Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU44000 Internet Applications

Week 2 Lecture 2

Conor Sheedy

Modules in Browser Javascript

In a browser,

- a loaded HTML file can contain many independent <script> elements
- These get loaded in order into a single global scope
- Tools have emerged (Webpack) to take programs developed as separate modules and pack them into a single HTML with <script>s
  - Webpack is a module bundler for JavaScript
  - takes the dependencies and generates a dependency graph allowing web developers to use a modular approach

Modules in Node.js (CommonJS Style)

- One of the earlier methods of modularisation
- Largely superseded
- Each .js file becomes a module
- Each file can explicitly decide what to export by adding it to their module.exports array
- The consumer module can import all of the exports using the 'require' function
- This works in Node.js but does not work in the browser
  - Multiple scripts can be used, but they are all in global scope
  - There are tools (like browserify and webpack) that will transpile this into a single <script> for a browser
- An ECMAscript standard module interface was introduced in ECMAScript 2015

circle.js

```
const PI = 3.14159265359;

function area(radius) {
  return (radius ** 2) * PI;
}

function circumference(radius) {
  return 2 * radius * PI;
module.exports ={ PI: PI,
                  area : area,
                  circumference:
                  circumference};
```

main.js

```
let circle = require('./circle.js');

const r = 3;

console.log('Circle with radius %d has
  area: %d and circumference %d',
r, circle.area(r), circle.circumference(r));
```

ES Modules – introduced in ES6 (ES2015)

In Node.js – module files now have the extension .mjs

New keywords export and import

circle.mjs

```
const PI = 3.14159265359;

function area(radius) {
  return (radius ** 2) * PI;
}

function circumference(radius) {
  return 2 * radius * PI;
export { PI,area,circumference};
```

main.mjs

```
import  * as circle from ".\circle.mjs"

const r = 3;

console.log(`Circle with radius %d has
  area: %d and circumference %d',
r, circle.area(r), circle.circumference(r));
```

Package Managers - NPM

The proliferation of modules led to the development of an online repository and package managers to build and resolve dependency graphs – The most popular of these is the Node Package Manager (NPM)

- maintained by npm, Inc.
- the default package manager Node.js
- command line client
  - Called Npm
- online database of packages
  - npm registry
- Creates ‘package.json’ file describing the package (‘init’)
  - To Add modules to the package ‘install’
    - E.g. npm install express

```
{
  "name": "tester",
  "version": "1.0.0",
  "description": "great example",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Conor",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.1"
  }
}
```

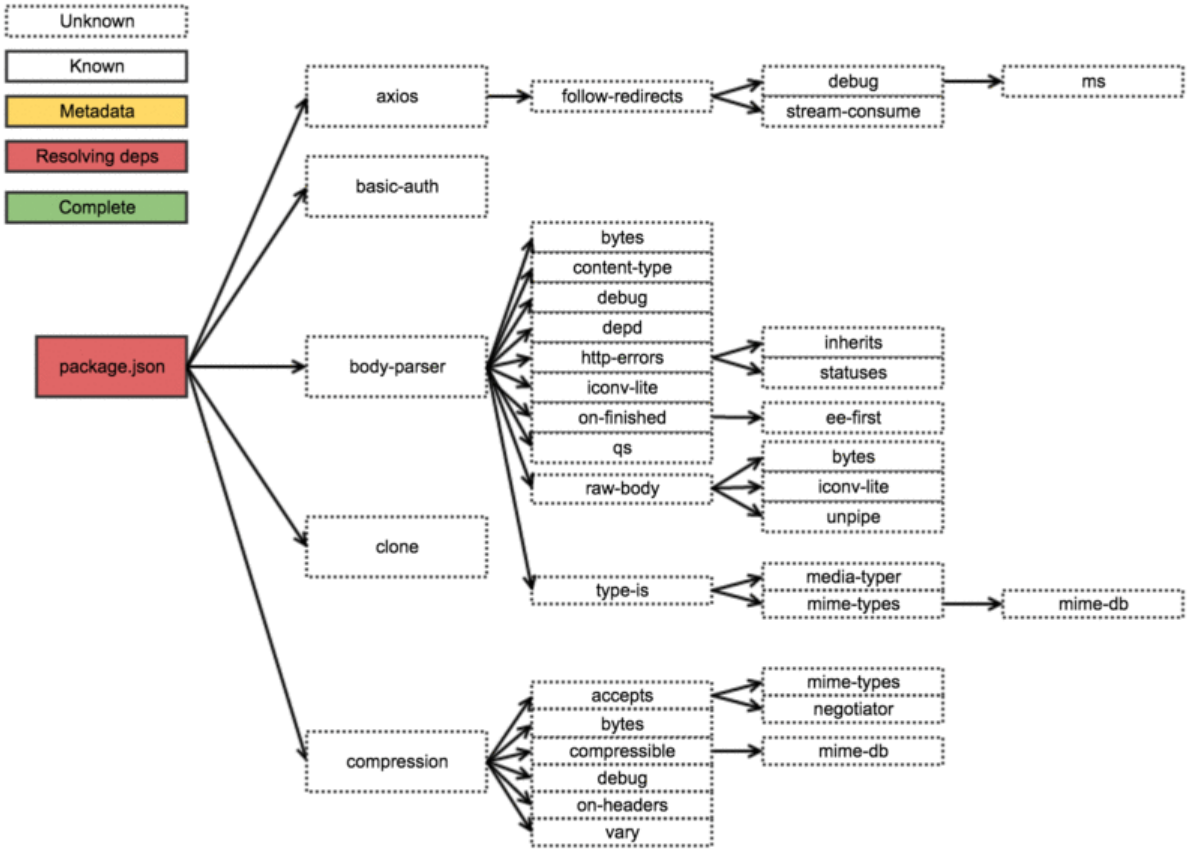
NPM

NPM installs the package and all of its dependencies

Can be version controlled

– exact version

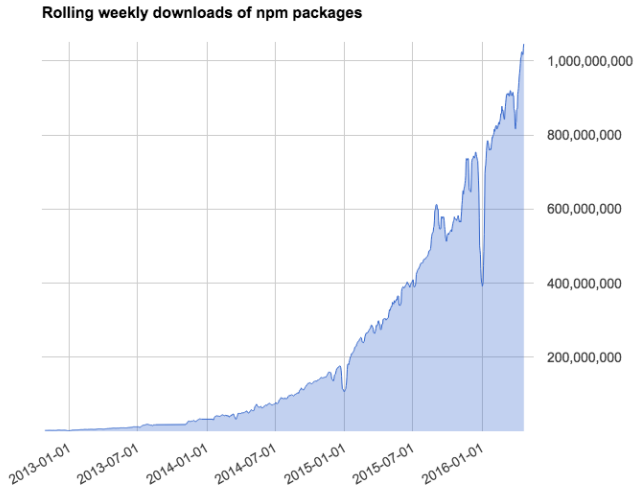
or >



NPM Public registry

Run by a commercial company called NPM Inc (since 2014), acquired by Github (Microsoft owned since 2018) in 2020

- Had > 1.3m packages by 2020! 75bn downloads per month
- Has become the primary evolution path for the JavaScript Ecosystem
- Modules are (mostly) unvetted
- Can lead to security vulnerabilities and Software Bloat



Callback pattern

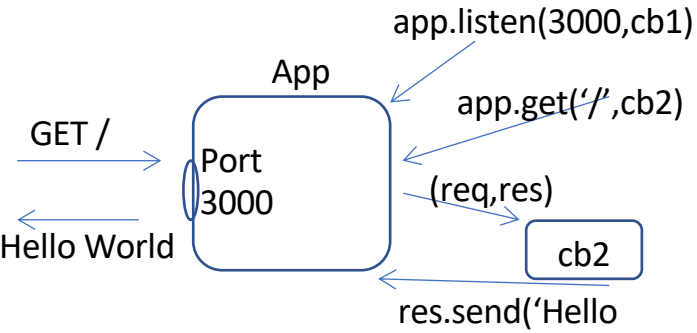
- JavaScript is single threaded
- Callbacks are used extensively for Asynchronous coding
  - or “doing something useful while a slow thing is underway”
- Dosomethingslow (function callmewhenyouarefinished)
- Do other useful things while the slow thing is happening
- Stops web pages ‘blocking’ when
- Long duration tasks are happening
  - E.g. code
  - setTimeout is passed a callback
  - Which is called when the time is finished

```
console.log("First Thing");
setTimeout( () => { console.log("Finished")}, 5000);
console.log("Second Thing");
```

```
First Thing
Second Thing
Finished      - 5 seconds later
```

Simple Express Example

Express is a very common framework for building web-based applications that uses callbacks extensively




```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => res.send('Hello World!'))

app.listen(port, () => console.log(`Example app listening on port ${port}!`))
```

- First install express using npm package manager
- Express is a framework, import it and call 'express()' to instantiate the framework
- starts a web server
- Which listens for requests coming in
- If request is get the url '/'
  - Then callback is invoked
  - req passed in
  - res is a function which is called when '/' is requested
- app.listen will tell express to listen at port 3000

Homework Exercise

Express is a simple way to create a webserver

`node index.js`

- Start the server,
  - use the code from the previous slide
  - You will have a lab on using node on Monday
- Use your browser to send requests to the server (running locally)
- http://localhost:3000/
  - 'Hello world'
    - Ensure that you get the expected response

Try another request, what response do you expect?

- http://localhost:3000/something
  - Cannot GET /something
  - No handler for that
- Add a new handler

Promises

- JavaScript has a lot of code that uses callbacks
- Promises are starting to replace callbacks

https://hackernoon.com/understanding-promises-in-javascript-13d99df067c1

- JavaScript runs a single-threaded event loop
- Current code runs until it blocks
  - then other events in the queue get scheduled
- Call-backs are one way to implement asynchrony
  - Promises build upon this

Example of Promise…then…catch

Taken from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

Doing sequential operations using callbacks results in an ugly pyramid of code

… 'Callback hell'

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('Got the final result: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

What is a Promise (new in ES6;EC2015) ?

Creating a Promise

let p = new Promise (<executor function>)

The executor function is a function that contains (usually blocking) actions that will get done eventually

A promise is either: *pending*, *fulfilled* or *rejected*

Executor Function:

- Take two arguments: both functions (resolve, reject)
- –The executor should get the job done
- –then call either resolve() or reject() to reflect the result in the promise
- these function can pass on data if required

When you call "new Promise".. The executor is called immediately.. When it blocks (or completes) control returns to "p = new Promise….."

Using the Promise

Generally you put something into an executor function which might take a while

There are often things you want to get done after the promise's work is finished (the promise is fulfilled or rejected) e.g. process data fetched from a website

You can queue these up in a chain of processing tasks using .then syntax

promise.then (onFulfilled, onRejected)

Up to 2 arguments: When the promise is resolved, then onFulfilled() or OnRejected() is called with the value of the promise

The .then call also returns a promise on which another .then can be invoked

Also

```
.catch (onRejected)    // essentially a synonym for .then(null, onRejected)
.finally (onSettled)   // calls onSettled regardless of whether the promise is fulfilled or rejected
```

Example of using the Promise…

- We create a new Promise and store it in p2
- The executor function is passed in when the Promise is created
- This example immediately resolves, value 1 (toy example)
- The value which was passed to resolve gets passed to onFulfilled (or onRejected)
- .then returns a promise
- Data can be passed through 'layers' of processing actions
- The sequence depends on detailed understanding of the event loop (next lecture)

```
var p2 = new Promise(function(resolve, reject) {
  resolve(1);
});

p2.then(function(value) {
  console.log(value); // 1
  return value + 1;
}).then(function(value) {
  console.log(value + ' - A synchronous value works');
});

p2.then(function(value) {
  console.log(value); // 1
});

///What does this print????
```

Quiz:
Answer 1
1
2 A synchronous value works
1

Quiz:
Answer 2
1
1
2 A synchronous value works

Quiz answer and why?

- We create two promises which execute when the first one resolves
- In general, we wouldn't be sure of the sequence if we didn't know how long each promise would take
- In this toy example the promise resolves 'immediately' so we can predict the output based on knowledge of how the even loop works

```
var p2 = new Promise(function(resolve, reject) {
  resolve(1);
});

p2.then(function(value) {
  console.log(value); // 1
  return value + 1;
}).then(function(value) {
  console.log(value + ' - A synchronous value works');
});

p2.then(function(value) {
  console.log(value); // 1
});

///What does this print????
```

Quiz:
Answer 2
1
1
2 A synchronous value works