sitepoint.com

Understanding module.exports and exports in Node.js - SitePoint

James Hibbard

7-9 minutos

In programming, modules are self-contained units of functionality that can be shared and reused across projects. They make our lives as developers easier, as we can use them to augment our applications with functionality that we haven't had to write ourselves. They also allow us to organize and decouple our code, leading to applications that are easier to understand, debug and maintain.

In this article, I'll examine how to work with modules in Node.js, focusing on how to export and consume them.

Different Module Formats

As JavaScript originally had no concept of modules, a

variety of competing formats have emerged over time. Here's a list of the main ones to be aware of:

- The Asynchronous Module Definition (AMD) format is used in browsers and uses a `define` function to define modules.
- The CommonJS (CJS) format is used in Node.js and uses `require` and `module.exports` to define dependencies and modules. The npm ecosystem is built upon this format.
- The ES Module (ESM) format. As of ES6 (ES2015), JavaScript supports a native module format. It uses an `export` keyword to export a module's public API and an `import` keyword to import it.
- The System.register format was designed to support ES6 modules within ES5.
- The Universal Module Definition (UMD) format can be used both in the browser and in Node.js. It's useful when a module needs to be imported by a number of different module loaders.

  Please be aware that this article deals solely with the **CommonJS format**, the standard in Node.js. If you'd like to read into any of the other formats, I recommend this article, by SitePoint author Jurgen Van de Moere.

Requiring a Module

Node.js comes with a set of built-in modules that we can use in our code without having to install them. To do this, we need to require the module using the `require` keyword and assign the result to a variable. This can then be used to invoke any methods the module exposes.

For example, to list out the contents of a directory, you can use the file system module and its `readdir` method:

```
const fs = require('fs');
const folderPath = '/home/jim/Desktop
/';

fs.readdir(folderPath, (err, files) =>
{
  files.forEach(file => {
    console.log(file);
  });
});
```

Note that in CommonJS, modules are loaded synchronously and processed in the order they occur.

Creating and Exporting a Module

Now let's look at how to create our own module and export it for use elsewhere in our program. Start off by creating a `user.js` file and adding the following:

```
const getName = () => {
  return 'Jim';
};

exports.getName = getName;
```

Now create an `index.js` file in the same folder and add this:

```
const user = require('./user');
console.log(`User: ${user.getName()}`);
```

Run the program using `node index.js` and you should see the following output to the terminal:

```
User: Jim
```

So what has gone on here? Well, if you look at the `user.js` file, you'll notice that we're defining a `getName` function, then using the `exports` keyword to make it available for import elsewhere. Then in the `index.js` file, we're importing this function and executing it. Also notice that in the `require` statement, the module name is prefixed with `./`, as it's a local file. Also note that there's no need to add the file extension.

Exporting Multiple Methods and Values

We can export multiple methods and values in the same way:

```
const getName = () => {
  return 'Jim';
};

const getLocation = () => {
  return 'Munich';
};

const dateOfBirth = '12.01.1982';

exports.getName = getName;
exports.getLocation = getLocation;
exports.dob = dateOfBirth;
```

And in `index.js`:

```
const user = require('./user');
console.log(
  `${user.getName()} lives in
${user.getLocation()} and was born on
${user.dob}.`
);
```

The code above produces this:

```
Jim lives in Munich and was born on
12.01.1982.
```

Notice how the name we give the exported `dateOfBirth` variable can be anything we fancy (`dob` in this case). It doesn't have to be the same as the original variable name.

Variations in Syntax

I should also mention that it's possible to export methods and values as you go, not just at the end of the file.

For example:

```
exports.getName = () => {
  return 'Jim';
};

exports.getLocation = () => {
  return 'Munich';
};

exports.dob = '12.01.1982';
```

And thanks to destructuring assignment, we can cherry-pick what we want to import:

```
const { getName, dob } =
require('./user');
```

```
console.log(
  `${getName()} was born on ${dob}.`
);
```

As you might expect, this logs:

```
Jim was born on 12.01.1982.
```

Exporting a Default Value

In the above example, we're exporting functions and values individually. This is handy for helper functions that could be needed all over an app, but when you have a module that exports just the one thing, it's more common to use `module.exports`:

```
class User {
  constructor(name, age, email) {
    this.name = name;
    this.age = age;
    this.email = email;
  }

  getUserStats() {
    return `
      Name: ${this.name}
      Age: ${this.age}
      Email: ${this.email}
    `;
```

```
  }
}

module.exports = User;
```

And in `index.js`:

```
const User = require('./user');
const jim = new User('Jim', 37,
'jim@example.com');

console.log(jim.getUserStats());
```

The code above logs this:

```
Name: Jim
Age: 37
Email: jim@example.com
```

What's the Difference Between `module.exports` and `exports`?

In your travels across the Web, you might come across the following syntax:

```
module.exports = {
  getName: () => {
    return 'Jim';
  },
```

```
  getLocation: () => {
    return 'Munich';
  },

  dob: '12.01.1982',
};
```

Here we're assigning the functions and values we want to export to an `exports` property on `module` — and of course, this works just fine:

```
const { getName, dob } =
require('./user');
console.log(
  `${getName()} was born on ${dob}.`
);
```

This logs the following:

```
Jim was born on 12.01.1982.
```

So what *is* the difference between `module.exports` and `exports`? Is one just a handy alias for the other?

Well, kinda, but not quite …

To illustrate what I mean, let's change the code in `index.js` to log the value of `module`:

```
console.log(module);
```

This produces:

```
Module {
  id: '.',
  exports: {},
  parent: null,
  filename: '/home/jim/Desktop
/index.js',
  loaded: false,
  children: [],
  paths:
   [ '/home/jim/Desktop/node_modules',
     '/home/jim/node_modules',
     '/home/node_modules',
     '/node_modules' ] }
```

As you can see, `module` has an `exports` property. Let's add something to it:

```
// index.js
exports.foo = 'foo';
console.log(module);
```

This outputs:

```
Module {
  id: '.',
  exports: { foo: 'foo' },
  ...
```

Assigning properties to `exports` also adds them to `module.exports`. This is because (initially, at least)

`exports` is a reference to `module.exports`.

So Which One Should I use?

As `module.exports` and `exports` both point to the same object, it doesn't normally matter which you use. For example:

```
exports.foo = 'foo';
module.exports.bar = 'bar';
```

This code would result in the module's exported object being `{ foo: 'foo', bar: 'bar' }`.

However, there is a caveat. Whatever you assign `module.exports` to is what's exported from your module.

So, take the following:

```
exports.foo = 'foo';
module.exports = () => {
console.log('bar'); };
```

This would only result in an anonymous function being exported. The `foo` variable would be ignored.

If you'd like to read more into the difference, I recommend this article.

Conclusion

Modules have become an integral part of the JavaScript ecosystem, allowing us to compose large programs out of smaller parts. I hope this article has given you a good introduction to working with them in Node.js, as well as helping to demystify their syntax.

If you have any questions or comments, please feel free to hop on over to the SitePoint forums to start a discussion.