Javascript V: Modules & Bundlers

Pau Fernández Rafa Genés Jose L. Muñoz

Universitat Politècnica de Catalunya (UPC)

Table of Contents

CommonJS Modules

Single file utilities module

To implement the `circle` module, write a file `circle.js` with:

```
// Define some utility functions
function area(r) {
  return Math.PI * r ** 2;
}

function circumference(r) {
  return 2 * Math.PI * r;
}

// Put them into the preexisting 'exports' object
exports.area = area;
exports.circumference = circumference;
```

How modules behave

Isolated Global Context

The global context in the module is invisible to the outside, so you can use any ”private” data and functions you need. You choose what to export.

Sequential execution

”Loading” the module means *sequentially executing the code in the module* (unlike C or Java). While loading a module, you can use the `console` or do any sort of computed initialization.

Exporting Functions and Variables

To make symbols visible to the loader, add them to the **exports** object (which is already existing and empty):

```
exports.someConstant = 42; // We just made this up
exports.area = area; // This function was defined earlier
exports.circumference = circumference; // So was this one
```

Exporting a single object

You can create the **exports** object yourself, assigning directly to `module.exports`,

```
module.exports = function () {
  console.log("Sorry, circle doesn't wanna work today");
}
```

(**module** is also preexisting and it is an object representing your module)

```
// Alternative way of doing the same as before
module.exports = {
  circle,
  circumference
};
```

Using CommonJS Modules

To use a module, load it with `require`:

```
const circle = require('./circle');
```

Every module returns an object, in which you will find the variables and functions as fields:

```
console.log(circle.area(1));
console.log(circle.circumference(1));
```

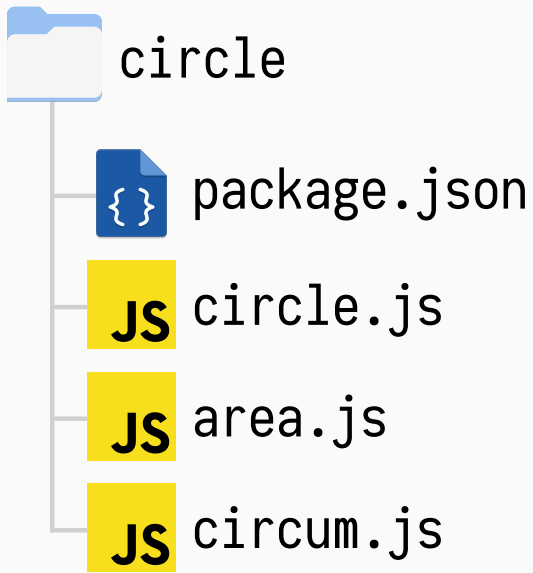
Destructuring works:

```
const { area, circumference } = require('./circle');
```

Directory Modules

To write a module in a directory:

- Create a **package.json** file.
- Use as many file modules as you want.
- Define an *entry point* (or "main module").



Directory Modules 2

area.js

```
function area(r) {
   return Math.PI * r**2;
}
exports.area = area;
```

circum.js

```
module.exports = function (r) {
  return 2 * Math.PI * r;
}
```

circle.js

```
exports.area = require('./area').area;
exports.circumference = require('./circum');
```

package.json

The `package.json` resides at the base directory of a module and describes its properties:

- `name`: Name of the module.
- `version`: Version number.
- `description`: Textual description of the module.
- `main`: module ID that is the primary entry point.
- `dependencies`: Object that maps package names to version ranges.
- `private`: If the module is not to be published.

More properties: `homepage`, `directories`, `keywords`, `repository`, `bugs`, `license`, `files`, `browser`, `bin`, ...

Details: https://docs.npmjs.com/files/package.json

Creating a `package.json`

To easily create a `package.json` for a new module:

```
npm init
```

This command will ask for:

- Package name
- Version
- Description
- Entry point (javascript file that will be loaded as the ”main” file)
- Test command
- Git repository
- Keywords
- Author
- License

Module Loading Rules

```
let mod = require('MODULE');
```

1) If `MODULE` is a core module, just load it.

2) If `MODULE` begins with `'./'` or `'../'`
   a) Load as file (`'MODULE.js'`).
   b) Load as directory:
      b1) Parse `MODULE/package.json`, look for `"main"` field.
      b2) Load `MODULE`/<the file specified as `"main"`>.

3) Load from `node_modules` (either in the local directory or from any parent directory)

Minimalistic **require** Implementation

This simplified implementation of **require** might throw some light about the process:

```
require.cache = Object.create(null);

function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name);
    let module = { exports: {} };
    require.cache[name] = module;
    let wrapper = Function("require, exports, module", code);
    wrapper(require, module.exports, module);
  }
  return require.cache[name].exports;
}
```

The Module Wrapper

Before execution, modules are wrapped in a function that looks like this:

```
(function(require, exports, module, __filename, __dirname) {
   // Module code actually lives in here
});
```

This has the following consequences:

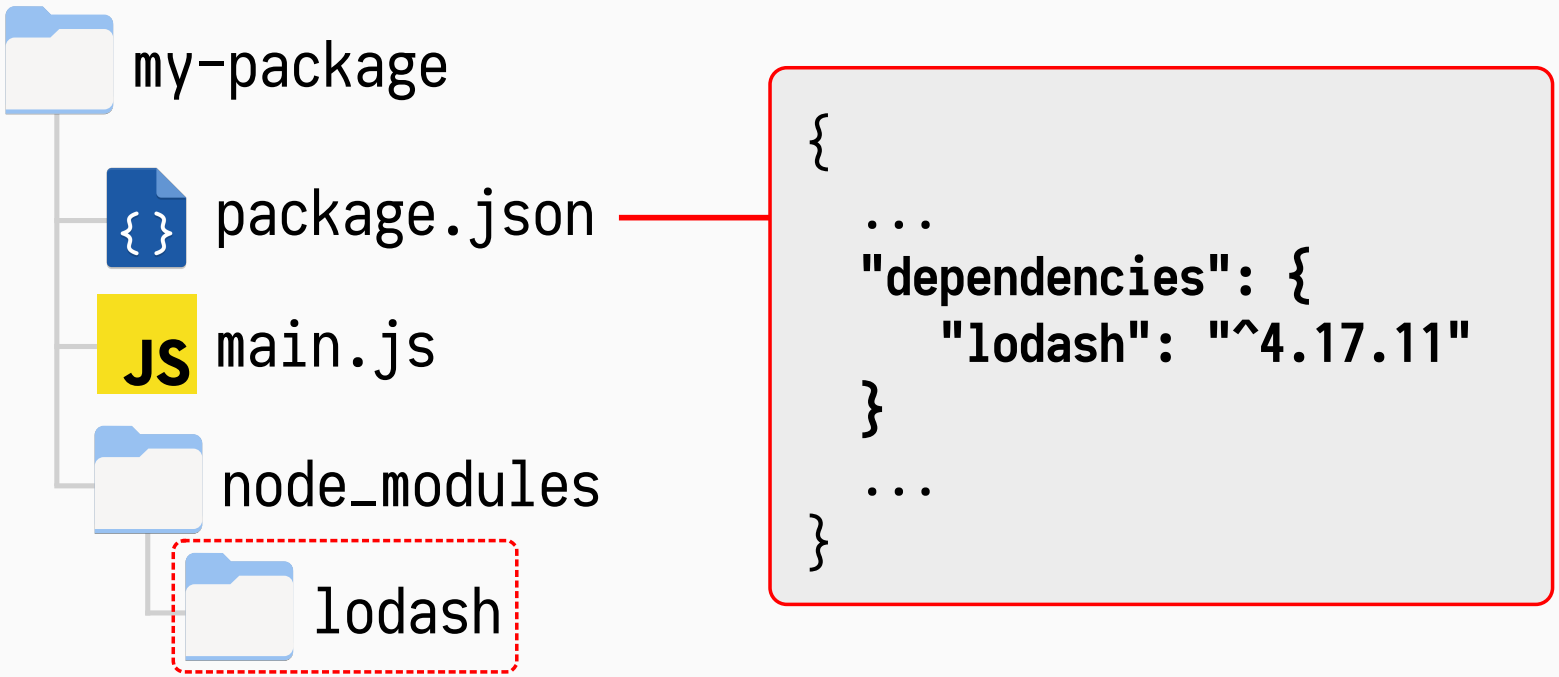
- Top-level variables are confined to the interior of the function and are thus local variables.
- It helps to provide some global-looking variables that are in fact specific to the module:
  - `module` and `exports` that the implementor can use to export values to the outside.
  - Convenience variables like `__filename`, `__dirname`.

Installing modules

Inside a Javascript package directory, installing a module is accomplished with:

```
npm install lodash
```

Two things happen: **a)** the module is installed into the **node_modules** local subdirectory and **b)** the dependency is registered in **package.json**:



package-lock.json

A package-lock.json describes a particular `node_modules` tree (and associated `package.json` file), for the following purposes:

- Make things exactly reproducible: the `package-lock.json` will ensure that the `node_modules` folder installed by npm in different places is exactly the same.
- Provide a way to "time-travel": save the state of previous `node_modules` tree so that it is not necessary to save the whole tree.
- Make changes to the `node_modules` tree observable in diffs.
- Optimize npm module installation by caching metadata resolution for already installed packages.

Details: https://docs.npmjs.com/files/package-lock.json

NodeJS Core Modules

NodeJS comes with core modules, implemented directly in the binary:

```
const nodejs_core_modules = {
   os:      require('os'),       // Operating System
   fs:      require('fs'),       // FileSystem (~POSIX)
   http:    require('http'),     // HTTP servers/clients
   https:   require('https'),    // HTTP over TLS/SSL
   net:     require('net'),      // TCP or IPC servers/clients
   events:  require('events'),   // API for Emitters and Listeners
   path:    require('path'),     // API for file and directory paths.
   cprocs:  require('child_processes'),
                                 // Spawn child child_processes
   // ...
}
```

Table of Contents

ES6 Modules

ES6 Modules

ES6: first time Javascript had a module system.

Named exports

We can have several **export**ed things per module:

```
// circle.js
export function area(r) {
  return Math.PI * r * r;
}
export function circumference(r) {
  return 2 * Math.PI * r;
}
export const tau = 2 * Math.PI;
```

```
// index.js
import { area, tau } from './circle';
```

Named exports

The names are important, we will use them when importing.

Write code as usual, then mark things you want to `export`.

Different `export`s can be `import`ed selectively.

Selective importing allows bundlers to do ”tree-shaking”.

Single **export** statement

Instead of marking everything with **export**, we can issue a single **export** statement at the end:

```
// circle.js
function area(r) {
  return Math.PI * r * r;
}
function circumference(r) {
  return 2 * Math.PI * r;
}
const tau = 2 * Math.PI;

// We choose here what to export
export { area, circumference, tau };
```

Import all named exports

To **import** all functions at once, we use * and name the object that will contain all the imported functions:

```
// circle.js
export const area = (r) => Math.PI * r * r;
export const circumference = (r) => 2 * Math.PI * r;
```

```
// index.js
import * as circFuncs from './circle';
console.log(circFuncs.area(4.5));
```

default export

One special thing can be marked as the **default** export

```
// my-component.js
export default class MyComponent {
  // ...
}
```

The default export does not need a name (it is an expression, actually).

The default export is **independent** of named exports.

Importing the default thing in a module has the simplest syntax:

```
// index.js
import MyComponent from './my-component';
```

Renaming when importing

```
// circle.js
export const circleArea = (r) => Math.PI * r * r;
export const circleCircum = (r) => 2 * Math.PI * r;
```

Named imports can be renamed when importing

```
// index.js
import {
  circleArea as circ,
  circleCircum as circumference
} from './circle';
```

Renaming when exporting

Or renamed when exporting

```
// circle.js
const circleArea = (r) => Math.PI * r * r;
const circleCirc = (r) => 2 * Math.PI * r;

export {
  circleArea as area,
  circleCirc as circumference
};
```

```
// index.js
import { area, circumference } from './circle';
```

Re-exporting

To write modules in different files, you can directly export imported things:

```
// Export all named symbols from "alpha"
export * from 'alpha';
```

```
// Export specific things from "beta"
export { a, b } from 'beta';
```

```
// Export specific renamed things from "beta"
export { a as aaa, b as bbb } from 'beta';
```

```
// Export the default from "gamma"
export { default } from 'gamma';
```

```
// Export a named thing as the default from "delta"
export { c as default } from 'delta';
```

ES6 Modules in browsers

```
// util.js
const removeAllChildren = (elem) => {
  while (elem.firstChild) {
    elem.firstChild.remove();
  }
}
```

The `type="module"` attribute tells the browser a script is an ES6 module.

```
<!-- index.html -->
  <script type="module">
    import { removeAllChildren } from './util.js';
    removeAllChildren(document.body);
  </script>
<!-- ... -->
```

Importing from other modules loads them relative to the current URL.

Table of Contents

Bundlers

Bundlers

Making a web application requires many files, connected in specific ways.

Types of files: HTML, CSS, Javascript, Image (pngs, jpg, svg, ...), icons, fonts, etc.

We want to process these files: transpile (ES6 to ES5, PostCSS), minify, tree-shake, split, chunk.

Bundlers automate this process. They read dependencies from `import` statements, create a **dependency Graph** and produce an output bundle (a single or multiple files), which are a transformation of our original files.

Webpack

The most popular bundler nowadays is **webpack**.

Webpack start

Create a directory and `package.json` inside:

```
npm init -y
```

Install `webpack` as a library and CLI:

```
npm install -D webpack webpack-cli
```

Create `index.js` and `dist/index.html`

Add scripts to `package.json`

```
{
  "scripts": {
    "dev": "webpack --mode development",
    "build": "webpack --mode production"
  }
}
```

Execute `webpack`

```
npm run build
```

Webpack configuration

If the file **webpack.config.js** exists, it is executed and exports a configuration object:

```
// webpack.config.js
const path = require('path');

module.exports = {
  // Entry point
  entry: './src/index.js',
  output: {
    // Generated bundle name
    filename: 'bundle.js',
    // Output directory
    path: path.resolve(__dirname, 'public')
  }
};
```