Extra Notes - Node.js - An Introduction to Node.js and NPM

- Dr Nick Hayward

A brief introduction to Node.js, including initial setup and package management with NPM.

Contents

Intro and outline Node.js is, in essence, a JavaScript runtime environment designed to be run outside of the browser.

It has been designed as a general purpose utility, and can be used for many different tasks including,

- asset compilation
- monitoring
- scripting
- web servers

With Node.js, we now see JavaScript moving from client-side to a support role in back-end development.

One of the key advantages touted for Node.js is its speed. A primary reason for this speed boost is its underlying architecture. It uses an **event-based** architecture instead of a threading model popular in compiled languages. So, Node.js uses a single event thread by default, and all I/O is asynchronous.

Conceptual model for processing In essence, how does Node.js, and its underlying processing model, actually work.

The client sends a hypertext transfer protocol, HTTP, request, or customarily requests, to the Node.js server. The event loop is then informed by the host OS, which passes applicable request and response objects as JavaScript closures to associated worker functions with callbacks. Any long running jobs continue to run on various assigned worker threads, and responses are sent from the non-blocking workers back to the main event loop via a callback. The event loop returns any results back to the client, effectively when they're ready.

For example,

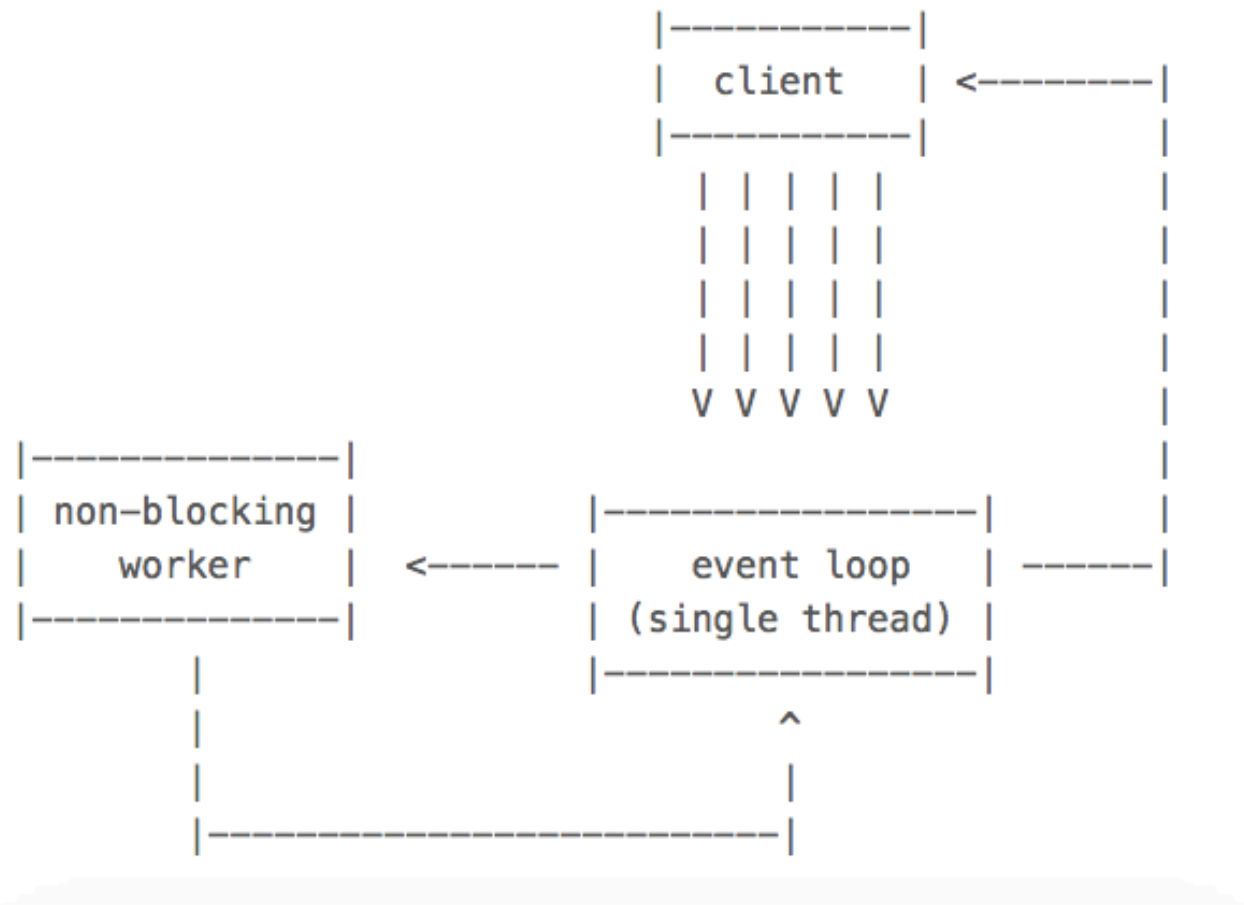

Figure 1: Node.js - conceptual model for processing

Event-driven architecture JavaScript, by its very nature, was originally designed to work within the confines of the web browser. It had to handle the restrictive nature of a single thread and single process for the whole page. Therefore, synchronous blocking in code would lock up a web page from all actions. JavaScript was built with this in mind.

Due to this style of I/O handling, Node.js is able to handle millions of concurrent requests on a single process. This has been added, using libraries, to many other existing languages, including

- Akka for Java
- EventMachine for Ruby
- Twisted for Python
- …

but JavaScript syntax already assumes events through its use of callbacks. However, as a note of caution, if a query etc is CPU intensive instead of I/O intensive, the thread will be tied up and everything will be blocked as it waits for it to finish.

Callbacks In most languages, you send an I/O query and wait until a result is returned before you can continue your code procedure. For example, if you submit a query to a database for a user ID, the server will pause that thread/process until the database returns a result for the ID query.

In Javascript, this concept is rarely implemented. Instead, it is more common to pass the I/O call a *callback*. This *callback* will need to run when the task is completed. For example, find a user ID and then do something, such as output to a HTML element.

The biggest difference in these approaches is that whilst the database is fetching the user ID query, our app is free to do whatever else might be useful. We might accept another web request, listen to a different event, and so on. This is one of the reasons that Node.js returns good benchmarks and is easily scaled.

This makes Node.js well suited for I/O heavy and intensive scenarios, including web development.

Install and setup There are a number of different ways to install *Node.js*, *npm*, and the lightweight, customisable web framework **Express**.

To run and test Node.js on a local Mac OS X, Linux, or Windows machine, simply download and install a package from the following URL,

- Node.js - download

Node.js package manager, **npm**, is included with the default installers available at the Node.js website.

We can check it has been installed successfully with the following terminal commands,

```
node -v
npm -v
```

Each of these commands will return the version number for their respective install.

NPM To install existing **npm** modules, use the following type of command

```
npm install express
```

This will install the module named `express` in the current directory. It will act as a local installation within the current directory, installing in a folder called `node_modules` . This is the default behaviour for current installs.

We can also specify a global install for modules. For example, we may wish to install the **express** module with global scope

```
npm install -g express
```

The `-g` option sets a flag for Express to global instead of a limited local directory install.

We can then import, or effectively add, installed modules in our Node.js code using the following declaration as an example,

```
var module = require('express');
```

When we run this application, it will look for the required module library and its source code.

Find modules with NPM The official online search tool for **npm** can be found at

- npmjs

Top packages include options such as

- browserify (helps us bundle require modules in the browser…)
- express (a lightweight web application framework for Node.js)
- grunt (a task runner to help with automation of various development processes…)
- bower (a package manager for web development…)
- karma (a JS test runner…)

and many more…

We can also search for modules directly from the command line using the following command,

```
npm search express
```

This will return results for module names and descriptions matching *express*.

Specifying dependencies For Node.js applications, we can ease their installation by specifying any required dependencies in an associated `package.json` file. This allows us, as developers, to simply specify the modules to install for our application, which can then be run using the following command

```
npm install
```

This helps reduce the need to install each module individually, and helps other users install an application as quickly as possible. Also, our application's dependencies are stored in one place.

Package JSON file An example `package.json` file might be as follows,

```
{
  "name": "app",
  "version": "0.0.1",
  "dependencies": {
    "express": "4.2.x",
    "underscore": "-1.2.1"
  }
}
```

The `package.json` file helps us manage and track dependencies within our app. It also helps us track other aspects of our application, including name, description, version, and so on.

An example `package.json` file might be as follows

```
{
  "name": "test-node-app",
  "version": "1.0.0",
  "description": "test app for nodejs",
  "main": "server.js",
  "dependencies": {
    "body-parser": "^1.14.1",
    "express": "^4.13.3",
    "redis": "^2.3.0"
  },
  "author": "ancientlives",
  "license": "ISC"
}
```

We could create `package.json` by hand or use the following command to walkthrough various options

```
npm init
```

This creates a `package.json` file for us, and then we can add any required dependencies as follows,

```
npm install redis --save
```

This will update our `package.json` file for the installed package.

References

- Node.js
  - Node.js home
  - Node.js - download