INTRODUCING NODE.JS

Agenda

- Understand the power of Node.js
- Discuss Node.js architecture
- Use nvm & npm
- Understand module system
- Control flow strategies
- General topics

What

- Node.js is a server side JavaScript platform
- Built on Chrome's V8 engine
- Is open source
- Single threaded
- Event-driven, non blocking I/O
- Developed in 2009 by Ryan Dahl
- Supported by Joyent

Node.js as BFF

- Node.js cannot easily replace existing server side infra written in Java/.NET
- Common scenario is to put Node.js at the front of Java/.NET → Backend for Frontend
- Usually is controlled by Front End engineers
  - Thus allowing the developer to push JavaScript code to the server
  - Improve client side performance

Node.js as Development Tools

- This is where Node.js really shines
- Extreme echo system of development tools
  - Build tools – Webpack, Gulp, Grunt
  - Compilers – Typescript, Babel
  - Testability – Selenium, Jasmine, Mocha
  - Desktop applications – VSCode, GithubDesktop

When should we use?

- Node.js is great when most work is I/O
- Think of a web server. The “hard” work relates to
  - HTTP → Networking I/O
  - Database → Networking I/O
  - File system
- The server is more of a controller/facade

When NOT to use

- Heavy server-side computation
  - Can offload the “hard” work to background processes
  - Can use threads (not common)
- Direct access to OS API is required
  - Can integrate C/C++ code

Installation

- Depends on the OS
- Starts with https://nodejs.org
- Amazingly you can just download Node.js as a tar/zip file and start using it
  - https://nodejs.org/dist/latest-v8.x/
- On Windows you may execute nodevars.bat which fixes the PATH with
  - node
  - npm

NVM

- Each Node.js project may be dependent on different Node.js version
- Can resolve that by installing Node.js per project
  - Less common
- NVM allows managing multiple versions of Node.js at the machine level while having only ONE active version at a time

NVM

- Ensure you don't have any previous installation of Node.js
- nvm list – Get a list of all installed versions
- nvm install latest – Installs latest Node.js version
- nvm use 9.8.0 – Configure machine to use the specified version

Hello World Sample

- Create new main.js file
- Paste the following

```
console.log("Hello Node.js");
```

- From the command line execute

```
node main.js
```

- Can it be simpler ?

Http Server Sample


```
const http = require('http');

const requestHandler = (req, res) => {
   res.end('Hello Node.js Server!');
}

const server = http.createServer(requestHandler);

server.listen(3000, (err) => {
   if (err) {
      return console.log('something bad happened', err);
   }

   console.log(`server is running`);
});
```

Better abstraction with Express

- npm install express

```
const express = require("express");

const app = express();

app.get("/api/contact", function (req, res) {
   res.json([
      {id: 1, name: "Ori"},
      {id: 2, name: "Roni"}
   ]);
});

app.listen(3000, function() {
   console.log("Server is running");
});
```

Toolings

- But what if we just need a simple web server that returns static content from current directory
- No need to re-implement that
- npm install http-server
- node_modules/.bin/http-server
- A web server is up and running on port 8080 …

Typscript

- Adds type safety to Node.js
- npm install typescript
- npx tsc -init
- npm install @types/node
- npx tsc

Typescript generates compilation error ☺

```
import * as fs from "fs";

fs.readFile("main.ts", function(err, data: string) {
    console.log(data);
});
```

NODE.JS ARCHITECTURE

Agenda

- Discuss Node.js architecture
- Understand main characteristics
- Write some code

Characteristics

- Built on Chrome's V8 engine
- Uses libuv
- Single threaded
- Event-driven
- Non blocking I/O

V8

- JavaScript engine
- Compiles JS to native machine code
- Written in C++
- Used in Chrome & Node.js
- Supports Windows, macOS, Linux
- Can be embedded into C++
- Hello world sample

V8 vs. The World

- Same role as Java's JVM or .NET's CLR
- However, JavaScript is dynamic language
- Therefore less optimization opportunities
- V8 profiles code at runtime and optimizes it
  - Same as Java HotSpot technique
  - Has two compilers Full-Codegen & Crankshaft
  - Therefore can be faster than GCC
  - See some benchmarks

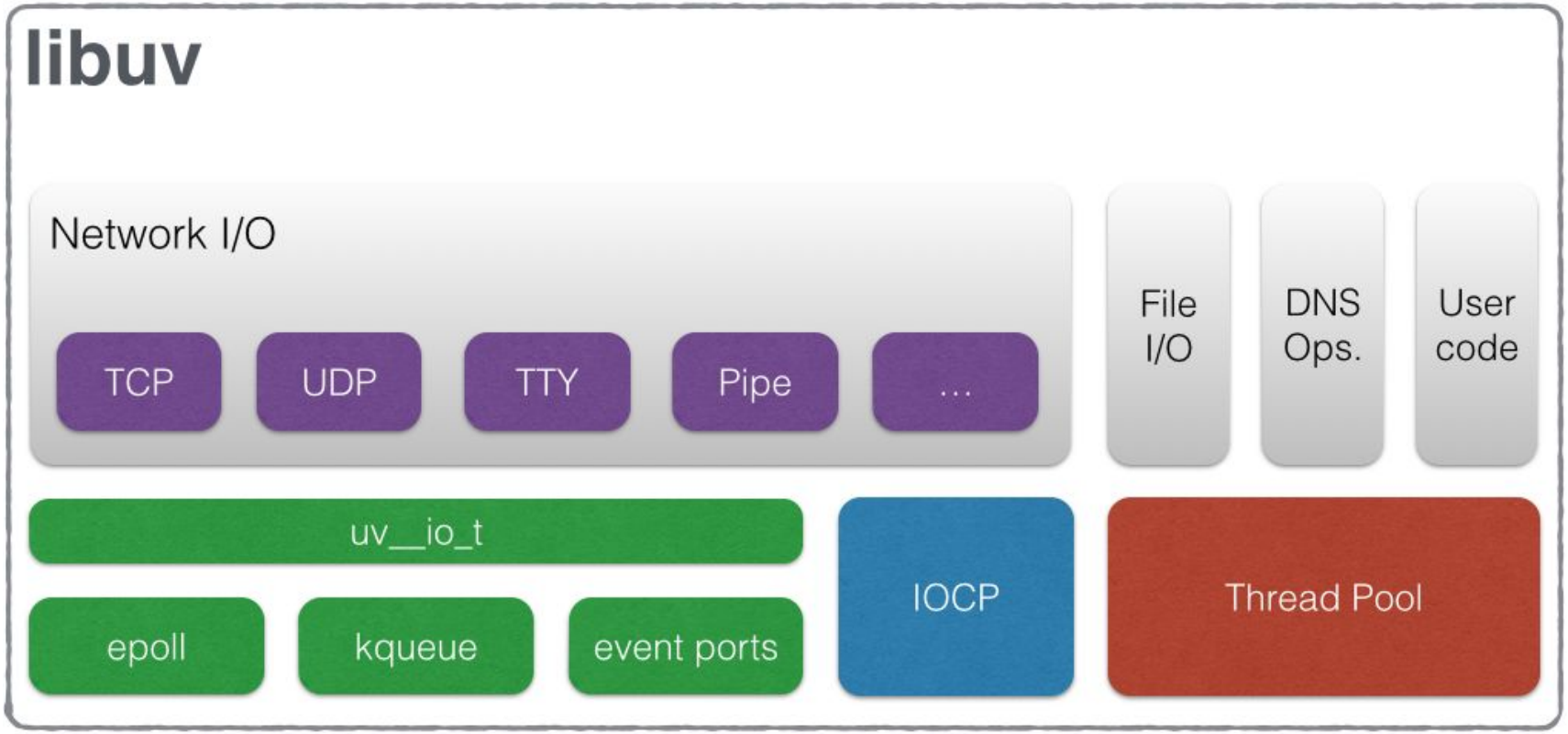
libuv

- Multi platform library with focus on asynchronous I/O
- Was developed for use by Node.js
  - But is now used by others
- Supports all the goodies of Node.js
  - Event loop
  - Async TCP & UDP sockets
  - Async file system operations
  - IPC
  - More …
- Create thread sample

libuv

- When possible uses OS asynchronous API
- Surprisingly does not use asynchronous file I/O
  - Code complexity
  - Poor APIs
  - Poor implementation
- Uses thread pool instead

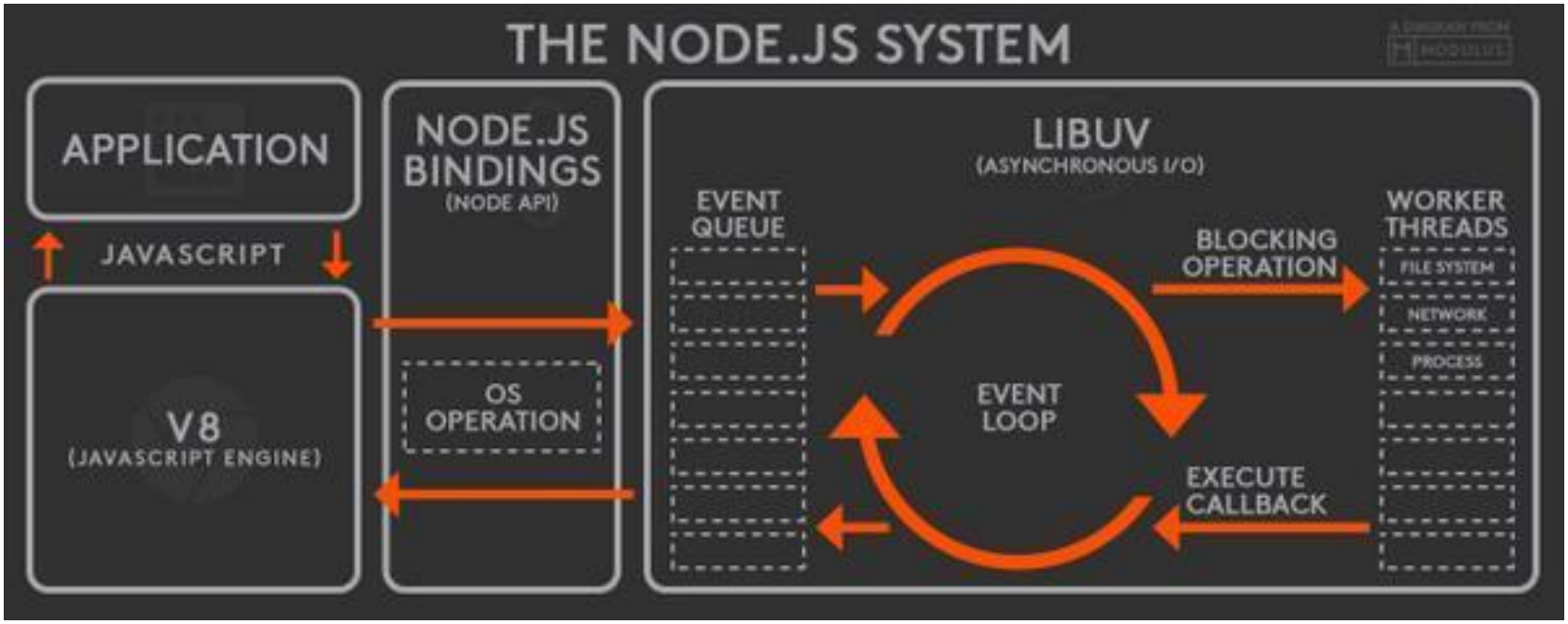
libuv



Integrating

- Take V8
- Combine it with libuv
- Implement some JavaScript API to be consumed by the application
- And voila … Node.js

Node.js Architecture



Traditional Web Server

- Spawns new thread for each request
  - May use some kind of thread pool
- Each thread consumes memory and increases context switching
- Thread blocks when accessing file system/networking
- Programmer must synchronize access to shared/static data
  - Thus increasing even more blocking time

Single Threaded

- Only JavaScript code is Single Threaded
  - NodeJS has multiple worker threads

```
setTimeout(function() {
   console.log("timeout");
}, 1000);

console.log("Before");
sleep(2000);
console.log("After");

function sleep(ms) {
   const before = new Date();

   while(new Date() - before < ms);
}
```

Before
After
timeout

Event Queue

- Continuing with our previous sample
- What happens after 1000 milliseconds ?
- A worker thread handles the timer event by putting an appropriate event inside the queue
- Only when our JavaScript code completes it returns to the event loop and fetches the next waiting event

Asynchronous I/O

- Node.js uses callbacks to handle async operations
- The function returns immediately and the “real work” executes at the background
- Once completes, an event is pushed to the event queue waiting to be processed by the main thread

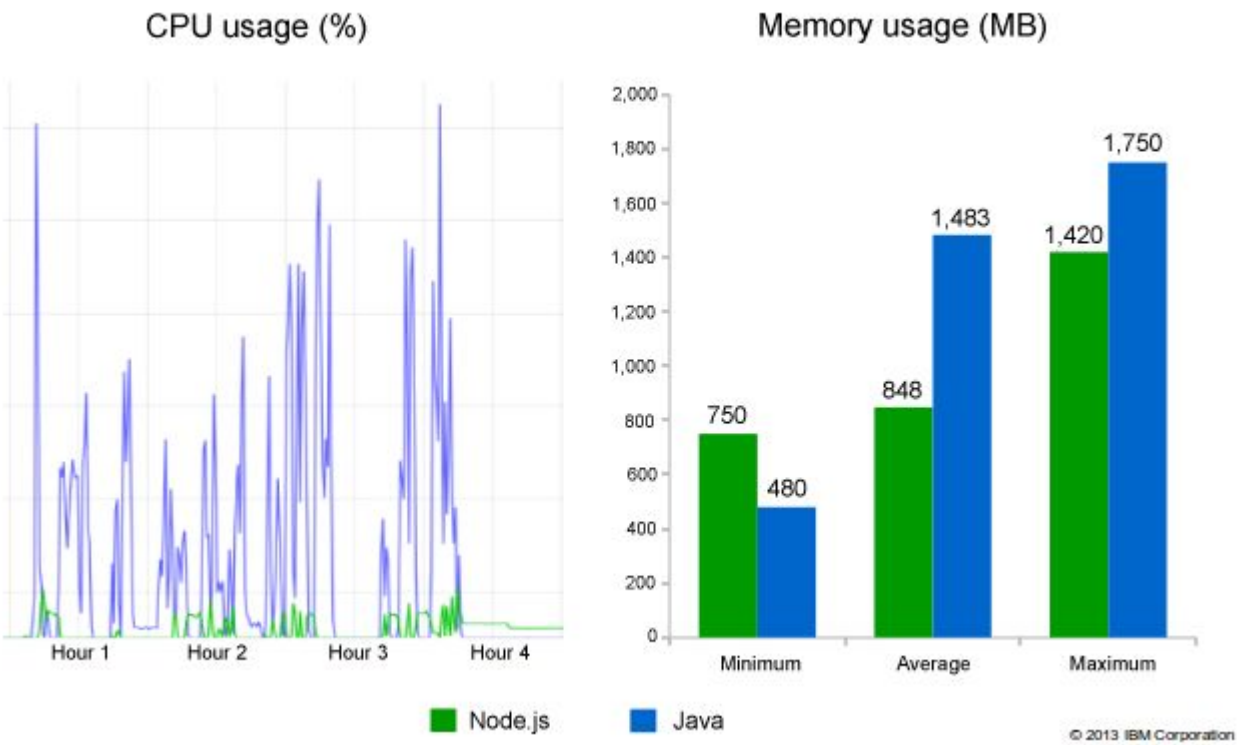
```
const fs = require("fs");

fs.readFile("main.js", function(err, buffer) {
   if(err) {
      return;
   }

   console.log(buffer.toString());
});
```

Performance

4



REPL

- Execute “node” and then enter
- Interactive mode
- Write and evaluate JavaScript code

```
> node
> 8 + 5
13
>
```

Debugging

- node --inspect --inspect-brk main.js
- Open Chrome at chrome::/inspect
- Wait for remote target list to refresh
- Click inspect
- Use Console/Sources/Memory tabs

Native Modules

- When Node.js public API is not enough you may implement native modules which access OS directly
- Not straightforward ☹
- Need to write cross platform C++ code
  - May use libuv to achieve that
- Must use V8 APIs to interact with JavaScript code
  - V8 changes a lot over time
  - Thus, native module tend to break cross Node.js versions

C++ Addon


```
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void Method(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  args.GetReturnValue().Set(String::NewFromUtf8(isolate, "world"));
}

void init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method);
}

NODE_MODULE(NODE_GYP_MODULE_NAME, init)

}  // namespace demo
```

Compile the Addon

- Create binding.gyp file

```
{
  "targets": [
    {
      "target_name": "addon",
      "sources": [ "hello.cc" ]
    }
  ]
}
```

- npm install –g node-gyp
- node-gyp configure → Makefile/vcxproj file is created
- node-gyp build → addon.node is created

```
const addon = require('./build/Release/addon');
console.log(addon.hello());
```

CONTROL FLOW

The Challenge

- Node.js asynchronous nature impose non intuitive programming model

Callback hell

```
function readFileIfExists(filePath, callback) {
  fs.stat(filePath, function (err, stat) {
    if (err) {
      callback(err);
      return;
    }

    if (stat.isFile()) {
      fs.readFile(filePath, function (err, data) {
        if (err) {
          callback(err);
          return;
        }

        callback(null, data.toString());
      });
    }
  });
}
```

async package

- Async utilities for node and the browser
- npm install async

```
function readFileIfExists(filePath, cb) {
    async.seq(
        fs.stat,
        (stat, cb) => stat.isFile() ? fs.readFile(filePath, cb) : cb(new Error("Not a file")),
        )(filePath, cb);
}
```

Promise Flow

- Convert each function to promise based

```
function readFile(filePath) {
    return new Promise((resolve, reject)=> {
        fs.readFile(filePath, function(err, data) {
            if(err) {
                reject(err);
                return;
            }

            resolve(data);
        });
    });
}
```

- Can wrap that logic inside a promisify helper

promisify


```
function promisify(func) {
   return function (...args) {
      return new Promise((resolve, reject) => {
         args.push(callback);
         func.apply(this, args);

         function callback(err, res) {
            if(err) {
               reject(err);
               return;
            }

            resolve(res);
         }
      });
   }
}
```

es6-promisify package offers an almost identical function

Use the new functions


```
function readFileIfExists(filePath) {
   return stat("1.txt").then(stat => {
      if (stat.isFile()) {
         return readFile(filePath);
      }

      throw new Error("Not a file");
   });
}
```

Return a promise to allow "continuation"

Must throw exception to signal an error

Callback hell ?

- The promise flow simplifies code since middle layers does not have to deal with errors
- However, the code still suffers from the callback hell

```
function readFileIfExists(filePath) {
   return stat("1.txt").then(stat => {
      if (stat.isFile()) {
         return readFile(filePath);
      }

      throw new Error("Not a file");
   });
}
```

async/await

- Code feels almost synchronous

```
async function readFileIfExists(filePath) {
    const info = await stat(filePath);
    if(!info.isFile()) {
        throw new Error("Not a file");
    }

    return await readFile(filePath);
}
```

```
async function main() {
    try {
        const data = await readFileIfExists("1.txt");
        console.log(data.toString());
    }
    catch(err) {
        console.error(err);
    }
}
```

Promise Flow

- Unfortunately most Node.js APIs are callback based
- Need to manually wrap the code
- Be careful when wrapping instance methods
  - Must keep the correct this

```
const obj = {
   id: 123,
   oldStyle: function(callback) {
      callback(null, this.id);
   }
};

const newStyle = promisify(obj.oldStyle.bind(obj));
```

Promise Limitation

- Promise can be resolved only once
- Therefore, it cannot represent a recurring event
  - Stream
  - Button clicks
- Runs immediately

GENERAL TOPICS

Auto Restart

- When developing a web server it is convenient that the server is automatically restarted with each code modification
- npm install nodemon
- node_modules/.bin/nodemon main.js
- Other alternatives
  - forever
  - pm2

Summary

- Node.js is a lean platform
  - Less Than 20MB of installation
- Easily installed and getting started
- Lot's of open source packages
  - Some time its hard to choose the right one

Building RESTful APIs

Agenda

- Discuss architecture
- Error handling pointers
- Performance pointers
- Security pointers

Architecture

Main concerns

- There are a few main concerns when building an Express app
  - Durability – error handling, graceful shutdown
  - Security – request sanitizing, SSL
  - Performance – compression, async code
  - Maintainability – code structuring, testing
- There many other concerns that should be addressed, could you suggest one?

Separation of concerns

- Try not to be naive when designing an app
- Separate network concerns & API declaration
- Use Express for it's fundamental http / web application features. That's it!
- Keep Express within its boundaries
  - Separate middleware and business logic
- Split the app into components
  - Will be discussed later on

Naive approach

- A common implementation of an express app mixes all the layers in one big horrible mess

```
app.get('/user/:id', async (req, res) => {
    try {
        const user = await DAL.getUserById(req);  // returns User

        res.json(user.toJSON());
    } catch(e) {
        console.error('Failed to fetch user with error', e);

        res.status(500).send('Whoops, something went terribly wrong');
    }
});
```

Naive approach

- "Naive" implementation will lead to
  - Coupling with Express implementations
  - Boilerplate when writing tests
  - Lesser test coverage reports
  - A less maintainable codebase

Layering approach

- A more common hard headed approach will be to separate the app into component and then into layers
  - Router – web handler
  - Controller – mediation
  - Service – business logic
  - Model – data access
- Controller and service may be unified in smaller applications
- Can you think of the benefits?

Layering approach - PROS

- Decoupling from specific implementations
  - Better migration options (Koa, Hapi, Socket.io)
- Better testing options for each layer

Layering approach - CONS

- May lead to A LOT of boilerplate
  - Code spaghetti may be just around the corner
  - Duplication of code
  - More folders, more files -> more code to maintain
- Can you think of any other disadvantages?

Error handling

Error handling - general

- Always use a mature logger like Winston / Bunyan
  - Eliminate console.log / console.error from your code. It is synchronous!
- When in-doubt, gracefully restart
- Handle your code centrally, prevent handling code duplication
- Make sure to monitor with an APM tool

Error handling - Express

- Validate request input using a dedicated library
  - Joi will do the trick
- Avoid “on the spot” error handling
- Handle errors centrally
  - Reduces error handling code duplication
  - Express provides us with a middleware for error handling
- Distinguish between operational and internal errors

Error handling middleware

- Writing a naive error handling middleware is pretty straight forward

```
app.use(function errorHandler(err, req, res, next) {
    const error = "Huston, we have an error: " + err;

    logger.log('error', error);
    mailer.report().error('fatal', error);

    res.status(500);
    res.send('error', { error: err });
}
```

- Notice that the middleware accepts four arguments

Performance

Performance

- Use gzip to compress response body
- Do not block the loop, use async only functions
  - Use an async parsers to parse requests
  - Run your app with --trace-sync-io to print a warning every time it uses a sync API
- Delegate anything possible to a reverse proxy
  - Node is awful at doing CPU intensive tasks
  - Including gzip compression, SSL termination, throttling requests and static file serving

Performance

- Try and stay stateless, try and restart daily
- Monitor the heap - process.memoryUsage()
  - Javascript code has a tendency to leak
- Don't forget to NODE_ENV=production

Security

Security

- Do not expose your errors
  - May reveal information about your service
- Only use secure cookies
- When in doubt, use a helmet (middleware)
  - Mitigates many common attack vectors
  - Really easy to implement
  - https://github.com/helmetjs/helmet