Extracted from:

Node.js 8 the Right Way

Practical, Server-Side JavaScript That Scales

This PDF file contains pages extracted from *Node.js 8 the Right Way*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2018 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

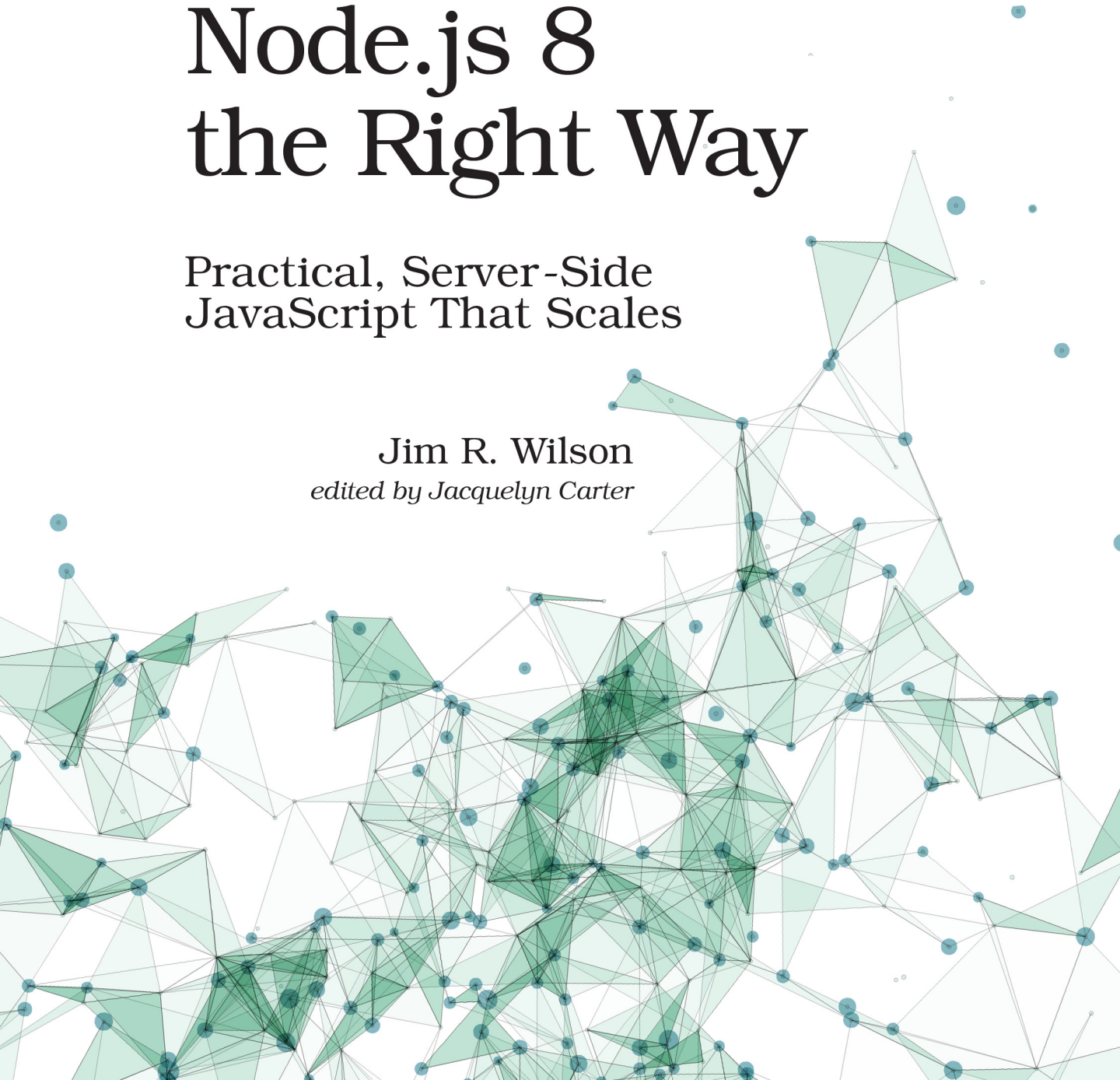
The Pragmatic Programmers

Node.js 8 the Right Way

Practical, Server-Side JavaScript That Scales

Jim R. Wilson

edited by Jacquelyn Carter



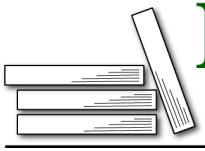
Node.js 8 the Right Way

Practical, Server-Side JavaScript That Scales

Jim R. Wilson

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Indexing: Potomac Indexing, LLC
Copy Editor: Candace Cunningham
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2018 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-68050-195-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2018

Behavior-Driven Development with Mocha and Chai

It's a maxim of programming that testing is good for code health. One strategy for approaching testing is BDD, which advocates articulating your expected behaviors in tests even before you start writing the implementation.

Not all programming problems are equally well suited to the BDD approach, but data processing is one area where it makes a lot of sense. Since the inputs and outputs of the program are quite well defined, you can specify the desired behavior to a high degree even before implementing the functionality.

So in this chapter, let's take the opportunity to use BDD while parsing the RDF content. Along with Mocha, last seen back in *Developing Unit Tests with Mocha*, on page ?, we'll use Chai, a popular assertion library.

Of course, you can use Mocha, or any test framework, without an assertion library, so why use one? The answer comes down to expressiveness. Using an assertion library like Chai, you can express your test conditions in a way that's more readable and maintainable.

Chai supports several different styles of assertions, each with their own pros and cons. Here I'll show you how to use Chai's *expect* style. It offers a readable syntax without too much magic, making it a good balance between the other two Chai styles: *assert* and *should*.

I'll give you an example, first using Node.js's built-in `assert` module, and the same using Chai's expect style:

```
assert.ok(book.authors, 'book should have authors');
assert.ok(Array.isArray(book.authors), 'authors should be an array');
assert.equal(book.authors.length, 2, 'authors length should be 2');
```

If you read the code carefully, you can deconstruct that it's confirming that the `book` object has a property called `authors` that is an `Array` of length 2. By contrast, check out this example using Chai's `expect()` method:

```
expect(book).to.have.a.property('authors')
  .that.is.an('array').with.lengthOf(2);
```

By comparison to the native Node.js assert code, this reads like poetry. Let's get Mocha and Chai set up for this project, then we'll dig further into how to make good use of `expect()`.

Setting Up Tests with Mocha and Chai

First you need to install Mocha and Chai as development dependencies. Open a terminal to your databases project directory and create a minimal package.json, then use npm to install Mocha and Chai.

```
$ cd databases
$ npm init -y
$ npm install --save-dev --save-exact mocha@2.4.5 chai@3.5.0
```

Next, open your package.json for editing. Find the scripts section and update the test entry to look like this:

```
"scripts": {
  "test": "mocha"
}
```

The test entry invokes Mocha with all its default arguments. This provides good output for running all the unit tests once with npm test.

By default, Mocha will look for a directory called test in which to find tests to execute. Create a test dir now.

```
$ mkdir test
```

Now let's run npm test and see what happens. Open a terminal to your databases project directory and give it a try.

```
$ npm test

> @ test ./databases
> mocha


  0 passing (2ms)
```

Unsurprisingly, Mocha successfully runs, but since we have no tests we see 0 passing. Let's change that by adding a test.

Declaring Expectations with Chai

Now that Mocha is ready to run your tests, open a text editor and enter the following:

databases/test/parse-rdf-test.js

```
'use strict';

const fs = require('fs');
const expect = require('chai').expect;

const rdf = fs.readFileSync(`${__dirname}/pg132.rdf`);
```

```
describe('parseRDF', () => {
  it('should be a function', () => {
    expect(parseRDF).to.be.a('function');
  });
```

Save this file as parse-rdf-test.js in your databases/test directory.

Stepping through this code, first we pull in the fs module and Chai's expect function. expect() will be the basis of our assertions going forward.

Next, we load *The Art of War*'s RDF content via fs.readFileSync(). Most of the time in Node.js, you want to avoid synchronous I/O, but in this case the proper execution of the test absolutely depends on it loading, so it's OK.

With the setup out of the way, we use Mocha's describe() and it() functions to declare behavior we want from the as-yet-unwritten parseRDF function. Using expect(), we can state our requirements in a very sentence-like form. So far all we require is that parseRDF is a function, but we'll add more requirements soon.

Before you run npm test again, you will need to copy pg132.rdf into the test directory. Run the following command from your databases project directory to take care of it:

```
$ cp ../data/cache/epub/132/pg132.rdf test/
```

Now we're ready to run npm test again.

```
$ npm test

> @ test ./databases
> mocha


  parseRDF
    1) should be a function

  0 passing (11ms)
  1 failing

  1) parseRDF should be a function:
     ReferenceError: parseRDF is not defined
      at Context.it (test/parse-rdf-test.js:14:12)


npm ERR! Test failed.  See above for more details.
```

As expected, we now have one failing test. Since parseRDF is not defined, it cannot be a function.

Now we're ready to create the parse-rdf library, which will define the function and make the test pass.

Developing to Make the Tests Pass

To start, create a new directory called lib in your databases project directory. Putting your modules in a lib directory is a strong convention in the Node.js community.

Open your text editor and enter the following:

databases/lib/parse-rdf.js

```
'use strict';

module.exports = rdf => {
};
```

Save this file as parse-rdf.js in your databases/lib directory.

At this point, all the library does is assign a function to module.exports. This function takes a single argument called rdf and returns nothing, but it should be good enough to make the test pass, so let's pull it in and find out.

Back in your parse-rdf-test.js, add a require() to the top of the file—right after Chai's require line—to pull in the module you just created:

databases/test/parse-rdf-test.js

```
const parseRDF = require('../lib/parse-rdf.js');
```

Here we're taking the anonymous function that we assigned to module.exports in parse-rdf.js and placing it in a constant called parseRDF.

After you save the file, rerun npm test.

```
$ npm test

> @ test ./databases
> mocha


  parseRDF
    ✓ should be a function

  1 passing (8ms)
```

Great! With our test harness in place, we're now in position to iteratively improve the RDF parser library as we add tests and then make them pass.

Enabling Continuous Testing with Mocha

Head back to your parse-rdf-test.js file; it's time to add some more assertions. Insert the following into the describe() callback after our original it('should be a function'):

databases/test/parse-rdf-test.js
```
it('should parse RDF content', () => {
  const book = parseRDF(rdf);
  expect(book).to.be.an('object');
});
```

This test asserts that when we call the parseRDF function, we get back an object. Since the function currently returns nothing (undefined), it's no surprise that our test now fails when we run npm test:

```
$ npm test

> @ test ./databases
> mocha


  parseRDF
    ✓ should be a function
    1) should parse RDF content

  1 passing (45ms)
  1 failing

  1) parseRDF should parse RDF content:
     AssertionError: expected undefined to be an object
      at Context.it (test/parse-rdf-test.js:24:24)


npm ERR! Test failed.  See above for more details.
```

No problem—to make the test pass, we just have to add code to the parseRDF function so it will create and return an object. In your parse-rdf.js file, expand the exported function to this:

databases/lib/parse-rdf.js
```
module.exports = rdf => {
  const book = {};
  return book;
};
```

And now when you run npm test, it should pass again:

```
$ npm test

> @ test ./databases
> mocha


  parseRDF
    ✓ should be a function
    ✓ should parse RDF content (38ms)

  2 passing (46ms)
```

You may have noticed by now that by going back and forth between the test and the implementation, we have established a pretty strong development pattern:

1. Add new criteria to the test.
2. Run the test and see that it fails.
3. Modify the code being tested.
4. Run the test and see that it passes.

We can significantly speed up the running of the tests in steps 2 and 4 above by running the tests continuously, rather than having to invoke `npm test` from the command line each time.

Mocha can help us here. When you invoke Mocha with the `--watch` flag, it will continuously monitor any files ending in `.js` that it can find, and then rerun the tests whenever they change.

Let's add another script to our `package.json` to run Mocha in this way:

```
"scripts": {
  "test": "mocha",
  "test:watch": "mocha --watch --reporter min"
}
```

Now, instead of executing `npm test` to run the tests just once, you can use `npm run test:watch` to begin continuous monitoring. By using the `--reporter min` option, every time Mocha runs it will clear the screen and provide only minimal output for passing tests. Failing tests will still show their full output.

Try this command in your terminal:

```
$ npm run test:watch
```

If everything is working properly, the terminal screen should be cleared and you should see only this:

```
2 passing (44ms)
```

Mocha has a number of other built-in reporters with various pros and cons that you can explore on the Mocha website.[2]

With Mocha still running and watching files for changes, let's begin another round of development.

2. https://mochajs.org/#reporters