NodeJS

A Quick Introduction

Quick Intro

Tobias Gray

Senior DevOps Engineer BPDTS

Recent Work

- DWP - Migrating apps to AWS
- Turnitin - Rewriting Legacy Codebase for AWS

Agenda

- The birth of NodeJS
- What's great about NodeJS
- What's not so great about NodeJS
- How it looks - code snippet
- Demo
  - NPM
  - A simple web server (express)
  - Common packages (lodash, moment etc.)
- The ecosystem - linting, testing, security etc.

The Birth of NodeJS

- ~2009 JavaScript in the browser was seeing use cases for more complex situations
- In 2009 NodeJS was developed
  - Written using C it combined the Google V8 engine, an event loop and low level I/O API
  - Non blocking (async)
  - Allowed JavaScript to be ran outside the browser
  - This allowed developers to write command line tools or server side scripts using JS

What's Great About NodeJS

- Context switching
- Fast(ish)
- Large community
- Lots of tooling, packages
- Fast paced & ever evolving

- Context switching between frontend and backend can be a drain and NodeJS helps alleviate this.
- NodeJS is fast especially when it comes to I/O and thanks to it being async it's also non blocking (note not fast for CPU intensive tasks see next slide).
- Large community and conferences available:
  - https://events.linuxfoundation.org/events/node-js-interactive-2018/
  - http://www.nodesummit.com/
  - https://www.nodeconf.eu/
  - And more local ones on https://www.meetup.com/ etc.
- Lots of tools/packages/libraries/frameworks for all sorts of tasks. Checkout https://www.npmjs.com/ to start scratching the surface.
- Fast paced releases https://github.com/nodejs/Release ever since the nodejs foundation took over. That along with new frameworks and packages available every day, though some may consider this ever shifting ground a downside.

What's Not So Great About NodeJS

- Single threaded
- Not suited for CPU intensive tasks

- Rule - Always use the best tool for the job and NodeJS may not be it
- JavaScript is single threaded so can't take advantage of multiple cores that most compute environments have these days.
    - However this can be mitigated by running multiple services i.e. microservices repeated for scalability and fault tolerance using tools such as Docker and a container orchestration tool for example.
    - JavaScript (and in turn NodeJS) is async so while being single threaded processes each get a slice of time on the CPU and don't clock each other.
    - Multi threaded programs can lead to complexity, you may think of the limitation as a plus?
- Not suited for CPU intensive tasks, If you want something even more performant a language like C or GoLang may be a better choice.
    - NodeJS will however be able to handle the vast majority of day to day workloads.

If it looks like a duck and quacks like a duck, it's a duck

```
var utilObject = {
    add: function (a, b) {
        return a + b;
    }
}

module.exports = utilObject;
```

```
var utilObject = {
    add: function (a, b) {
        return a + b;
    }
}
```

```
<script src="utils.js"></script>
```

- Two things to take from this, 1) NodeJS and JS in the browser can be near to identical though there are differences and 2) JavaScript is a loosely typed language so if you come from a Java world or other language that defines types things may seem strange at first.

1. NodeJS and JS in the browser can be near to identical though there are differences
    a. NodeJS does not suffer from needing to wait for browsers to catch up, the Node version you run you can controll so able to user new ES6 features without transpilers like Babel
    b. Node JS uses module.exports and require while traditional JS uses namespaces and script tags. Once again modern transpiled browser JS can make the two much more similar
    c. NodeJS has extra features such as I/O to read/write files while Browser JS doesnt (security limitations)
2. JavaScript is a loosely typed language
    a. JavaScript (NodeJS) does not define variable types. At first this may seem risky if you come from a strongly typed language but it's something you get used to.

Demo - A simple API server

- Running “npm init” to setup a project
  - What is NPM (Node Package Manager), checkout https://www.npmjs.com
  - Describe “package.json” - scripts, project info, (dependencies later)
- Create a simple hello world node script
- Install express using npm
  - Describe dependencies in “package.json”
  - Describe “package.lock.json”
- Create a web server
  - Require express
  - Init an express app “var app = express()”
  - Start listening on a port
  - Go to localhost:port to see server is running
- Add an endpoint
  - Add root endpoint to print hello world to webpage
  - Demo index/home page
  - Add GET API endpoint to show the server time
  - Install moment and require it, use it to print a formatted date
  - Semo new endpoint
- Add a POST endpoint
  - Install “body-parser” for express
  - Init/configure express to use body parser “app.use(bodyParser.json());”
  - Add POST request handler
    - Example of getting data from request body

        - Example of getting data from URL
- Overview of what we have made and where to go next
    - Split files so not in one mammoth file
    - Express router enhancements
    - Tests, linting etc.

The Ecosystem

- Testing - Mocha, Chai, Jest, Sinon, Jasmine, AVA, NYC
- Security Checks - npm outdated, npm audit
- Linting - ESLint, JSHint, JSLint
- Docker - Alpine and Full fat images
- Package managers - NPM, Yarn

Questions