClick`.

The loop gives priority to the call stack, and it first processes everything it finds in the call stack, and once there's nothing in there, it goes to pick up things in the message queue.

> Read this article https://dev.to/cpuram1/nodejs-event-loop-architecture-53jn for more information about event loop.

JavaScript timers

- **setTimeout()**

When writing JavaScript code, you might want to delay the execution of a function.

This is the job of `setTimeout`. You specify a callback function to execute later, and a value expressing how later you want it to run, in milliseconds

Syntax:

setTimeout(function, time delay in milli seconds)

- **setInterval()**

This function similar to `setTimeout`, with a difference: instead of running the callback function once, it will run it forever, at the specific time interval you specify (in milliseconds)

JavaScript asynchronous programming and callbacks

Asynchronous means that things can happen independently of the main program flow.

JavaScript is **synchronous** by default and is single threaded. This means that code cannot create new threads and run in parallel. Lines of code are executed in series, one after another.

Normally, programming languages are synchronous and some provide a way to manage asynchronicity in the language or through libraries. C, Java, C#, PHP, Go, Ruby, Swift, and Python are all synchronous by default. Some of them handle async by using threads, spawning a new process.

To deal with asynchronous activities, callbacks can be used.

Callbacks:

A callback is a simple function that's passed as a value to another function, and will only be executed when the event happens.

For example, if we want a function to execute only when user click on a button, then

```
document.getElementById('button').addEventListener('click', () => {
  //item clicked
})
```

Here, the second argument for “addEventListener()” is a callback which will be executed only when “click” event fires.

Another example with timer function is

```
setTimeout(() => {
        // runs after 2 seconds
}, 2000)
```

Here also, the argument function of “setTimeout” is callback which will be executed after 2 seconds.

Reading & Writing data of a file

Module: "fs"

Install: There is no need to install it. Being part of the Node.js core, it can be used by simply requiring it:

Import:

```
const fs=require("fs")
```

reading a file:

```
readFile("path of file ",callback);
```

Ex:

```
readFile("path of file", (err , data)=>{
    if(err)
        {
        console.log(err);
        Return;
        }
    console.log(data);
})
```

Writing data to a file:

```
writeFile("path of file", content,callback);
```

appending to a file:

```
appendFile("path of file",content,callback)
```

Note: All above methods will read or write or append the full content of the file in memory before returning the data. For large files, This means that big files are going to have a major impact on your memory consumption and speed of execution of the program.

In this case, a better option is to read the file content using streams.

Reading a file using stream:

Syntax: createReadStream("path of file")

Ex:

```
const fs=require("fs");
var stream= fs.createReadStream("path of file");
stream.on("data",(data)=>{
        var chunk=data.toString();
        console.log(chunk);
})
```

Writing a file using stream:

Syntax: createWriteStream("path of file")

Ex:

```
var stream=fs.createWriteStream("path of file");
stream. write("data.....");
```

Copying data of one file to another using pipe():

Ex:

```
var readStream=fs.createReadStream("source file");
var writeStream=fs.createWriteStream("target file");
readStream.pipe(writeStream)
```