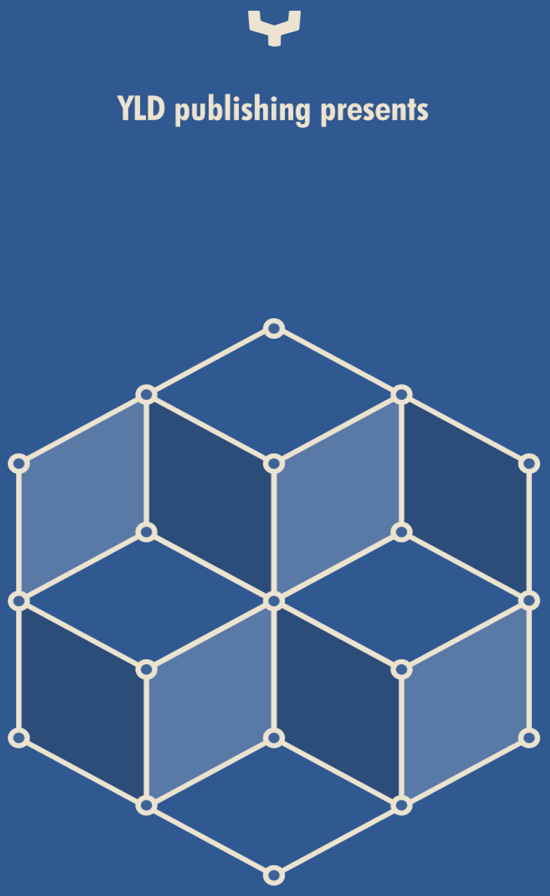

YLD publishing presents

MODULE PATTERNS

UNDERSTANDING AND USING THE NODE MODULE SYSTEM

Pedro Teixeira

Module Patterns

Understanding and using the Node module system

Pedro Teixeira

This book is for sale at http://leanpub.com/modulepatterns

This version was published on 2015-02-25

Contents

1. Foreword

Since its first introduction back in 2009, Node.js has been widely adopted, not only by cutting-edge startups and programmers, but also by big companies like Microsoft, PayPal, Walmart and many others.

Node.js provides the minimum viable foundation to support the main Internet protocols. Given its low-level API, any reasonably complex application needs to make extensive use of libraries.

Node.js is very indifferent in the way that you organise your code. This means that any team is free to use any set of libraries and code conventions, which makes it difficult to onboard new programmers onto projects and communicate across teams.

Not only does the Node and NPM ecosystem enable you to easily assemble and create a traditional HTTP Server, but it also allows you to easily create and invent new networked service providers and consumers, allowing programmers to use JavaScript to build Peer-to-Peer, Internet of Things, Real-time communications, gossip networks, and many other types of system.

This series of books aims to extract and explain some of the common patterns used throughout several types of applications, and introduce some new less common ones. It covers small-scale code patterns, best practices, and also large-scale networking patterns, allowing you to collect and develop a portfolio of concepts, libraries and code that you can use to communicate, and to make the development of new systems easier. I hope you find these books useful!

2. The source code

You can find the source code for the entire series of books in the following URL:

https://github.com/pgte/node-patterns-code[1]

You can browse the code for this particular book here:

https://github.com/pgte/node-patterns-code/tree/master/01-module-patterns[2]

You are free to download it, try it and tinker with it. If you have any issue with the code, please submit an issue on Github. Also, pull requests with fixes or recommendations are welcome!

[1] https://github.com/pgte/node-patterns-code
[2] https://github.com/pgte/node-patterns-code/tree/master/01-module-patterns

3. Introduction

JavaScript started life without modules being part of the language, being designed for small browser scripts. Historically the lack of modularity led to shared global variables being used to integrate different pieces of code.

```
var a = 1;
var b = 2;
```

For instance, this last piece of JavaScript code, if run in a browser, assigns two variables into the global scope. Another piece of unrelated code, when running on the same browser page, can get hold of the values contained in this variable. If two pieces of code running on the same page rely unintentionally on the same global variable, awkward behaviour is sure to occur.

To avoid creating global variables, you can use a function to create a new scope:

```
function() {
  var a = 1;
  var b = 2;
}
```

But this last piece of code only declares an anonymous function, which does not get executed. You can invoke this function immediately after declaring it via an "Immediately Invoked Function Expression":

```
(function() {
  var a = 1;
  var b = 2;

  console.log('inside scope: a: ' + a + ', b:' + b);
})();
console.log('outside scope: a: ' + a + ', b:' + b);
```

If you execute this last piece of code inside a browser you will get something like the following output:

```
inside scope: a: 1, b:2
ReferenceError: a is not defined
```

This is just what we wanted: our variables are declared in a non-global scope, and that scope does not leak into the global one.

4. Returning the exported value

One way of creating an isolated module in the browser is to extend this technique of wrapping the module code inside a function by returning the interface that that module should export.

For instance, if we wanted to create a currency module that rounds a currency value to cents, we could do this:

```
var currency =
(function() {
  return {
    round: function(amount) {
      return Math.round(amount * 100) / 100;
    }
  };
}());

currency.round(1);     // 1
currency.round(1.2);   // 1.2
currency.round(1.24);  // 1.24
currency.round(1.246); // 1.25
```

We could now enrich our `currency` module by adding other functions to the object that our anonymous function wrapper returns.

The anonymous function wrapper also gives us privacy – data and behaviour that supports the public API, without being visible or modifiable by the code that uses our module. For instance, we could keep some kind of (very naive) cache of which the calling code is completely ignorant:

```
var currency =
(function() {

  var cache = {};

  return {
    round: function(amount) {
      var rounded = cache[amount];
      if (! rounded) {
        rounded = cache[amount] = Math.round(amount * 100) / 100;
```

```
      }
      return rounded;
    }
  };
}());

currency.round(1.246); // 1.25
currency.round(1.246); // 1.25  (cached value)
```

5. Modules in the browser

Using this technique we can build a very simple module system where we keep each module in a separate file, concatenate them in a specific order, and then use the global namespace to resolve each of the modules.

For instance, if you had two modules that your application depended on:

cache.js:

```
var Cache =
(function() {
  return function(max) {
    var keys = [];
    var cache = {};

    return {
      get: function(key) {
        return cache[key];
      },
      set: function(key, value) {
        keys.push(key);
        if (keys.length > max) {
          var oldestKey = keys.shift();
          delete cache[oldestKey];
        }
        cache[key] = value;
      }
    };
  }
}());
```

currency.js:

```
var currency =
(function() {

  var cache = Cache(100);

  return {
    round: function(amount) {
      var rounded = cache.get(amount);
      if (! rounded) {
        rounded = Math.round(amount * 100) / 100;
        cache.set(amount, rounded);
      }
      return rounded;
    }
  };
}());
```

app.js:

```
[12, 12.34, 12.345].forEach(function(val) {
  var rounded = currency.round(val);
  console.log('rounded ' + val + ' is ' + rounded);
});
```

The first file, `cache.js` implements a cache and is meant to be used as a constructor for cache objects.

The second file, `currency.js` implements our currency module. (For the sake of brevity this module only contains a `round` function.) This module depends on the previous `cache` module already being defined at the global level.

Now, within a directory containing these three files, you can concatenate them on the shell using a command-line utility like `cat`:

```
$ cat cache.js currency.js app.js > all.js
```

6. A Module registry

This last approach has at least a couple of problems:

Firstly, the dependencies are not explicit: it's not clear which modules any one module depends on.

Secondly, creating a module implies creating all of this boilerplate code for wrapping the module inside a function, invoking that function, and capturing the return value into a global variable.

Thirdly, the modules must be loaded in the correct order of dependency.

To solve this we can create a module registry system that provides two functions: `define` and `require`.

6.0.1 Define and require

The first one, `define`, is to be used to define a module like this:

```
modules.define('module name', function() {
  // module code
  // exports = ...
  return exports;
});
```

Let's implement it then:

```
var modules =
(function() {

  var modules = {};

  function define(name, fn) {
    modules[name] = fn();
  }

  return {
    define: define
  };
}());
```

We have a way of creating modules using `modules.define()` – let's now create the way to consume them using `modules.require()`:

```
var modules =
(function() {

  var modules = {};

  function define(name, fn) {
    modules[name] = fn();
  }

  function require(name) {
    if (!modules.hasOwnProperty(name)) {
      throw new Error('Module ' + name + ' not found');
    }
    return modules[name];
  }

  return {
    define: define,
    require: require
  };
}());
```

Let's then save this into a file named `modules.js`.

6.0.2 Using the module system

Now let's redefine our modules using this new system:

cache.js:

```
modules.define('cache', function() {
  return function(max) {
    var keys = [];
    var cache = {};

    return {
      get: function(key) {
        return cache[key];
      },
      set: function(key, value) {
        keys.push(key);
        if (keys.length > max) {
```

```
          var oldestKey = keys.shift();
          delete cache[oldestKey];
        }
        cache[key] = value;
      }
    };
  }
});
```

currency.js:

```
modules.define('currency', function() {
  var cache = modules.require('cache')(100);

  return {
    round: function(amount) {
      var rounded = cache.get(amount);
      if (! rounded) {
        rounded = Math.round(amount * 100) / 100;
        cache.set(amount, rounded);
      }
      return rounded;
    }
  };
});
```

app.js:

```
var currency = modules.require('currency');

[12, 12.34, 12.345].forEach(function(val) {
  var rounded = currency.round(val);
  console.log('rounded ' + val + ' is ' + rounded);
});
```

Now you need to concatenate these files using the command line, starting with the `modules.js` one:

```
$ cat modules.js cache.js currency.js app.js > all.js
```

If you're running a Windows system instead of a Unix-based one, you can use `type` instead of `cat`:

```
$ type modules.js cache.js currency.js app.js > all.js
```

To execute it, you can create this simple HTML page:

test.html:

```
<script src="all.js"></script>
```

Now, load it on a browser. The browser console should show:

```
rounded 12 is 12
rounded 12.34 is 12.34
rounded 12.345 is 12.35
```

7. Transitive module loading

What happens to this module system once we get a circular dependency? Direct circular dependencies are hard to find: a module A that depends on module B, which depends on module A, is a rare thing (and probably a sign that something is wrong with your module design). A less rare occurrence is an indirect circular dependency: module A which depends on module B, which depends on module C, which depends on module A. Let's see how our module system reacts when something like this happens:

a.js:

```
modules.define('a', function() {
  modules.require('b');
});
```

b.js:

```
modules.define('b', function() {
  modules.require('c');
});
```

c.js:

```
modules.define('c', function() {
  modules.require('a');
});
```

app.js:

```
modules.require('a');
```

Let's then concatenate these files into one:

```
$ cat modules.js a.js b.js c.js app.js > all.js
```

Again, if you're running Windows, use `type` instead.

Load these modules into a browser by loading the `all.js` file in the browser:

test.html:

```
<script src="all.js"></script>
```

When executing this you will see that there is an infinite recursion where the module system is desperately trying to resolve each module in a never-ending loop.

To fix this we could simply prevent recursive loading in the module system by keeping track of which modules are currently being loaded, but this still doesn't allow us to get modules that contain circular dependencies, as a consensus on the return value will never be reached. Instead, we'll get an error.

The problem here is that, by using return values to define the modules exports, we're falling into infinite recursion. To allow circular dependencies we need to change the way by which modules export their public interfaces. Instead of returning the exposed value, we need the module to export its interface by modifying an object, allowing it then to have temporary module attributes and values before returning from initialisation. Here is an example of such a module system implementation:

modules.js

```
var modules =
(function() {

  var modules = {};

  function define(name, fn) {
    if (modules[name])
      throw Error('A module named ' + name + ' is already defined');

    var module = {
      exports: {},
      fn: fn,
      executed: false
    };

    modules[name] = module;
  }

  function require(name) {
    var module = modules[name];
    if (! module)
      throw new Error('Module ' + name + ' not found');

    if (! module.executed) {
      module.executed = true;
      module.fn.call(module, module, module.exports);
```

```
    }

    return module.exports;
  }

  return {
    define: define,
    require: require
  };
}());
```

Here we're lazily evaluating modules the first time they're required. Also, for each module we keep around the exported value in `module.exports` and we pass the `module.exports` and the `module` objects as arguments to the module function:

```
      module.fn.call(module, module, module.exports);
```

The first argument allows the module itself to change the exported value at will. The last one allows the module to change the exported value completely by reassigning `module.exports`, making it possible for a module to export things other that objects: functions or single scalar values, strings, etc.

Let's see how such a system would behave when you have a circular dependency: module A which depends on module B, which depends on module C, which depends on module A:

a.js:

```
modules.define('a', function(module, exports) {
  exports.array = ['a'];
  modules.require('b');
});
```

Here the module A simply exports an attribute named `array` which contains an array, initially populated with the string `"a"`.

b.js:

```
modules.define('b', function(module) {
  var c = modules.require('c');
  c.push('b');
  module.exports = c;
});
```

c.js:

```
modules.define('c', function(module) {
  var a = modules.require('a');
  a.array.push('c');
  module.exports = a.array;
});
```

Here module `c` requires module `a`, pushes the string `c` into `a.array`, and then assigns that array to `module.exports`, effectively defining that array as the exported value. This allows module `b`, which depends on module `c`, to use that value to push the string `b` to it.

Let's then define an "app" that depends on module A, and concatenate them into a bundle like we did before:

app.js:

```
var a = modules.require('a');
console.log('a.array:', a.array);
```

```
cat modules.js a.js b.js c.js app.js > all.js
```

When executing this bundle, we get the following console log:

```
a.array: [ 'a', 'c', 'b' ]
```

The order may not be what you were expecting, but if you come to think of it, module C is the first to get a hold of module A, adding to its array. Then comes module B, which finally gets a hold of module C, and adds to the array.

8. The Node.js module system

Node.js covers the lack of modules in JavaScript by implementing its own module system. This system is based on a standard older than Node.js itself, called CommonJS. Let's see how the Node.js module system works:

8.1 Defining a module

Generally speaking, a JavaScript file is a module. You define a module by having a file containing any JavaScript code:

```
var a = 1;
var b = 2;
```

This module defines two variables and, unlike a bare browser environment, these variables do not leak to any outer scope. Instead, if you want to export a value, you define attributes on the `export` variable like this:

```
exports.round = function(amount) {
  return Math.round(amount * 100) / 100;
};
```

Here we're changing the exported object to add a `round` attribute, which is a function.

But what if you want to export a single value instead of an object with multiple attributes? You could, for instance, want to export only a single function. (More about these types of module patterns later.) To do that you could change the `module.exports` value. The `module` variable represents the current module, and the value that the `module.exports` attribute has is what gets returned to the users of this module.

```
/// exporting a single value
module.exports = function round(amount) {
  return Math.round(amount * 100) / 100;
};
```

Both the `exports` and the `module` objects are implicit module variables that the Node module system makes available.

8.2 Using a module

To access a module you use the `require` function, passing in the absolute or relative path of the module file. For instance, if you're looking to import a module named `cache` and the module file is sitting on the same directory as the current file, you can import it by doing:

```
var cache = require('./cache.js');
```

The `require` function finds the module, parses and executes it, and then returns the module exported value, which we then assign to a local variable. The name of the variable that holds the module exported value is indifferent, and in this case, it's just a coincidence that it shares a similar name with the module that is being imported.

For brevity you can omit the `.js` extension:

```
var cache = require('./cache');
```

If the required module sits in a different directory from the current file that's requiring it, you can use a relative path like this:

```
var cache = require('../lib/cache');
var users = require('./models/users');
```

9. Module types

The Node.js module system then allows for a module to export any value, be it a string, a number, a single function, or a more complex object. Let's see some common patterns that can emerge:

9.1 One function

The quintessential module is one that exports only one function, and this function performs one action. As an example, a module that provides a logging function could be defined like this:

```
var fs = require('fs');
var file = fs.createWriteStream('/tmp/log.txt');

module.exports = function log(what) {
  var date = new Date();
  file.write(JSON.stringify({when: date.toJSON(), what: what}) + '\n');
};
```

If you strictly adhere to the UNIX and functional-programming mantra "do one thing and do it well", this is the only module pattern you may need. But since life may never be as simple as we might wish, let's now look at others.

9.2 Singleton object

When modelling your system, you may be tempted to create classes for every object type you imagine, but on some occasions you may only need one instance of one object in each Node.js process. In this case, you don't need to create a class – you only need to use the module system to encapsulate that object state.

For instance, if you were to define one logging object that exposes three functions, one for each logging level:

singleton_logger.js:

```
exports = module.exports = log;

var fs = require('fs');
var file = fs.createWriteStream('/tmp/log.txt');

const DEFAULT_LEVEL = 'info';

function log(what, level) {
  var entry = {
    when: new Date,
    level: level || DEFAULT_LEVEL,
    what: what
  };

  file.write(JSON.stringify(entry) + '\n');
};

exports.info = function(what) {
  log(what, 'info');
};

exports.warn = function(what) {
  log(what, 'warning');
};

exports.critical = function(what) {
  log(what, 'critical');
};
```

As you can see, this module not only exports the `info`, `warn` and `critical` functions, it only allows its base value to be used as a function:

singleton_client.js:

```
var log = require('./singleton_logger');

log('one');
log.info('two');
log.warn('three');
log.critical('four!');
```

9.3 Examples

9.3.1 Configuration

One common use of the Singleton pattern is to concentrate configuration. Typically, at the root of each repo lies a `config` folder containing all the configuration files:

```
config/
├── index.js
├── mail.json
├── couchdb.json
└── redis.json
```

The `index.js` would then aggregate each of the configuration files:

config/index.js:

```
exports.couchdb = require('./couchdb');
exports.redis = require('./redis');
```

The modules that need some configuration values only need to require the config dir like this:

```
var config = require('../config');
var mailer = Mailer(config.mail);
```

(Later we discuss this index.js aggregating pattern in mode detail.)

9.3.1.1 Connection-sharing

If your application depends on external services like a database server or an HTTP server, it's easier to encapsulate access in one Singleton module for easy access.

For instance, if your application depends on one Redis server, this pattern is common:

redis.js:

```
var Redis = require('redis');
var config = require('./config').redis;

// initialize the redis client connection
var redisClient = Redis.createClient(config.port, config.host, config.options);

// share the redis client connection
module.exports = redisClient;
```

Then the modules that need a connection to the Redis server can simply do:

```
var redis = require('./redis');
redis.get('somekey', function(err, reply) {
  //...
});
```

9.4 Closure-based class

If you're going to have to have more than one instance of similar objects, you may then resolve to create a class that a) defines the common behaviour of all those objects and b) provides a way of constructing these objects. Since JavaScript doesn't have classes, we're going to explore function closures to create such a thing.

Here is an example of a logger class where you can specify the path of the log file and default log level:

closure_class_logger.js:

```
module.exports = Logger;
var extend = require('util')._extend;
var fs = require('fs');

const defaultOptions = {
  path: '/tmp/log.txt',
  defaultLevel: 'info'
};

function Logger(options) {
  var self = log;

  var opts = extend({}, defaultOptions);
  opts = extend(opts, options || {});
```

```
  var file = fs.createWriteStream(opts.path);

  function log(what, level) {
    var date = new Date;
    var entry = {
      when: date.toJSON(),
      level: level || opts.defaultLevel,
      what: what
    };

    file.write(JSON.stringify(entry) + '\n');
  };

  self.info = function(what) {
    log(what, 'info');
  };

  self.warn = function(what) {
    log(what, 'warning');
  };

  self.critical = function(what) {
    log(what, 'critical');
  };

  return self;
}
```

Here we're exporting only one function which we internally named `Logger`. This is a constructor function (hence the capital L), that:

- creates an options object (`opts`) based on the user options and the default options;
- opens the log file according to the preferred file path;
- creates the logger (`self` variable), which is, in this case, a function;
- enriches the logger with a set of methods that use the specific `log` function.

As you can see this implementation uses JavaScript closures to hide internal object states. There is no state in the `self` object itself – all is stored in variables that are declared inside the constructor, and are only accessible to functions on the same or lower scopes.

Here is an example client that instantiates two loggers, each with different options:

closure_client.js:

```
var Logger = require('./closure_class_logger');

var log1 = Logger();
var log2 = Logger({
  path: '/tmp/log2.txt',
  defaultLevel: 'warn'
});

log1('one');
log1.info('two');

log2('three');
log2.critical('four');
```

The client then instantiates each of the loggers by calling the constructor function, each time passing the logger options.

Using function closures is a good way of clearly differentiating between what is the object internal state and the object public interface, simulating what other languages define as private properties and methods. The downside is that, using this technique, you are defining a new function scope that persists throughout the life of the object, and a set of functions on that scope for each object you are creating, consuming memory and taking CPU time at initialisation. This may be fine if, like in this client example, you're not defining many instances of each object. If you want to avoid this problem, then you will need to use the prototype-based modelling capabilities that JavaScript provides.

9.5 Prototype-based class

Here is an alternative implementation of the Logger class module using JavaScript prototypes:

prototype_class_logger.js:

```
module.exports = Logger;
var extend = require('util')._extend;
var fs = require('fs');

const defaultOptions = {
  path: '/tmp/log.txt',
  defaultLevel: 'info'
};

function Logger(options) {
  if (! (this instanceof Logger)) return new Logger(options);
```

```
  var opts = extend({}, defaultOptions);
  this._options = extend(opts, options || {});
  this._file = fs.createWriteStream(opts.path);
}

Logger.prototype.log = function(what, level) {
  var date = new Date;
  var entry = {
    when: date.toJSON(),
    level: level || this._options.defaultLevel,
    what: what
  };

  this._file.write(JSON.stringify(entry) + '\n');
};

Logger.prototype.info = function(what) {
  this.log(what, 'info');
};

Logger.prototype.warn = function(what) {
  this.log(what, 'warning');
};

Logger.prototype.critical = function(what) {
  this.log(what, 'critical');
};
```

Here is a user of this module:

prototype_client.js:

```
var Logger = require('./prototype_class_logger');

var log1 = Logger();
var log2 = Logger({
  path: '/tmp/log2.txt',
  defaultLevel: 'warn'
});

log1.log('one');
log1.info('two');
```

```
log2.log('three');
log2.critical('four');
```

There are several structural differences between this implementation and the previous closure-based one:

9.5.1 State storage

In the previous example we stored the state inside the constructor function closure. Here we're storing the needed state inside the `this` object.

In JavaScript there are two basic ways for `this` to be available: 1) in the constructor function when it's invoked with the `new` keyword, or 2) when invoking a function on an object, like in `object.method(arg)`.

By convention, the state properties that are not meant to be publicly accessible are prefixed with _. These properties indicate to the programmer that they are not supported as a public interface and should never be accessed from the outside.

9.5.2 Constructor

You may have noticed that the constructor has an initial check to verify if the `this` object is an instance of the current constructor function. JavaScript constructors are meant to be called with the `new` keyword in this form:

```
var instance = new Class(args);
```

When called without the `new` keyword, a constructor is just a function with no explicit `this` bound to it. We can then verify if the `this` is correctly assigned to allow clients to use our class like this:

```
var instance = Class(args);
```

When invoked this way, this activates the following condition in the constructor, effectively replacing the current function call with a "proper" JavaScript object instantiation:

```
    if (! (this instanceof Logger)) return new Logger(options);
```

9.5.3 Functions are only declared once.

In the previous solution, for each created object we declared the method functions for each method, resulting in considerable overhead if many instances of this object are going to exist.

In a prototype-based solution, you assign the functions to the constructor prototype. When constructing a new object, the JavaScript runtime simply assigns the constructor `prototype` attribute to the new object's `__proto__` attribute.

When a method on an object is to be used, the JavaScript runtime looks into the object itself for an attribute with that name. If not found, it looks into the object's `__proto__` attribute for that attribute, and if found, that's what it uses. This allows us to only declare the functions once inside the prototype, and to allow the method onto which the method was called — the `this` object, which carries all of the state — to vary.

9.5.4 Inheritance

By using prototypical inheritance you allow your class to be extendable. For instance, if we wanted to implement a Logger base class and two implementations, one for file logging and another that only emits `log` events, you could do this:

base_logger.js:

```
module.exports = BaseLogger;
var extend = require('util')._extend;
var fs = require('fs');

const defaultOptions = {
  defaultLevel: 'info'
};

function BaseLogger(options) {
  if (! (this instanceof BaseLogger)) return new BaseLogger(options);

  var opts = extend({}, defaultOptions);
  this._options = extend(opts, options || {});
}


BaseLogger.prototype.log = function(what, level) {
  var date = new Date;
  var entry = {
    when: date.toJSON(),
    level: level || this._options.defaultLevel,
```

```
    what: what
  };

  this._log(entry);
};

BaseLogger.prototype.info = function(what) {
  this.log(what, 'info');
};

BaseLogger.prototype.warn = function(what) {
  this.log(what, 'warning');
};

BaseLogger.prototype.critical = function(what) {
  this.log(what, 'critical');
};
```

This module serves as a base class for specific Logger implementations; it relies on subclasses implementing the `_log` method. Let's see an example of a file logger that extends this `BaseLogger` class:

file_logger.js:

```
var fs = require('fs');
var inherits = require('util').inherits;

var BaseLogger = require('./base_logger');

module.exports = FileLogger;

inherits(FileLogger, BaseLogger);

const DEFAULT_PATH = '/tmp/log.txt';

function FileLogger(options) {
  if (! (this instanceof FileLogger)) return new FileLogger(options);
  BaseLogger.call(this, options);
  this._file = fs.createWriteStream(this._options.path || DEFAULT_PATH);
}

FileLogger.prototype._log = function(entry) {
  this._file.write(JSON.stringify(entry) + '\n');
};
```

This specific implementation defines a constructor named `FileLogger` that inherits from the `BaseLogger` class.

For setting up inheritance it uses Node.js's `util.inherits` utility function to set up the protoype chain correctly.

The specific constructor has to call the super-class constructor to make sure the `this` object gets constructed correctly:

```
BaseLogger.call(this, options);
```

It then relies on the private `_options` attribute that should have been set after the base constructor has been called.

Here we're seing some problems with inheritance that are not particular to JavaScript. Sub-classes may have to know implementation details of the super-class (such as the private `_options` attribute) in order to work. It also may lead to implementation dependency: if a sub-class uses a specific private attribute of the object, then the base class cannot freely use it in the future. This makes object-oriented code somewhat brittle in nature. For this reason some programmers prefer to compose objects rather than extend classes.

9.5.5 Can't export a base function

Another downside of using prototype-based classes is that, unlike we did in the closure-based approach, an instance cannot be a function; it must be an object that's implicitely created when the `new` keyword is used.

9.6 Façade module

Good practice dictates that you keep the number of lines of code per module low: one module does one thing well, should be easily comprehended by a foreign programmer, and should have good code coverage without the need to artificially alter the code or expose private functionality.

But sometimes you need to group related functions. For instance, all the functions that manage users on a given system should be somewhat centralised, making it easier to organise and identify pieces of functionality. If they're not going to fit inside one single file, you can, for instance, store them inside a folder named `models/users`. Each module exports only one function:

```
models/
├── index.js
└── users/
  ├── index.js
  ├── create.js
  ├── remove.js
  ├── update.js
  └── list.js
```

We can then define an `index.js` file that groups and exposes a set of modules. This `index.js` file would look like this in this case:

```
exports.create = require('./create');
exports.remove = require('./remove');
exports.update = require('./update');
exports.list = require('./list');
```

The users of the `models/users` module can then just require the directory path:

```
var users = require('./models/users');
```

You could then use each of the functions:

```
users.create();
users.remove();
```

10. Browserify: Node modules in the browser

Earlier on we filled this gap in JavaScript by creating a custom module system and a very simple way to concatenate module files to create a single JavaScript bundle that can be loaded and parsed at the same time on the browser.

Instead of doing this you can use a nifty tool called Browserify. Not only does Browserify make the Node.js module system available on the browser; it also allows you to use some of the Node.js core API in the browser by providing browser-specific implementations of that API.

Browserify comes with a command-line tool that takes a set of JavaScript files and outputs a bundle. When specifying the list of modules you don't need to specify the whole set of JavaScript modules. If a module depends on another module by using `require`, Browserify can understand that and also bundle that dependency in, and do that recursively until all dependencies are included in the bundle.

Here is an example taken from the Browserify README:

main.js:

```
var foo = require('./foo.js');
var bar = require('../lib/bar.js');
var gamma = require('gamma');

var elem = document.getElementById('result');
var x = foo(100) + bar('baz');
elem.textContent = gamma(x);
```

First you need to instal browserify:

```
$ npm install browserify -g
```

You can then generate a working bundle like this:

```
$ browserify main.js > bundle.js
```

And use that file in the browser:

```
<script src="bundle.js"></script>
```

Browserify is a lot more than this:

- It is able to bundle source maps for easier debugging,
- It allows for a bundle to be used externally,
- It has an API so that you can control the bundle creation from inside a Node.js script or service;
- It supports transform plugins that enable you to include file contents;
- and more…

I'm not going to do a deep dive into Browserify – that would easily be the subject of another book. Here you can see that you can use absolute or relative paths for loading modules, which means that you can use every pattern that has been described here, but in the browser!

11. Summary

Even though JavaScript did not provide a module system, it's possible to simulate one by using function closures. It's also possible to create a module system that makes dependencies explicit, and where you use names to define and require modules.

Instead of modules returning the exported value, modules can support circular dependencies by modifying an exported object.

The Node.js runtime implements a file-based module pattern that implements the CommonJS standard. Using this pattern you can create several module patterns: Singletons, closure-based classes, prototype-based classes, and façades.