Node.js By Example

Learn to use Node.js by creating a fully functional social network

Krasimir Tsonev
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Managing Assets'
- A synopsis of the book’s content
- More information on Node.js By Example

About the Author

Krasimir Tsonev is a coder with over 10 years of experience in web development. The author of Node.js Blueprints, Packt Publishing, he works with a strong focus on quality and usability. Krasimir is interested in delivering cutting-edge applications. He enjoys working in the software industry and has a passion for creating and discovering new and effective digital experiences. Right now, he is working with technologies such as HTML5/CSS3, JavaScript, PHP, and Node.js, but he originally started out as a graphic designer. Later, being a Flash developer, he spent several years using ActionScript3 and frameworks such as RobotLegs. After that, as a freelancer, he continued to deliver full-stack web services for his clients, taking care of the graphic design and frontend and backend programming. Right now, with the rise of mobile application development, Krasimir is enthusiastic about working on responsive applications that target various devices. He currently lives and works in Bulgaria. He graduated from the Technical University of Varna with both a bachelor’s and a master's degree in computer science. He loves blogging, writing books, and giving talks on the latest trends in web development.

Node.js By Example

Node.js is one of the present day's most popular technologies. Its growing community is known to produce a large number of modules every day. These modules can be used as building blocks for server-side applications. The fact that we use the same language (JavaScript) on both the server- and client-side make development fluent.

This book contains 11 chapters that contain a step-by-step guide to building a social network. Systems such as Facebook and Twitter are complex and challenging to develop. It is nice that we will learn what Node.js is capable of, but it is going to be much more interesting if we do that within a concrete context. The book covers basic phases such as the architecture and management of the assets' pipeline, and it discusses features such as users' friendship and real-time communication.

What This Book Covers

Chapter 1, Node.js Fundamentals, teaches the basics of Node.js, what stands behind the technology, and its module management system and package manager.

Chapter 2, Architecting the Project, reveals the power of build systems such as Gulp. Before starting with our social network, we will plan the project. We will talk about test-driven development and the Model-View-Controller pattern. The chapter will cover the Node.js modules that are needed to bootstrap the project.

Chapter 3, Managing Assets, covers the building of a web application. So, we have to deal with HTML, CSS, JavaScript, and images. In this chapter, we will go through the processes behind the serving of assets.

Chapter 4, Developing the Model-View-Controller Layers, is about the basic structure of our application. We will create classes for views, models, and controllers. In the next few chapters, we will use these classes as a base.

Chapter 5, Managing Users, is about implementing user registration, authorization, and profile management.

Chapter 6, Adding Friendship Capabilities, explains one of the main concepts behind modern social networks—friendship. The ability to find friends and follow their walls is an important part. This chapter is dedicated to the development of this relationship between users.

Chapter 7, Posting Content, states that the backbone of every social network is the content that users add into the system. In this chapter, we will implement the process of post making.
Chapter 8, *Creating Pages and Events*, states that providing the ability to users to create pages and events will make our social network more interesting. Users can add as many pages as they want. Other users will be able to join the newly created places in our network. We will also add code to collect statistics.

Chapter 9, *Tagging, Sharing, and Liking*, explains that besides posting and reviewing content, the users of a social network should be able to tag, share, and like posts. This chapter is dedicated to the development of these functions.

Chapter 10, *Adding Real-time Chat*, talks about the expectations of users, in today's world, to see everything that is happening right away. They want to communicate faster with each other. In this chapter, we will develop a real-time chat so that the users can send messages instantly.

Chapter 11, *Testing the User Interface*, explains that it is important to get the job done, but it is also important to cover working functionalities with tests. In this chapter, we will see how to test a user interface.
Managing Assets

The first two chapters were a good introduction to the building blocks and structures of Node.js application development. We learned about the fundamentals of the technology and revealed important patterns such as Model-View-Controller. We talked about test-driven development and REST APIs. In this chapter, we will create the base of our social network. The proper delivery and management of an application's assets is an essential part of the system. In most of the cases, it determines our workflow. We will go through the following topics in this chapter:

- Serving files with Node.js
- CSS preprocessing
- Packing client-side JavaScript
- Delivering HTML templates

Serving files with Node.js

Node.js differs from the usual Linux-Apache-MySQL-PHP setup. We have to write the server that handles the incoming request. When the user requires an image from our backend, Node.js doesn't serve it automatically. The very first file of our social network will be server.js with the following content:

```javascript
var http = require('http');
var fs = require('fs');
var path = require('path');

var files = {};
var port = 9000;
var host = '127.0.0.1';
```
Managing Assets

```javascript
var assets = function(req, res) {
    // ...
};

var app = http.createServer(assets).listen(port, host);
console.log("Listening on " + host + ":" + port);
```

We require three native modules that we will use to drive the server and deliver assets. The last two lines of the preceding code run the server and print a message to the console.

For now, the entry point of our application is the `assets` function. The main purpose of this method is to read files from the hard disk and serve it to the users. We will use `req.url` to fetch the current request path. When a web browser accesses our server and requests `http://localhost:9000/static/css/styles.css` in the browser, `req.url` will be equal to `/static/css/styles.css`. From this point onwards, we have a few tasks to handle:

- Checking whether the file exists and if not, sending a proper message (HTTP error code) to the user
- Reading the file and finding out its extension
- Sending the file's content to the browser with the correct content type

The last point is an important one. Serving files with a wrong or missing content type may cause problems. The browser may not be able to recognize and process the resource properly.

To make the process smooth, we will create a separate function for each of the tasks mentioned. The shortest one is the one that sends an error message to the user:

```javascript
var sendError = function(message, code) {
    if(code === undefined) {
        code = 404;
    }
    res.writeHead(code, {'Content-Type': 'text/html'});
    res.end(message);
}
```

By default, the value of the `code` variable is `404`, which means **Not Found**. However, there are different types of errors, such as client errors (4XX) and server errors (5XX). It is good to leave an option to change the error's code.
Let's say that we have the content of the file and its extension. We need a function that recognizes the correct content type and delivers the resource to the client. For the sake of simplicity, we will perform a simple string-to-string check of the file's extension. The following code does exactly that:

```javascript
var serve = function(file) {
  var contentType;
  switch (file.ext.toLowerCase()) {
    case "css": contentType = "text/css"; break;
    case "html": contentType = "text/html"; break;
    case "js": contentType = "application/javascript"; break;
    case "ico": contentType = "image/ico"; break;
    case "json": contentType = "application/json"; break;
    case "jpg": contentType = "image/jpeg"; break;
    case "jpeg": contentType = "image/jpeg"; break;
    case "png": contentType = "image/png"; break;
    default: contentType = "text/plain";
  }
  res.writeHead(200, {'Content-Type': contentType});
  res.end(file.content);
}
```

The `serve` method accepts a `file` object with two properties—`ext` and `content`. In the next few chapters, we will probably add more file types to the list. However, for now, serving JavaScript, CSS, HTML, JPG, and PNG images is enough.

The last task that we have to cover is the actual reading of the file. Node.js has a built-in module to read files called `fs`. We will use the asynchronous versions of its methods. With synchronous functions, the JavaScript engine may be blocked till the particular operation is fully executed. In this case, that is a reading of a file. In asynchronous programming, we allow our program to execute the rest of the code. In this scenario, we normally pass a callback—a function that will be executed when the operation ends:

```javascript
var readFile = function(filePath) {
  if (files[filePath]) {
    serve(files[filePath]);
  } else {
    fs.readFile(filePath, function(err, data) {
      if (err) {
        sendError('Error reading ' + filePath + '.');
      }
    });
  }
}
```
Managing Assets

    return; 
    
    files[filePath] = { 
        ext: filePath.split(".").pop(),  
        content: data 
    };  
    serve(files[filePath]); 
    }
    
}

The function accepts the path and opens the file. If the file is missing or there is a problem reading it, it sends an error to the user. In the beginning, we defined a files variable, which is an empty object. Every time we read a file, we are storing its content there so that the next time we read it, we don't have to access the disk again. Every I/O operation, such as reading a file, takes time. By using this simple caching logic, we improve the performance of our application. If everything is okay, we call the serve method.

Here is how you combine all the preceding snippets:

    var http = require('http'); 
    var fs = require('fs'); 
    var path = require('path'); 
    var files = {}; 
    var port = 9000; 
    
    var assets = function(req, res) { 
        var sendError = function(message, code) { ... } 
        var serve = function(file) { ... } 
        var readFile = function(filePath) { ... } 
        
        readFile(path.normalize(__dirname + req.url)); 
    } 
    
    var app = http.createServer(assets).listen(port, '127.0.0.1'); 
    console.log("Listening on 127.0.0.1:" + port); 

Every HTTP request sent to our server is processed by the assets handler. We compose the file's path, starting from the current directory. The path.normalize parameter guarantees that our string looks alright on different operating systems. For example, it does not contain multiple slashes.
CSS preprocessing

CSS preprocessors are tools that accept source and produce CSS. Very often, the input is similar to the CSS language with regard to the syntax. However, the main idea of preprocessing is to add features that are missing and, at the same time, wanted by the community. Over the past few years, CSS preprocessing has become a hot topic. It comes with lots of benefits and the concept has been warmly accepted by the community. There are two main CSS preprocessors — Less (http://lesscss.org/) and Sass (http://sass-lang.com/). Sass is based on the Ruby language and it requires more effort to run it in a Node.js project. So in this book, we are going to use Less.

In the previous chapter, we talked about building systems and task runners. CSS preprocessing and a few other tasks that we will talk about in a bit should happen automatically. Gulp seems like a good option. Let's move forward and add a `package.json` file where we will describe all the Gulp-related modules that we need:

```json
{
    "name": "nodejs-by-example",
    "version": "0.0.1",
    "description": "Node.js by example",
    "scripts": {
        "start": "node server.js"
    },
    "dependencies": {
        "gulp": "3.8.8",
        "gulp-less": "1.3.6",
        "gulp-rename": "-1.2.0",
        "gulp-minify-css": "-0.3.11"
    }
}
```

The setting of "start": "node server.js" will allow us to type `npm start` and run our server. The dependencies that we will start with are as follows:

- Gulp itself
- gulp-less: This is a plugin that wraps the Less preprocessor
- gulp-rename: This changes the name of the produced file
- gulp-minify-css: This compresses our CSS
Managing Assets

So, along with server.js, we now have package.json. We run npm install and the package manager adds a node_modules directory containing the modules. Let's define our Gulp tasks in another file named gulpfile.js:

```javascript
var path = require('path');
var gulp = require('gulp');
var less = require('gulp-less');
var rename = require('gulp-rename');
var minifyCSS = require('gulp-minify-css');

gulp.task('css', function() {
  gulp.src('./less/styles.less')
    .pipe(less({
      paths: [ path.join(__dirname, 'less', 'includes') ]
    }))
    .pipe(gulp.dest('./static/css'))
    .pipe(minifyCSS({keepBreaks:true}))
    .pipe(rename({suffix: '.min'}))
    .pipe(gulp.dest('./static/css'));
});

gulp.task('watchers', function() {
  gulp.watch('less/**/*.less', ['css']);
});

gulp.task('default', ['css', 'watchers']);
```

We start with two tasks—css and watchers. The first one expects us to have a less directory and a styles.less file inside. This will be our entry point to all the CSS styles. As seen from the Gulp task, we pipe the content of the file to the preprocessor and export the result to the static/css directory. Since everything with Gulp is a stream, we can continue and minify the CSS, rename the file to styles.min.css, and export it to the same folder.

We do not want to run the building processes by ourselves every time we make changes to a file. So, we register watchers for the files in the less folder. A watcher is a process that monitors specific files and notifies the rest of the system once these files are changed.
At the end of this step, our project looks like this:

```
- less
  - styles.less
- node_modules
- static
  - css
    - styles.css
    - styles.min.css
- gulpfile.js
- package.json
- server.js
```

### Packing client-side JavaScript

As with CSS, our goal should be to serve only one JavaScript file to the client's browser. We do not want to force the user to make more than one request, because this is less efficient and it means that the web browser takes longer to process and display the content of the page. Nowadays, the client-side part of applications is fairly complex. As with complex systems, we split our logic into different modules. Often, different modules mean different files. Thankfully, Node.js is full of tools that can be used to pack JavaScript. Let's see two of the most popular tools.

### Concatenating with Gulp

Gulp, as a build system, has several modules to concatenate files. The one that we are interested in is called gulp-concat. Let's add it to the package.json file:

```json
"dependencies": {
  "gulp": "3.8.8",
  "gulp-less": "1.3.6",
  "gulp-rename": "1.2.0",
  "gulp-minify-css": "0.3.11",
  "gulp-concat": "2.4.1"
}
```
Managing Assets

The next step is to write a task that uses it. Again, we will use the src and dest Gulp methods, and in between is the concatenation:

```javascript
var concat = require('gulp-concat');

gulp.task('js', function() {
    gulp.src('./js/*.js')
        .pipe(concat('scripts.js'))
        .pipe(gulp.dest('./static/js'))
});
```

It's important to mention that the files will be added to the final file in alphabetical order. So, we should be careful whenever there are some code dependencies. If this is the case, we should name the files in such a way that their names start with a unique number—01, 02, 03, and so on.

The next logical task that we will do is to minify our JavaScript. Like the Less compilation, we want to serve a file that is as small as possible. The module that will help us achieve this is gulp-uglify. Again, we should add it to the package.json file ("gulp-uglify": "1.0.1"). After this, a little tweak to our newly created task will minify the JavaScript:

```javascript
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');

gulp.task('js', function() {
    gulp.src('./js/*.js')
        .pipe(concat('scripts.js'))
        .pipe(gulp.dest('./static/js'))
        .pipe(uglify())
        .pipe(rename({suffix: '.min'}))
        .pipe(gulp.dest('./static/js'))
});
```

Note that we used the gulp-rename plugin again. This is necessary because we want to produce a different file.

Modularity in the browser with RequireJS

While building software, one of the most important concepts to think about is the splitting of our system into modules. Node.js has a nice built-in system to write modules. We mentioned this in Chapter 1, Node.js Fundamentals. We encapsulate our code in a single file and use module.exports or exports to create the public API. Later, via the require function, we access the created functionalities.
However, for the client-side JavaScript, we do not have such a built-in system. We need to use an additional library that allows us to define modules. There are several possible solutions. The first one that we will take a look at is RequireJS (http://requirejs.org/). We will download the library (version 2.1.16) from the official site and include it in our page like this:

```html
<script data-main="scripts/main" src="scripts/require.js"></script>
```

The key attribute here is `data-main`. It tells RequireJS about our application's entry point. In fact, we should have the `scripts/main.js` file in our project's folder to get the preceding line working. In `main.js`, we can use the `require` global function:

```javascript
// scripts/main.js
require(['modules/ajax', 'modules/router'], function(ajax, router) {
  // ... our logic
});
```

Let's say that our code in `main.js` depends on two other modules—the Ajax wrapper and router. We describe these dependencies in an array and provide a callback, which is later executed with two parameters. These parameters are actually references to the necessary modules.

The defining of modules is possible with the help of another global function—`define`. Here is how the Ajax wrapper looks:

```javascript
// modules/ajax.js
define(function () {
  // the Ajax request implementation
...
  // public API
  return {
    request: function() { ... }
  }
});
```

By default, behind the scenes, RequireJS resolves the dependencies asynchronously. In other words, it performs an HTTP request for every required module. In some cases, this may lead to performance issues because every request takes time. Thankfully, RequireJS has a tool (optimizer) that solves the problem. It can bundle all the modules into a single file. The tool is available for Node.js too and it is distributed with the `requirejs` package:

```
npm install -g requirejs
```
Managing Assets

After a successful installation, we will have the r.js command in our terminal. The basic call looks like this:

```bash
// in code_requirejs folder
r.js -o build.js
```

As with Grunt and Gulp, we have a file that instructs RequireJS on how to work. The following is a snippet that covers our example:

```javascript
// build.js
({
  baseUrl: ".",
  paths: {},
  name: "main",
  out: "main-built.js"
})
```

The `name` property is the entry point and `out` is the resulting file. It's nice that we have the `paths` property available. It is a place where we can describe the modules directly; for example, `jquery: "some/other/jquery"`. Later in our code, we do not have to write the full path to the files. Just a simple `require(['jquery'], ...)` is enough.

By default, the output of the `r.js` command is minified. If we add an `optimize=none` argument to the command in the terminal, we will get the following:

```javascript
// main-built.js
define('modules/ajax',[],function () {
  ...
});

define('modules/router',[],function () {
  ...
});

require(['modules/ajax', 'modules/router'], function(ajax, router) {
  ...
});
define("main", function(){
```

The `main-built.js` file contains the main module and its dependencies.
Moving from Node.js to the browser with Browserify

RequireJS indeed solves the problem with modularity. However, it makes us write more code. Also, we should always describe our dependencies by following a strict format. Let's look at the code that we used in the previous section:

```javascript
require(['modules/ajax', 'modules/router'], function(ajax, router) {
    ...
});
```

It is indeed better if we use the following code:

```javascript
var ajax = require('modules/ajax');
var router = require('modules/router');
```

The code is much simpler now. This is how we should fetch a module in the Node.js environment. It would be nice if we could use the same approach in the browser.

Browserify (http://browserify.org/) is a module that brings the require module of Node.js to the browser. Let's install it first by using the following code:

```
npm install -g browserify
```

Similarly, to illustrate how the tool works, we will create the main.js, ajax.js and router.js files. This time, we are not going to use a global function such as `define`. Instead, we will use the usual Node.js `module.exports`:

```javascript
// main.js
var ajax = require('./modules/ajax');
var router = require('./modules/router');

// modules/ajax.js
module.exports = function() {};

// modules/router.js
module.exports = function() {};
```

By default, Browserify comes as a command-line tool. We need to provide an entry point and an output file:

```
browserify ./main.js -o main-built.js
```
Managing Assets

The result in the compiled file is as follows:

```javascript
// main-built.js
(function e(t,n,r){function s(o,u){if(!n[o]){if(!t[o]){var a=typeof require=='function'&&require;if(!u&&a)return a(o,!0);if(i)return i(o,!0);var f=new Error("Cannot find module "+'"'+o+'"""""");throw f.code="MODULE_NOT_FOUND",f}var l=n[o]={exports:{}};t[o][0].call(l.exports,function(e){var n=t[o][1][e];return s(n?n:e)},l,l.exports,e,t,n,r)}return n[o].exports}var i=typeof require=='function'&&require;for(var o=0;o<r.length;o++)s(r[o]);return s})({1:[function(require,module,exports){
  var ajax = require('./modules/ajax');
  var router = require('./modules/router');
},"./modules/ajax":2,"./modules/router":3}),2:[function(require,module,exports){
  module.exports = function() {};
},"./modules/ajax":2
}),3:[function(require,module,exports){
  module.exports=require(2)
},".../modules/ajax.js":2])
```

Note that along with the modules, the compiled file also contains the require function's definition and implementation. It's really just a few bytes of code that makes Browserify one of the most popular ways to deliver modular JavaScript in the browser. This is what we are going to use in the next few chapters.

We have started a Gulp setup. Let's add Browserify there. We have already made a concatenation of the JavaScript. Let's replace it with Browserify. We will add the module to the package.json file as follows:

```json
"dependencies": {
  "gulp": "3.8.8",
  "gulp-less": "1.3.6",
  "gulp-rename": "1.2.0",
  "gulp-minify-css": "0.3.11",
  "gulp-concat": "2.4.1",
  "gulp-uglify": "1.0.1",
  "gulp-browserify": "0.5.0"
}
```
After running `npm install`, we will get the plugin installed and ready to use. We need to make two changes, replacing `concat` with `browserify` and pointing out the application's main file:

```javascript
var browserify = require('gulp-browserify');
var uglify = require('gulp-uglify');

gulp.task('js', function() {
    gulp.src('./js/app.js')
    .pipe(browserify())
    .pipe(gulp.dest('./static/js'))
    .pipe(uglify())
    .pipe(rename({suffix: '.min'}))
    .pipe(gulp.dest('./static/js'))
});
```

Now, the `src` method accepts only one file. It's our entry point. This is the place where Browserify starts resolving dependencies. The rest is the same. We still use `uglify` for minification and `rename` to change the file's name.

## Delivering HTML templates

In the previous sections, you saw how you can package CSS and JavaScript for the browser. At the end of this chapter, we will explore the various ways to deliver HTML. In the context of client-side applications, the templates still contain HTML. However, we need a dynamic way to render and fill them with data.

### Defining the templates in script tags

The Ember.js framework adopts the concept of adding HTML templates directly into the page by using the popular `handlebars` ([http://handlebarsjs.com/](http://handlebarsjs.com/)) template engine. However, since we do not want to mess up the markup that is already there, we place them in the `<script>` tags. The good thing about this is that if we set a custom value of the `type` attribute, the browser does not process the code inside. Here's a demonstration of this:

```html
<script type="text/x-handlebars" id="my-template">
    <p>Hello, <strong>Hello!</strong></p>
</script>
```
Managing Assets

Since the tag has an id attribute, we can get its content easily in the following way:

```javascript
var template = document.querySelector('#my-template').innerHTML;
```

The benefit of this technique is that the template is on the page and we have instant access to it. Also, templates only display the desired content after being processed by JavaScript. So, if JavaScript is not enabled in the browser, we do not want to display the unprocessed raw template. A major problem with this concept is the fact that we will flood our HTML page with a lot of code. If we have a big application, then the user will have to download all the templates even if he/she uses only a part of it.

Loading the template externally

It's also a common practice to define the templates as external files and load them on the page with an Ajax request. The following pseudocode uses jQuery's get method to do the job:

```javascript
$.get('/templates/template.html', function(html) {
    // ...
});
```

We have clear markup, but the user has to make an additional HTTP request in order to fetch the template. This approach makes the code more complex because the process is asynchronous. It also makes the processing and rendering of the content slower than the preceding method.

Writing HTML inside the JavaScript

With the rise of mobile applications, many big companies have started developing their own frameworks. Since these companies have enough resources, they usually produce something interesting. For example, Facebook created a framework called React (http://facebook.github.io/react/). It defines its templates directly in the JavaScript as follows:

```html
<script type="text/jsx">
 var HelloMessage = React.createClass({
    render: function() {
        // Note: the following line is invalid JavaScript,
        // and only works using React parser.
        return <div>Hello {this.props.name}</div>;
    }
  });
</script>
```
The developers from Facebook adopted the first technique mentioned in this section. They put some code inside a `<script>` tag. In order to get things working, they have their own parser. It processes the script and converts it into valid JavaScript.

There are solutions that do not have templates in the form of HTML. There are tools that use templates written in JSON or YAML. For example, **AbsurdJS** (http://absurdjs.com/) can keep its template inside the JavaScript class definition as follows:

```javascript
body: {
  'section.content#home': {
    nav: [
      { 'a[href="#" class="link"]': 'A' },
      { 'a[href="#" class="link"]': 'B' },
      { 'a[href="#" class="link"]': 'C' }
    ]
  },
  footer: {
    p: 'Text in the Footer'
  }
}
```

**Precompiling templates**

Another popular way to deliver templates to the client side is by using precompilation. This is what we are going to use in our project. Precompilation is a process that converts the HTML template to a JavaScript object, which is ready for use in our code. This approach has several benefits, some of which are as follows:

- We do not have to think about accessing the HTML template
- The markup is still separated from the JavaScript code
- We do not lose time in fetching and processing the HTML

Different client-side frameworks have different tools to precompile templates. We will cover this in detail later, but the instrument that we are going to use for our social network application is called Ractive.js (http://www.ractivejs.org/). It's a client-side framework that was originally developed by the people at TheGuardian to produce a news application. It's cross-browser and it performs well on mobile devices.
Managing Assets

In order to transform our HTML into Ractive-precompiled templates, we need two new modules in the `package.json` file:

```
"ractive": "0.6.1",
"gulp-tap": "0.1.3"
```

The `gulp-tap` plugin allows us to process every file sent to the Gulp's pipeline. Here is the new task that we have to add to the `gulpfile.js` file:

```
var Ractive = require('ractive');
var tap = require('gulp-tap');

gulp.task('templates', function() {
  gulp.src('./tpl/**/*.html')
    .pipe(tap(function(file, t) {
      var precompiled = Ractive.parse(file.contents.toString());
      precompiled = JSON.stringify(precompiled);
      file.contents = new Buffer('module.exports = ' + precompiled);
    }))
    .pipe(rename(function(path) {
      path.extname = '.js';
    }))
    .pipe(gulp.dest('./tpl'))
});
```

```
gulp.task('default', ['css', 'templates', 'js', 'watchers']);
```

Ractive.parse returns the precompiled template. Since it is a JavaScript object, we use `JSON.stringify` to convert it to a string. We use Browserify to control our client-side modularity so `module.exports` is attached in front of the template's code. In the end, we use `gulp-rename` and produce a JavaScript file.

Let's say that we have a `/tpl/template.html` file with the following content:

```
<section>
  <h1>Hello {{name}}</h1>
</section>
```

When we run the gulp command, we will receive `/tpl/template.js`, which contains the JavaScript that is equivalent to the preceding markup:

```
module.exports = {
  "v": 1,
  "t": [{
    "t": 7,
    "e": "section",
    "f": [{
      "t": 7,
      "e": "h1",
      "f": [{
        "Hello",
        "t": 2,
        "r": "name"}]
    }]
  }]
}
```

It probably looks strange now, but in the next few chapters, you will see how you can use such templates.
Summary

Assets are a major part of web applications. Often, companies do not pay enough attention to this part, which leads to slower loading time and increased web hosting costs, especially when your site grows in popularity. In this chapter, we saw that it is important to find the right setup and deliver the images, CSS, JavaScript, and HTML in the most efficient way.

In the next chapter, we will start working heavily on our social network. We will explore the world of the Model-View-Controller pattern.
Where to buy this book
You can buy Node.js By Example from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.