Notes - JavaScript - CommonJS Modules

- Dr Nick Hayward

A collection of notes &c. on plain JavaScript modules, in particular usage of CommonJS modules and Node.js.

Contents

Intro CommonJS is a module library and pattern popularised by Node.js. It is still in regular use for modern apps, both Node.js and client-side based apps. For client-side, we may use build tools, such as *WebPack* and *Browserify*, to bundle module dependencies for online usage in a browser-based environment.

General usage In CommonJS, every file is a module.

Each module has an implicit local scope, and the *global* scope needs to be accessed explicitly. The module code is wrapped in a function by the loading call, which provides this local scope.

CommonJS modules may export a public interface for consumption in apps - dependencies are resolved using `require` function calls.

`require` function calls are synchronous, returning the exposed interface for the required module.

By default, functions in a module will be local to that module. We may then choose the functions, properties &c. to export in the public API for the module.

e.g.

```
// CONSTRUCTOR - create basic keyboard object for testing...
function Keyboard(keyDetails) {
  ({
        total: this.total,
    x: this.x,
    y: this.y,
    white: this.white,
    black: this.black
     } = keyDetails
  );
}

module.exports = Keyboard;
```

The function `Keyboard` is now available to other files (modules) by requiring the file,

```
const keyboard = require('./basic1');
```

Local, relative modules may use the standard Unix path to define file access for the module relative to the application, e.g.

- `/` , `./` , `../`

If this path identifier is not included, a module is assumed to be either a core module or located in the app's local `node_modules` directory.

n.b. filename ending may be omitted if `.js` , `.json` , or `.node` . These are implicitly checked first as part of the module loading. Common endings include `.js` and `.json` .

We can then access the exported function &c. as follows,

```
const myKeyboard = new keyboard(keyDetails);
console.log(`exported key total = ${myKeyboard.total}`);
```

Modules Modules provide a wide range of features and functionality. These include built-in functionality, such as file system, and additional features provided by the Node.js community.

Modules may be installed using NPM from Node's package repository,

- NPM

We may also develop custom modules for local usage and optional publication to NPM.

Folders as modules In addition to single files, we may also use folders in an application to organise related modules or custom libraries. We may then access this library directory in a number of ways. e.g.

- package.json
- index.js (index.node) file

`package.json` We may start by creating a custom module, which will use a `package.json` file for metadata. We may either create this from scratch or use the command `npm init` .

This file includes an option to specify a *main* or start file for that directory, and module. This is usually specified as `index.js` .

e.g.

```
{
  "name" : "spire and the signpost",
  "main" : "./lib/spire.js"
}
```

This `package.json` file should be placed at the root of the library folder. If we then require this library in our app,

```
// require directory for custom library - package.json checked for main file location...
const spire = require('./spire');
```

Node.js will try to load the main file for this library from the `./spire/lib/spire.js` file. We may then call library methods using the standard pattern.

`index.js` We may also use this `index.js` file at the root of a directory without a `package.json` file. This also allows us to reference or `require` a whole directory. To use a directory, we `require` it and then add an `index.js` file for this directory.

We may then create custom modules in the library, require them in the `index.js` file, and export for use in the `app.js` file.

e.g.

- log.js

```
// basic logger
function log(value, ...values) {
  const logValue = console.log(value, ...values);

  return logValue;
}

// exports
module.exports = {
  log
};
```

This may be required in the `index.js` file, and exported for use in the `app.js` file.

```
const { log } = require('./log');

module.exports = {
  log
}
```

We then call this exported method as follows in the `app.js` file,

```
// require directory for custom library - Node.js checks index.js in ./lib
const lib = require('./lib');

// call method from library - defined in module file & exported from index.js
lib.log('greetings from planet earth...');
```

`require` pattern We may export functions &c. from a given module, and then use the following pattern to simplify `require` usage,

```
const listView = require('./views/item').listView;
const renderHTML = require('./views/render').renderHTML;
```

Dynamic require for modules `require` may be used dynamically, as with other JS functions.

This is commonly used for dynamically requiring different modules to conform to a particular *interface* in an app.

e.g. we might create a templates directory with different view template functions. Each template will take a model and return a HTML string.

1. item.js module

```
module.exports = model => `
<ul>
  <li>${model.title}</li>
  <li>${model.author}</li>
  <li>${model.year}</li>
</ul>
`
```

2. render.js module

A module for abstraction and dynamic modules,

```
function render(template, model) {
  return require(`./views/${ template }.js`)(model);
}

module.exports = render;
```

3. app.js

We may then consume these modules in `app.js` , e.g.

```
// require render.js file
const genericRender = require('./render');

const html = genericRender('item', {
  title: 'Perfume from Provence',
  author: 'Lady Winifred Fortescue',
  year: 1935,
})

console.log(html);
```

Module design *Bundlers* are useful tools for combining multiple files, usually one for each module, into a large file. For client-side performance, it tends to be faster to load a single, large file compared to multiple smaller files. This, of course, does not include various asynchronous options, but instead standard page load.

We may also use *minifier* tools to help condense and prepare production code. Minifiers make code smaller by automatically removing comments and whitespace, renaming bindings, and replacing pieces of code with equivalent code that takes up less space

Node.js - Core modules Node.js includes several core modules, which may be installed as part of a working app.

These core modules also gain preference in loading, including namespace resolution. For example, if we `require` a `http` module, Node's core *HTTP* module will load. This module will, effectively, overwrite and take precedence over local modules with the name `http` .